Casper

A Compiler for the Analysis of Security Protocols

User Manual and Tutorial

Gavin Lowe
Tom Gibson-Robinson
Philippa Broadfoot
Christopher Dilloway
Mei Lin Hui

Version 2.0

March 17, 2015

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, UK

Contents

1 Introduction

Many security protocols have appeared in the academic literature, with various goals, such as establishing a cryptographic key, or achieving authentication (where an agent becomes sure of the identity of the other agent taking part in the protocol). These protocols are supposed to succeed even in the presence of a malicious agent, called an *intruder*; this intruder is assumed to have complete control over the communications network, and so can intercept messages, and introduce new messages into the system using information from messages he has previously seen. Unfortunately, a large proportion of the protocols that have been suggested do not succeed in their stated goals!

Over the last few years, a method for analyzing security protocols using the process algebra CSP [Hoa85] and its model checker FDR [Ros94] has been developed [Ros95, Low96, LR97]. Briefly:

- Each agent taking part in the protocol is modelled as a CSP process;
- The most general intruder who can interact with the protocol is also modelled as a CSP process;
- The resulting system is tested against specifications representing desired security properties, such as "correctly achieves authentication", or "ensures secrecy"; FDR searches the state space to investigate whether any insecure traces (i.e. sequences of messages) can occur;
- If FDR finds that the specification is not met then it returns a trace of the system that does not satisfy the specification; this trace corresponds to an attack upon the protocol.

This method has proved remarkably successful, and has been applied to find attacks upon a number of protocols [Low96, LR97]. However, the task of producing the CSP description of the system is very time-consuming, and only possible for people well practiced in CSP—and even the experts will often make mistakes that prove hard to spot.

Casper simplifies this process. The user specifies the protocol using a more abstract notation, similar to the notation appearing in the academic literature, and Casper compiles this into CSP code, suitable for checking using FDR.

This version of the manual assumes some familiarity with FDR, at least to the level covered by the first three parts of the FDR2 manual [For97].

In the next section we give a gentle introduction to the Casper syntax by informally explaining an example input file. In Section 3 we describe how to use Casper to compile the CSP code from the input file, how to run the

code using FDR, and how to interpret the output. In Section 4, we present a case study of the Wide Mouthed Frog protocol. In Section 5 we describe how to model some more features that often crop up in security protocols. In Section 9 we describe an extension to Casper that supports the simplifying transformations of [HL99]. We sum up in Section 10.

2 An example input file

An example input file is given in Appendix A and is included in the file `NS3.spl` in the standard Casper distribution and in the Casper example library [Cas98]. This file gives a definition of the 3 message version of the Needham-Schroeder Public Key Protocol [NS78]. This protocol would normally be described as in Figure 1, where $PK(A)$ represents A's public key, and $\{m\}_k$ represents message m encrypted with key k.

$$
\begin{array}{lll}
1. & A \rightarrow B & : \; A, B, \{na, A\}_{PK(B)} \\
2. & B \rightarrow A & : \; B, A, \{na, nb\}_{PK(A)} \\
3. & A \rightarrow B & : \; A, B, \{nb\}_{PK(B)}
\end{array}
$$

Figure 1: The Needham-Schroeder Public Key Protocol

In the protocol, A seeks to establish a connection with B, and to achieve mutual authentication (both agents should become sure as to the identity of the other agent). The protocol begins with A selecting a nonce na, and sending it along with its identity to B (message 1), encrypted using B's public key. When B receives this message, it decrypts the message to obtain the nonce na. It then returns the nonce na, along with a new nonce nb, to A, encrypted with A's key (message 2). When A receives this message he should be assured that he is talking to B, since only B should be able to decrypt message 1 to obtain na. A then returns the nonce nb to B, encrypted with B's key. When B receives this message, it would seem that he should be assured that he is talking to A, since only A should be able to decrypt message 2 to obtain nb.

We consider a protocol description in "standard notation" to be a template, parameterized by the free variables appearing within it; these variables are instantiated with actual values in runs, normally different values in different runs. We adopt the following conventions in this paper: free variables representing agents' identities will be represented by a single capital letter while free variables representing other data items will be represented by small letters; when we instantiate these free variables with actual values, we will

represent agents' identities by proper names while other data items will have names beginning with a capital letter.

In defining the Casper input format, we have had to make various design decisions. Our philosophy has been to try to make the syntax as close as possible to the standard way of describing protocols, with as few special cases as possible. We have tried to define a set of default actions (made precise below) that are as close as possible to what would be expected by somebody familiar with security protocols: for example, when an agent receives a message containing a variable that he has seen in an earlier message, he should check that the value he receives matches the value he saw earlier; the protocol tester should not have to make this test explicit.

On the other hand, we have not tried to cover cases where the actions to be taken by the agents are not obvious; to do so would be to run the risk of having the agents perform actions different from those expected by the protocol tester. In such cases, the actions should be made explicit; failure to do so should cause Casper to raise an error message.

2.1 Overview of input file

FDR operates by explicitly enumerating and then exploring the state space of the system in question. Hence, it can only deal with finite state systems—typically up to a few million states—and so this method can only be used to check a *particular* (typically fairly small) system running the protocol, for example, with a single initiator and a single responder, rather than being able to check an *arbitrary* system with an arbitrary number of initiators and responders; similarly, FDR can only deal with systems where the underlying atomic datatypes—for example, the types of nonces or keys—are themselves finite.

It is my experience that most attacks upon protocols can be found by considering a fairly small system. (The question of proving the security of an arbitrarily sized system from the security of a particular system is the topic of current research; see [Low96, Low98] for approaches to this problem.)

Because of the above observations, the Casper input file must define not only the operation of the protocol, but also the system to be checked. Therefore, the input file contains two distinct parts:

- A definition of the way in which the protocol operates, describing the messages passed between the agents, the tests performed by the agents, the types of the data items used, the initial knowledge of the agents, a specification of what the protocol is supposed to achieve, and a definition of any algebraic equivalences over the types used.

- A definition of the actual system to be checked, defining the agents taking part in the actual system and the roles they play, the actual datatypes to be used, and the intruder's abilities.

The first part can be thought of as a function that returns a model of a system running the protocol; the second part can then be thought of as defining a particular image of that function, by instantiating the parameters of the protocol.

The two parts are themselves further subdivided by section headers; these are lines beginning with "`#`". Comments may be added to the file by beginning the relevant lines with "`--`". Any logical line may be split across two or more physical lines by preceding any non-logical linebreak by a backslash ("\").

2.2 The protocol definition

The first part of the input file defines the generic operation of the protocol.

2.2.1 Protocol description

The main part of the definition of the protocol is the definition of the sequence of messages in the protocol. This appears under the heading `#Protocol description`.

The notation used is similar to the standard method of describing protocols, as in Figure 1. However, we have to adopt the notation slightly to make the definition fully formal.

In order to represent the protocol in ascii, we use the notation `{m}{k}` for message m encrypted with key k, denoted by $\{m\}_k$ in Figure 1. Thus the three messages can be represented by:

```
1.  A -> B : {na, A}{PK(B)}
2.  B -> A : {na, nb}{PK(A)}
3.  A -> B : {nb}{PK(B)}
```

Casper adds extra fields to each message, representing the (apparent) sender and (intended) receiver, so there is no need to include these explicitly in the normal case where the intruder can be expected to know all the agents' identities (but see Section 5.8 for a discussion of this point).

We also need some way of starting the protocol run off. How does A know that he should run the protocol with B? We assume that the run is initiated by A receiving some message from a user, or the environment, including B's identity. We represent this by an extra message in the protocol description:

```
0.    -> A : A, B
```

The absence of a sender field in the above line represents that this message is sent by the environment. We assume that such messages cannot be overheard by the intruder; neither can they be faked.

The complete protocol description then takes the form:

```
#Protocol description
0.    -> A : B
1.  A -> B : {na, A}{PK(B)}
2.  B -> A : {na, nb}{PK(A)}
3.  A -> B : {nb}{PK(B)}
```

2.2.2 Free variables

The types of the variables and functions that are used in the protocol definition are defined under the heading "`#Free variables`"; this definition takes the following form:

```
#Free variables
A, B : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
```

For example, in the protocol description, the variables *na* and *nb* should be taken to be of type *Nonce*. This information is used in defining the types of valid messages in the system. The functions *PK* and *SK* return an agent's public key and secret key, respectively; these functions will be defined later.

We term these "free variables" because they will be instantiated with actual values when we consider an actual system running the protocol.

Under the same heading is a definition of which keys are inverses of which others:

```
InverseKeys = (PK, SK)
```

The above line means that the functions PK and SK, when applied to the same identity, return keys that are inverses of each other; so for every agent A, $PK(A)$ (A's public key) and $SK(A)$ (A's secret key) are inverses of one another.

2.2.3 Processes

Each agent running in the system will be represented by a CSP process. The names of the CSP processes representing the agents are defined as below:

```
#Processes
INITIATOR(A,na) knows PK, SK(A)
RESPONDER(B,nb) knows PK, SK(B)
```

These lines have several tasks:

- They give names to the roles played by the different agents (here *INITIATOR* and *RESPONDER*). These names are also used as the names of the CSP processes that represent the agents. For example, the agent represented by A in the protocol description will be represented by a CSP process *INITIATOR*, parameterized by the arguments A and na.

- The parameters and the variables following the keyword "`knows`" define the knowledge that the agent in question is expected to have at the beginning of the protocol run. For example, the initiator A is expected to know his own identity A, the nonce na, the public key function PK (i.e. he can look up public keys in some table), and his own secret key $SK(A)$. The first parameter should be the string that represents the agent in the protocol description (section 2.2.1), and will be instantiated with the agent's identity.

  This information is used to check that the protocol definition is feasible, in the sense that agents only send messages that they could be expected to create and only receive messages that they could be expected to decrypt.

- Later (under the `#System` heading, below), we will instantiate the parameters with actual values, so as to define a system and define the data values that each agent should use in their runs. For example, suppose we define a system with a particular initiator Alice: that is we instantiate the variable A with the value *Alice*. Then we would expect that in each run, Alice would use a different nonce in each run: that is na would be instantiated with different values in each run. On the other hand, the data items PK and $SK(A)$ do not appear as parameters, because we would expect the same values to be used in every run ($SK(A)$ depending upon the identity A).

Whenever an agent sends a message, it should be the case that the agent possesses all the components necessary to produce it; for example, B is able to send message 2 because he knows na (learnt from message 1), nb (from his parameter list), and $PK(A)$ (from his knowledge of PK and A). Similarly, if it is the intention that an agent should decrypt an encrypted component that he receives, then he should possess the decrypting key.

An agent will accept a message it receives if all fields represented by variables already in the agent's knowledge contain the expected values; for example, in message 2 above, A will accept any value for nb, but will only accept the value for na that matches the value in his current knowledge (that is, the same value that A sent in message 1), and will only accept a message that is encrypted with $PK(A)$ (his public key).

2.2.4 Specifications

The requirements of the protocol are specified under the `#Specification` heading, as below:

```
#Specification
Secret(A, na, [B])
Secret(B, nb, [A])
Agreement(A,B,[na,nb])
Agreement(B,A,[na,nb])
```

The lines starting `Secret` specify that certain data items should be secret. The first secret specification above may be paraphrased as:

> A thinks that na is a secret that can be known to only himself and B.

Of course, if B happens to be the intruder then there is nothing to prevent him passing the secret on to others. However, this line will cause a CSP specification to be generated with the meaning: if A runs the protocol, apparently with B, and B is not the intruder, then the intruder will never learn the value of na.

The lines starting `Agreement` are authentication specifications. The first specifies that A is correctly authenticated to B, and the two agents agree on the data values na and nb. More formally, it specifies:

If:

> responder B completes a protocol run, apparently with A, using the data values na and nb

then:

> the same agent A has previously been running the protocol, apparently with B, with A taking the role of initiator, using the same nonces; and further each such run of B corresponds to a *unique* run of A.

This is a particular form of authentication; we discuss various other forms in Section 5.4.

2.3 The system definition

The second part of the input file deals with the actual system to be checked.

2.3.1 Type definitions

The types of variables to be used in the actual system to be checked are defined in a similar way to the types of the free variables:

```
#Actual variables
Alice, Bob, Mallory : Agent
Na, Nb, Nm : Nonce
```

Thus we will be dealing with a system with three agents (*Mallory* will be the intruder), and three nonces; the public and secret keys of these agents are defined in the `#Functions` section, below.

I normally use the convention that free variables representing agents are denoted by a single capital letter (A, B, etc.) while actual variables representing agents are denoted by real names (Alice, Bob, etc.). Similarly, free variables representing other data items are denoted by small letters (e.g. na) while the corresponding actual values are denoted by identifiers starting with a capital letter (e.g. Na).

2.3.2 Functions

Any functions used by the agents in the protocol description have to be defined under the `#Functions` heading:

```
#Functions

symbolic PK, SK
```

The above defines the functions PK (which returns an agent's public key) and SK (which returns an agent's secret key) to be *symbolic*: this means that Casper produces its own values to represent the results of function applications.

2.3.3 System definition

The most important part of the system definition covers which agents should be present in the system to be checked. This is done by listing the agents, with the parameters suitable instantiated, as follows:

```
#System
INITIATOR(Alice, Na)
RESPONDER(Bob, Nb)
```

Thus we consider a system with a single initiator, Alice (taking the role of A in the protocol description), and a single responder, Bob, who can each run the protocol once; they use nonces Na and Nb. The types of the parameters of the processes should match the types of the parameters of the corresponding processes defined under the `#Processes` heading (Section 2.2.3).

2.3.4 The intruder

Finally, the operation of the intruder is specified by giving his identity, and the set of data values that he knows initially:

```
#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}
```

The above defines the intruder's identity to be *Mallory*, and defines that the intruder initially knows all the agents' identities, a single nonce Nm, the public key function PK, and his own secret key SKm. The inclusion of the function PK in the intruder's knowledge means that he can immediately calculate $PK(Alice)$, $PK(Bob)$ and $PK(Mallory)$.

Exercise 2.1 *Copy the input file* `NS3.spl`*, and then edit it so as to describe the following protocol:*

$$1.\ A \rightarrow B\ :\ A, B, \{na\}_{PK(B)}$$
$$2.\ B \rightarrow A\ :\ B, A, \{na, nb, B\}_{PK(A)}$$
$$3.\ A \rightarrow B\ :\ A, B, \{nb\}_{PK(B)}$$

♣

Exercise 2.2 *In some protocols, nonces are considered to be* predictable*: that is, an intruder may be able to predict which nonces other agents are about to use. How may we model predictable nonces within* Casper*?* ♣

Exercise 2.3 *In the above description of the Needham-Schroeder Public Key Protocol, B's public key $PK(B)$ was not included in the parameters of the process INITIATOR under the* `#Processes` *heading; instead, A was assumed to know the function PK, and calculated $PK(B)$ to send message 1. What would be the effect of including pkb as a parameter of INITIATOR under the* `#Processes` *heading, and then instantiating this with PKb (= $PK(Bob)$) under the* `#System` *heading?* ♣

Answers to the exercises appear in Appendix B.

3 Using Casper

3.1 Running Casper

The Casper compiler is written in the functional programming language Haskell. Executing the shell script `casper` will start up Haskell, and load in the Casper files.

The compiler is then run by typing `compile "`*filename*`"`, where the path for the input file is given by *filename*`.spl`. Note that the quotes are necessary. The suffix `spl` stands for *Security Protocol Language*.

This creates an output file in *filename*`.csp`. For example, running the compiler on the example of the 3 message Needham-Schroeder Public Key Protocol gives the following output:

```
Casper version 2.0
Parsing...
Type checking...
Consistency checking...
Compiling...
Writing output...
Output written to NS3.csp
Done
```

Alternatively, the function `vcompile` may be used in the same way; `vcompile` additionally writes a copy of the CSP code on the screen.

3.2 Running the output file with FDR

The output file can be loaded into FDR in the normal way.

Compiling creates refinement assertions corresponding to the specifications from the input file. For example, three refinement assertions are created for the file from Section 2:

```
SECRET_SPEC [T= SYSTEM_S
AuthenticateINITIATORToRESPONDERAgreement_na_nb [T= SYSTEM_1
AuthenticateRESPONDERToINITIATORAgreement_na_nb [T= SYSTEM_2
```

The first assertion corresponds to the secrecy specifications; the second assertion corresponds to the specification concerning authentication of the initiator to the responder; the third assertion corresponds to the specification concerning authentication of the responder to the initiator.

Each can be checked using FDR. If FDR finds an attack, then the FDR debugger can be used to find the corresponding CSP trace.

In the running example, testing the first assertion—concerning secrecy—finds that the refinement fails. The debugger shows that the right hand side will perform the following trace (with `tau`s not shown):

```
signal.Claim_Secret.Alice.Na.{Mallory}
signal.Claim_Secret.Bob.Nb.{Alice}
leak.Nb
```

The Casper package includes a function `interpret`, which can be used to interpret the output from FDR. This function is used by typing "`interpret`" at the Casper prompt (followed by return), pasting in the trace from the FDR debugger, and then typing a blank line (this might require you to hit return twice: once to end the last line from the debugger, and once for the blank line). A description of the attack will then appear on the screen. For the trace above, the following is printed:

```
Alice believes Na is a secret shared with Mallory
Bob believes Nb is a secret shared with Alice
The intruder knows Nb
```

This describes the attack at a high level, talking about the beliefs and knowledge of the agents: in this case, the error is that *Mallory* learns *Nb*, which *Bob* thought should have been kept secret.

The CSP definition of the system hides most of the details at the top level. However, moving down two levels in the debug tree, to the process `SYSTEM`, gives the following trace:

```
env.Alice.(Env0,Mallory,<>)
intercept.Alice.Mallory.(Msg1,Encrypt.(PK_.Mallory,<Na,Alice>),<>)
fake.Alice.Bob.(Msg1,Encrypt.(PK_.Bob,<Na,Alice>),<>)
intercept.Bob.Alice.(Msg2,Encrypt.(PK_.Alice,<Na,Nb>),<>)
fake.Mallory.Alice.(Msg2,Encrypt.(PK_.Alice,<Na,Nb>),<>)
intercept.Alice.Mallory.(Msg3,Encrypt.(PK_.Mallory,<Nb>),<Na>)
fake.Alice.Bob.(Msg3,Encrypt.(PK_.Bob,<Nb>),<Na>)
leak.Nb
```

We can use `interpret` to interpret this trace:

```
0.           ->   Alice   : Mallory
1.   Alice   -> I_Mallory : {Na, Alice}{PK_(Mallory)}
1.  I_Alice  ->    Bob    : {Na, Alice}{PK_(Bob)}
2.    Bob    ->  I_Alice  : {Na, Nb}{PK_(Alice)}
2. I_Mallory ->   Alice   : {Na, Nb}{PK_(Alice)}
3.   Alice   -> I_Mallory : {Nb}{PK_(Mallory)}
3.  I_Alice  ->    Bob    : {Nb}{PK_(Bob)}
  The intruder knows Nb
```

The notation `I_Alice` represents the intruder taking Alice's identity, either to fake a message (as in the second message 1) or to intercept a message intended for Alice (as in the first message 2). This attack can be re-written as follows, labelling the two runs α and β, writing for example $\beta.2$ for message 2 of run β:

$$\begin{array}{lrcl}
\alpha.1. & Alice & \rightarrow Mallory & : \{Na, Alice\}_{PK(Mallory)} \\
\beta.1. & I_{Alice} & \rightarrow Bob & : \{Na, Alice\}_{PK(Bob)} \\
\beta.2. & Bob & \rightarrow I_{Alice} & : \{Na, Nb\}_{PK(Alice)} \\
\alpha.2. & Mallory & \rightarrow Alice & : \{Na, Nb\}_{PK(Alice)} \\
\alpha.3. & Alice & \rightarrow Mallory & : \{Nb\}_{PK(Mallory)} \\
\beta.3. & I_{Alice} & \rightarrow Bob & : \{Nb\}_{PK(Bob)}
\end{array}$$

This is the attack from [Low95].

The second assertion checked concerns authentication of the initiator to the responder. This also fails. *Bob* believes that he has successfully completed a run of the protocol with *Alice* (using nonces *Na* and *Nb*), but *Alice* does not believe that she has been running the protocol with *Bob*. The attack is essentially the same as the previous.

Checking the final assertion, concerning authentication of the responder to the initiator, fails to find an attack.

The Casper package also includes a function `linterpret` that produces an interpretation of the attack as LaTeX source, suitable for use with the package `protdesc.sty`, which is included in the distribution.

Exercise 3.1 *Run* Casper *and FDR on your answer to Exercise 1 of Section 2. Use the* `interpret` *function to understand the attack.* ♣

4 Case study: The Wide-mouthed-frog Protocol

In this section we describe how to deal with some more protocol features, and illustrate some of the issues involved in modelling protocols by considering the example of the Wide-mouthed-frog Protocol of [BAN89]. The protocol can be described as follows:

$$1.\ A \rightarrow S\ :\ \{ts1, B, kab\}_{SKey(A)}$$
$$2.\ S \rightarrow B\ :\ \{ts2, A, kab\}_{SKey(B)}\ .$$

Here the server shares keys $SKey(A)$ and $SKey(B)$ with A and B, respectively; the protocol aims to establish a session key kab and to authenticate A to B. A invents a session key and sends it to S along with a timestamp $ts1$; S then forwards the key to B along with a new timestamp $ts2$.

As explained in Section 2, FDR cannot be used to consider the most general system running the protocol. Instead, we have to consider particular systems. We will consider four different systems running the protocol. FDR finds that there is no attack upon the first system, but finds three different attacks on the other systems. Part of the skill in using Casper is to know what systems to try checking. Larger systems require considerably more time to check (the rate of increase varies from protocol to protocol) so a pragmatic approach is to start with a small system, and work up, which is what we do here. The systems we check below have, admittedly, been tailored slightly to discovering certain attacks; however, the first three systems are among the first few I would try if coming at this protocol "blind".

4.1 Modelling the protocol

Most of the modelling of the protocol is straight-forward; below, we concentrate on the parts that require techniques not discussed above.

The free variables are declared as follows:

```
#Free variables
A, B : Agent
S : Server
SKey : Agent -> ServerKey
kab : SessionKey
```

```
ts, ts' : TimeStamp
InverseKeys = (SKey, SKey)
```

As explained in Section 2, most type names can be chosen by the user; one exception to this rule concerns timestamps, which must be represented by the type `TimeStamp`. Note that the `InverseKeys` line declares that keys returned by the *SKey* function are self-inverse, i.e. symmetric.

The agents taking part in the protocol can be defined as follows:

```
#Processes
INITIATOR(A,S,kab) knows SKey(A)
RESPONDER(B) knows SKey(B)
SERVER(S) knows SKey
```

Note that the server is assumed to know all the keys he shares with other agents, so we specify that he knows all of the *SKey* function. However, A and B only know the key that they personally share with the server, i.e. $SKey(A)$ and $SKey(B)$, respectively.

The exchange of messages can then be defined by:

```
#Protocol description
0.    -> A : B
1.  A -> S : {B, ts, kab}{SKey(A)}
[ts==now or ts+1==now]
2.  S -> B : {A, ts', kab}{SKey(B)}
[ts'==now or ts'+1==now]
```

The lines within square brackets represent tests that are performed by the agent who receives the message on the previous line; if the test fails, the agent aborts the run. Within tests, the distinguished variable `now` represents the current time. When S receives message 1 he checks that the timestamp is recent—either the current time or the previous time; if the test fails then S refuses to accept this message, and aborts the run. B makes a similar check when he receives message 2.

When an agent sends a timestamped message (as with messages 1 and 2 above), the value of the timestamp will be set to the current time.

We consider now the specification for the protocol. It would seem that if a responder B completes a run of the protocol apparently with A, then A should have been running the protocol within the previous two time units: the tests performed on the timestamps allow for a delay of one time unit per message (this assumes that there is negligible delay between S checking message 1 and sending message 2; we formalize this assumption below). Further, the two agents should agree on the value of *kab*. Our specification is a timed version of the `Agreement` specification of Section 2:

```
#Specification
TimedAgreement(A,B,2,[kab])
```

The above specification can be paraphrased as: "If responder B completes a protocol run, apparently with A, then the same agent A has been running the protocol with B within the last 2 time units, and both agents agreed as to which roles they took, and upon the value of *kab*; and further each such run of A corresponds to a unique run of B."

4.2 First system

It is normally a good idea to start off by checking a small system, because more often than not this will uncover any attacks. We therefore consider a system with a single initiator, Alice, and a single responder, Bob, each of whom can run the protocol once.

```
#System
INITIATOR(Alice, Sam, Kab)
RESPONDER(Bob)
SERVER(Sam)
```

The actual variables in the system are defined by:

```
#Actual variables
Alice, Bob, Mallory : Agent
Sam : Server
Kas, Kbs, Kms : ServerKey
Kab : SessionKey
```

The time domain to be considered is also defined under the `#Actual variables` heading; it should be defined to be a contiguous subsequence of the integers, as below. In this case, we choose a very small (one point!) time domain, so as to make checking as fast as possible. It is also necessary to make an implementation assumption about how long any particular run should last; this is defined by the variable `MaxRunTime`; if any run lasts for longer than this time, then the agent involved should timeout and abort the run.

```
TimeStamp = 0 .. 0
MaxRunTime = 0
```

The function *SKey*, which returns the key an agent shares with the server, is defined by:

```
#Functions
SKey(Alice) = Kas
SKey(Bob) = Kbs
SKey(Mallory) = Kms
```

We assume that the intruder initially knows all the agents' identities and his own key:

```
#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Sam, SKey(Mallory)}
```

When the above file is compiled using Casper, and then checked using FDR, FDR fails to find any attack upon this small system.

4.3 Second system

We now consider a slightly different system, where the agent Alice can run the protocol once as initiator and once as responder, possibly concurrently. We suppose that Bob is absent, so doesn't run the protocol.

```
#System
INITIATOR(Alice, Sam, Kab)
RESPONDER(Alice)
SERVER(Sam)
```

Note that this change to the system involves changing precisely one line of the input file.

When we check this system, FDR discovers that the protocol does not correctly authenticate the initiator Bob to responder Alice. By exploring the debug tree and using `interpret` we can discover that the attack takes the following form:

$$\begin{aligned} &\alpha.1.\ Alice \rightarrow Sam \ : \ \{Bob, 0, Kab\}_{SKey(Alice)} \\ &\beta.2.\ M_{Sam} \rightarrow Alice \ : \ \{Bob, 0, Kab\}_{SKey(Alice)} \ . \end{aligned}$$

The intruder simply replays Alice's first message back at her, which she interprets as being message 2 of a run initiated by Bob.

4.4 Third system

We will now consider a slightly different system, where the responder Bob can run the protocol twice, sequentially:

```
#System
INITIATOR(Alice, Sam, Kab)
RESPONDER(Bob) ; RESPONDER(Bob)
SERVER(Sam)
```

When we check this system, FDR tells us that Alice is not correctly authenticated. Using the debugger and `interpret` gives the following:

```
  Alice believes she is running the protocol, taking role INITIATOR,
with Bob, using data items Kab
  Bob believes he has completed a run of the protocol, taking role
RESPONDER, with Alice, using data items Kab
  Bob believes he has completed a run of the protocol, taking role
RESPONDER, with Alice, using data items Kab
```

Clearly the problem is that Bob thinks he has completed two runs of the protocol, while Alice only wanted to perform a single run.

Exploring the debug tree further, we can find that the attack takes the following form:

$$\begin{array}{rl} \alpha.1. & Alice \rightarrow Sam \ : \ \{Bob, 0, Kab\}_{SKey(Alice)} \\ \alpha.2. & Sam \rightarrow Bob \ : \ \{Alice, 0, Kab\}_{SKey(Bob)} \\ \beta.2 & M_{Sam} \rightarrow Bob \ : \ \{Alice, 0, Kab\}_{SKey(Bob)} \ . \end{array}$$

The intruder simply replays the message from Sam to Bob, so that Bob thinks that Alice is trying to establish a second session.

4.5 Fourth system

We now seek an attack that breaks the 2 time unit limit. To do this, we should consider a larger time domain:

```
TimeStamp = 0 .. 3
```

We will consider a system where initiator Alice and responder Bob each run the protocol once, but where the server can run the protocol three times:

```
#System
INITIATOR(Alice, Sam, Kab)
RESPONDER(Bob)
SERVER(Sam) ; SERVER(Sam) ; SERVER(Sam)
```

When this system is checked, FDR finds that initiator Alice is not authenticated according to the above timed specification. Using the debugger and `interpret` produces the following result:

```
  Alice believes she is running the protocol, taking role INITIATOR,
with Bob, using data items Kab
  Time passes
  Time passes
  Time passes
  Bob believes he has completed a run of the protocol, taking role
RESPONDER, with Alice, using data items Kab
```

Each "`time passes`" represents one unit of time passing. Bob completes his protocol run three time units after Alice was running the protocol.

Exploring the debug tree, we can find that the attack takes the following form:

$$
\begin{array}{lll}
\alpha.1. & Alice \rightarrow Sam & : \{Bob, 0, Kab\}_{SKey(Alice)} \\
\alpha.2. & Sam \rightarrow M_{Bob} & : \{Alice, 0, Kab\}_{SKey(Bob)} \\
 & & tock \\
\beta.1. & M_{Bob} \rightarrow Sam & : \{Alice, 0, Kab\}_{SKey(Bob)} \\
\beta.2. & Sam \rightarrow M_{Alice} & : \{Bob, 1, Kab\}_{SKey(Alice)} \\
 & & tock \\
\gamma.1. & M_{Alice} \rightarrow Sam & : \{Bob, 1, Kab\}_{SKey(Alice)} \\
\gamma.2. & Sam \rightarrow M_{Bob} & : \{Alice, 2, Kab\}_{SKey(Bob)} \\
 & & tock \\
\delta.1. & M_{Sam} \rightarrow Bob & : \{Alice, 2, Kab\}_{SKey(Bob)}.
\end{array}
$$

The intruder plays ping-pong with the server, continually replaying messages so that the timestamp is updated each time. It should be obvious how the intruder could continue such an exchange for longer, so as to break a timed specification with a weaker time constraint. This attack was described by Anderson and Needham in [AN95].

4.6 Timestamp Details

Timestamps are implemented internally within Casper using the concept of *agestamps* where each timestamp actually specifies how long ago it was created. This allows for a more accurate simulation of an infinite time domain than using absolute timestamps would allow. To implement this it was assumed that no time elapses between an agent receiving a message and sending a response. This means that time can only pass whilst the message is being held by the intruder.

In order to test protocols that use timestamps correctly the following rules of thumb should be obeyed. The maximum run time should be set to one and all checks of the form $[ts == now]$ should allow a timestamp

to either be *now* or *now − 1*. The time domain should then be chosen to be one greater than this, so typically the time bound should be specified as $MinTime = 0, MaxTime = 2$.

Since agestamps are used rather than timestamps Casper will report timestamps with negative values in counterexample traces. This indicates how old the message is so an agestamp of value -1 indicates that one time unit has passed since the timestamp was created.

5 Carrying on

In this section we explain how to handle a few features that often crop up in protocols.

5.1 The % notation

It will often be the case that the sender and receiver of a message treat that message differently. For example, in many protocols an agent receives an encrypted message that it does not decrypt, but instead simply forwards it to a third party. This is the case in the standard Yahalom protocol:

$$
\begin{array}{lll}
1.\ a \rightarrow b & : & na \\
2.\ b \rightarrow s & : & \{a, na, nb\}_{ServerKey(b)} \\
3.\ s \rightarrow a & : & \{b, kab, na, nb\}_{ServerKey(a)}, \{a, kab\}_{ServerKey(b)} \\
4.\ a \rightarrow b & : & \{a, kab\}_{ServerKey(b)}, \{nb\}_{kab}\ .
\end{array}
$$

a does not decrypt the second component of message 3, but simply forwards it to b in message 4.

Casper expects agents receiving messages to be able to decrypt them (this helps to trap many user errors); hence we need some way of indicating to Casper that certain messages really aren't intended to be decrypted: this is the role of the %-notation. We write `m % v`, where m is a message and v is a variable, to denote that the recipient of the message should not attempt to decrypt the message m, but instead store it in the variable v. Similarly, we write `v % m` to indicate that the sender should send the message stored in the variable v, but the recipient should expect a message of the form given by m. For example, we would model the standard Yahalom protocol using a script with a `#Protocol description` section as follows:

```
#Protocol description
0.    -> a : b
1.  a -> b : na
```

```
2.  b -> s : {a, na, nb}{ServerKey(b)}
3 . s -> a : {b, kab, na, nb}{ServerKey(a)}, \
             {a, kab}{ServerKey(b)} % v
4.  a -> b : v % {a, kab}{ServerKey(b)}, {nb}{kab}
```

a stores the second component of message 3 in the variable v and forwards it to b in message 4.

In an implementation, the agents would not be able to tell whether the message they receive is of the expected form. We therefore allow the agents to accept an arbitrary message of the expected type, or a special value `Garbage`, representing a random sequence of bits invented by the intruder.

The %-notation can be used not only for the case where a message is simply forwarded without decryption, but also more generally where the sender and receiver treat the message differently. For example, consider the following version of the 7 message Needham Schroeder Public Key Protocol:

$$
\begin{array}{l}
1.\ a \to s\ :\ b \\
2.\ s \to a\ :\ \{b, PK(b)\}_{SSK(s)} \\
3.\ a \to b\ :\ a, b, \{na, a\}_{PK(b)} \\
4.\ b \to s\ :\ a \\
5.\ s \to b\ :\ \{a, PK(a)\}_{SSK(a)} \\
6.\ b \to a\ :\ b, a, \{na, nb, b\}_{PK(a)} \\
7.\ a \to b\ :\ a, b, \{nb\}_{PK(b)}\ .
\end{array}
$$

The purpose of message 2 is for a to obtain b's public key. However, writing $PK(b)$ in the protocol description is rather misleading: a should be willing to accept any key, call it pkb, in this message, and then use that key pkb for the rest of the protocol. We hope that the form of message 2 ensures that the key that a receives really is $PK(b)$, but this is something we need to check; indeed, in Exercise 2 we will see an example where this is not the case.

The following Casper protocol description treats $PK(b)$ (and $PK(a)$) as required:

```
#Protocol description
0.    -> a : b
1.  a -> s : b
2.  s -> a : {b, PK(b) % pkb}{SSK(s)}
3.  a -> b : {na, a}{pkb % PK(b)}
4.  b -> s : a
5.  s -> b : {a, PK(a) % pka}{SSK(s)}
6.  b -> a : {na, nb, b}{pka % PK(a)}
7.  a -> b : {nb}{pkb % PK(b)}
```

The %-notation can also be used to model protocols that make use of either key certificates or tickets. For example, the Kehne-Langendorfer-Scoenwalder protocol has two phases:

- An initial exchange between A and B, which establishes a ticket of the form $\{A, kab\}_{Private(B)}$ where kab is a session key, and $Private(B)$ is a key known only to B[1];
- A re-authentication phase, where the ticket is re-used to re-establish authentication.

We can model the second phase as follows:

```
#Protocol description
0.    -> A : B, Shared(A,B) % kab, {A, Shared(A,B)}{Private(B)} % tickb
1.  A -> B : na, tickb % {A, kab}{Private(B)}
2.  B -> A : nb, {na}{kab}
3.  A -> B : {nb}{kab}
```

A receives three things in message 0:

- The identity of the agent B with whom A will run the protocol, as normal;
- The key kab to be used in the exchange, which we model as the result of a function application $Shared(A, B)$;
- A ticket of the form $\{A, Shared(A, B)\}_{Private(B)}$ which A stores in the variable $tickb$.

One can think of an environmental message such as this as representing an agent retrieving information from wherever it is stored.

Exercise 5.1 *Produce a* Casper *script to model the Yahalom-BAN protocol:*

$$
\begin{aligned}
&1.\ a \rightarrow b \ : \ na \\
&2.\ b \rightarrow s \ : \ nb, \{a, na\}_{ServerKey(b)} \\
&3.\ s \rightarrow a \ : \ nb, \{b, kab, na\}_{ServerKey(a)}, \{a, kab, nb\}_{ServerKey(b)} \\
&4.\ a \rightarrow b \ : \ \{a, kab, nb\}_{ServerKey(b)}, \{nb\}_{kab}\ .
\end{aligned}
$$

♣

[1] In the original version of the protocol, the ticket included a timestamp, which we omit here for simplicity.

Exercise 5.2 *Adapt the script for the seven message adapted Needham Schroeder Public Key Protocol to remove the identities from within the encrypted components of the key delivery messages, messages 3 and 6:*

$$
\begin{array}{l}
1.\ a \rightarrow s\ :\ b \\
2.\ s \rightarrow a\ :\ \{PK(b)\}_{SSK(s)} \\
3.\ a \rightarrow b\ :\ a, b, \{na, a\}_{PK(b)} \\
4.\ b \rightarrow s\ :\ a \\
5.\ s \rightarrow b\ :\ \{PK(a)\}_{SSK(a)} \\
6.\ b \rightarrow a\ :\ b, a, \{na, nb, b\}_{PK(a)} \\
7.\ a \rightarrow b\ :\ a, b, \{nb\}_{PK(b)}\ .
\end{array}
$$

Analyse this protocol using Casper *and FDR.* ♣

5.2 Vernam encryption

The notation `m (+) m'` represents the bit-wise exclusive-or, also known as Vernam encryption, of m and m'. The receiver of a message containing a Vernam encryption should be able to create at least one of m and m' so as to obtain the other.

Exercise 5.3 *Consider the TMN protocol from [TMN90]:*

$$
\begin{array}{l}
1.\ A \rightarrow S\ :\ B, \{ka\}_{pks} \\
2.\ S \rightarrow B\ :\ A \\
3.\ B \rightarrow S\ :\ A, \{kb\}_{pks} \\
4.\ S \rightarrow A\ :\ ka \oplus kb
\end{array}
$$

where pks is the public key of server s, ka and kb are session keys, and the intention is to establish a new session key kb shared between A and B. Use Casper *and FDR to analyse this protocol; you should discover an attack. Suggest how to adapt the protocol to prevent this attack, and then investigate whether the adapted protocol is secure.* ♣

5.3 Hash functions

Hash functions used in a Casper script should be declared as having the type `HashFunction` in the `#Free variables` section. Then `f(m)` represents the application of f to message m. In such cases, both the sender and the recipient should be able to create $f(m)$; the recipient will only accept a value for this message if the value received matches the value he calculates for himself. It is assumed that all hash functions are known to all agents.

5.4 Protocol specifications

Casper supports a number of different forms of specification for protocols, as follows:

- `Secret(A, s, [B1,...,Bn])` specifies that in any completed run, A can expect the value of the variable s to be a secret; $B_1, \ldots, B_n$ are the variables representing the roles with whom the secret is shared. More precisely, this specification fails if A can complete a run, where none of the roles $B_1, \ldots, B_n$ is legitimately taken by the intruder, but the intruder learns the value A gives to s.

- `StrongSecret(A, s, [B1,...,Bn])` is similar to `Secret(A, s, [B1, ...,Bn])`, except it also includes incomplete runs. Thus, this specification fails if A can take part in a run—complete or not—where none of the roles $B_1, \ldots, B_n$ is taken by the intruder, but the intruder learns the value A gives to s. It is arguable whether one should consider a failure of secrecy on an incomplete run to be an attack, because A will probably not do anything with the value of s if the run is incomplete; however, we should probably be aware of such behaviours. (Our main reason for including this specification form is that it is the form of secrecy considered in [Low98].)

- `Agreement(A, B, [v1,...,vn])` specifies that A is correctly authenticated to B, and the agents agree upon $v_1, \ldots, v_n$; more precisely, if B thinks he has successfully completed a run of the protocol with A, then A has previously been running the protocol, apparently with B, and both agents agreed as to which roles they took, and both agents agreed as to the values of the variables $v_1, \ldots, v_n$, and there is a one-one relationship between the runs of B and the runs of A.

- The specification `NonInjectiveAgreement(A, B, [v1,...,vn])` means that if B thinks he has successfully completed a run of the protocol with A, then A has previously been running the protocol, apparently with B, and both agents agreed as to which roles they took, and both agents agreed as to the values of the variables $v_1, \ldots, v_n$. Note that several runs of B may correspond to the same run of A.

- The specification `WeakAgreement(A, B)` means that if B thinks he has successfully completed a run of the protocol with A, then A has previously been running the protocol, apparently with B. Note that A and B may disagree as to which role each was taking.

- The specification `Aliveness(A, B)` means that if B thinks he has successfully completed a run of the protocol with A, then A has previously been running the protocol. Note that A may have thought he was running the protocol with someone other than B.

- The specification `TimedAgreement(A, B, t, [`$\mathtt{v}_1$`,...,`$\mathtt{v}_n$`])` is a timed version of `Agreement(A, B, [`$\mathtt{v}_1$`,...,`$\mathtt{v}_n$`])` where, in addition, A's run was within the previous t time units of B completing his run; by contrast, the `Agreement` specification macro places no constraints on the amount of time between the runs.

- Similarly, `TimedNonInjectiveAgreement(A, B, t, [`$\mathtt{v}_1$`,...,`$\mathtt{v}_n$`])`, `TimedWeakAgreement(A, B, t)`, and `TimedAliveness(A, B, t)` are timed versions of `NonInjectiveAgreement(A, B, [`$\mathtt{v}_1$`,...,`$\mathtt{v}_n$`])`, `WeakAgreement(A, B)`, and `Aliveness(A, B)`.

The different authentication specifications are discussed in more detail in [Low97].

5.5 Splitting large messages

If a protocol involves a particularly large message, then the resulting message space that FDR has to produce will be extremely large. In these circumstances, it is a good idea to split such large messages. For example, consider the Yahalom protocol, which could be written using the Casper notation as:

```
#Protocol description
0.    -> A : B
1.  A -> B : na
2.  B -> S : nb, {A, na}{SKey(B)}
3. S -> A : nb, {B, kab, na}{SKey(A)}, {A, kab, nb}{SKey(B)} % enc
4. A -> B : enc % {A, kab, nb}{SKey(B)}, {nb}{kab}
```

If we consider a small system with 3 agents, 3 shared keys, 3 nonces, and 2 session keys, then message 3 can take $8910 = 3^4.2.(3^3.2 + 1)$ different forms. It is better to write the protocol as:

```
#Protocol description
0.    -> A : B
1.  A -> B : na
2.  B -> S : nb, {A, na}{SKey(B)}
3a. S -> A : nb, {B, kab, na}{SKey(A)}
3b. S -> A : {A, kab, nb}{SKey(B)} % enc
```

```
4a. A -> B : enc % {A, kab, nb}{SKey(B)}
4b. A -> B : {nb}{kab}
```

Here we have split message 3 into two messages (now numbered 3a and 3b), and we have split message 4 into two messages (now numbered 4a and 4b). A few moments thought should convince you that this approach will identify the same attacks. (The above protocol is included in the Casper example library [Cas98].)

The one proviso to this approach is that if an agent learns a key in one part of the original message, that he then uses to decrypt another part of the message, then when the message is split, the parts must be ordered appropriately: in the above example, message 4a *must* precede message 4b, because B learns *kab* from message 4a, which he needs in order to decrypt message 4b.

It is also worth mentioning that splitting messages too much will increase the time taken by a check, because it will increase the number of ways in which simultaneous runs can be interleaved. It would probably be counter-productive to split message 2 above; it is arguable whether splitting message 3a further would be useful. (As a rule of thumb, I tend to split messages if they have more than about 100–200 possible forms, although more experience on this question would be useful.)

5.6 Agents withdrawing

Sometimes it is desirable to model the situation where agents can abort a run part way through, and then start a new run (where the agent is defined by a sequential composition within the `#System` section). This is done by putting the line

```
WithdrawOption = True
```

within the `#System` section. (The default is `WithdrawOption = False`, i.e. that agents do not abort runs part way through.)

5.7 Delaying decryption

Consider the following, simplified version of the SPLICE protocol of [YOM90]:

$$
\begin{aligned}
&1.\ A \to S\ :\ B.n1 \\
&2.\ S \to A\ :\ \{S.A.n1.PK(B)\}_{SK(S)} \\
&3.\ A \to B\ :\ \{A.\{n2\}_{PK(B)}\}_{SK(A)} \\
&4.\ B \to S\ :\ A.n3 \\
&5.\ S \to B\ :\ \{S.B.n3.PK(A)\}_{SK(S)} \\
&6.\ B \to A\ :\ \{B.n2\}_{PK(A)}
\end{aligned}
$$

In messages 1 and 2, A obtains B's public key from the key server S. In message 3, A sends a message to B, signed with his secret key. B then obtains A's public key in messages 4 and 5, and so can check message 3. Finally, B returns a message to A.

The important point to note here is that B cannot immediately decrypt message 3—he does not obtain A's public key until message 5. We model this within Casper by having B store the value of message 3 in a variable *enc* (using the % notation); B decrypting this message and performing the appropriate checks only after receiving message 5:

```
0.    -> A : B
1.  A -> S : B, n1
2.  S -> A : {S, A, n1, PK(B) % pkb}{SSK(S)}
3.  A -> B : {A, ts, {n2}{pkb % PK(B)}}{SK(A)} % v
4.  B -> S : A, n3
5.  S -> B : {S, B, n3, PK(A) % pka}{SSK(S)}
[decryptable(v, pka) and nth(decrypt(v,pka), 1) == A and \
  nth(decrypt(v,pka), 2) == now and \
  decryptable(nth(decrypt(v,pka), 3), SK(B))]
<n2 := nth (decrypt (nth(decrypt(v,pka), 3), SK(B)), 1)>
6.  B -> A : {B, n2}{pka % PK(A)}
```

In the test following message 5 and the assignment preceding message 6 we have made use of a few functions provided by Casper:

- The function `decryptable` takes a message and a key and tests whether the message may be decrypted with the key; that is, it tests whether the message is encrypted with the inverse of the key.
- The function `decrypt` takes a message and a key, and decrypts the message with the key; it should be applied only when the key is the correct decrypting key.

- The function `head` returns the first field from a message.
- The function `nth(_,`n`)` returns the nth field from a message.

Thus, the test following message 5 checks that: message 3 was encrypted with the inverse of the key received in message 5 (which B expects to be A's public key); that the first field inside the encryption was A's identity; and that the second field was encrypted with B's public key. The assignment preceding message 6 assigns *n2* to the contents of the inner encrypted component.

The way in which B processes message 3 when he receives the key in message 5 is of a fairly standard form, so I intend to introduce a primitive action `unpack_and_check`, representing this processing, in a future release of Casper.

5.8 Detecting type flaws

Up until now, we have been assuming that the different data types within a system were disjoint. That is, we have been considering systems where the atomic data items carried typing information, so that an agent receiving a data item could tell whether it was, for example, a nonce, an agent's identity, or a key. However, some protocol implementations do not achieve this, and as a result the protocols are open to attack.

Within Casper, it is possible to define that a particular data item can be interpreted as being of more than one type by including the name of that data item within more than one type definition line in the `#Actual variables` section. For example, we can define that Bob's identity could be interpreted as being either an agent's identity or a nonce via the lines:

```
#Actual variables
Alice, Bob, Mallory : Agent
Nb, Nb', Bob : Nonce
```

Exercise 5.4 *Consider the seven message version of the Needham-Schroeder Public Key Protocol:*

$$
\begin{array}{lll}
1.\ a \rightarrow s & : & b \\
2.\ s \rightarrow a & : & \{b, PK(b)\}_{SSK(s)} \\
3.\ a \rightarrow b & : & a, b, \{n_a, a\}_{PK(b)} \\
4.\ b \rightarrow s & : & a \\
5.\ s \rightarrow b & : & \{a, PK(a)\}_{SSK(a)} \\
6.\ b \rightarrow a & : & b, a, \{n_a, n_b\}_{PK(a)} \\
7.\ a \rightarrow b & : & a, b, \{n_b\}_{PK(b)}\ .
\end{array}
$$

Code up this protocol, using a system where two agents Alice and Bob can both run the protocol once as responder (but not as initiator), and the key server can run the protocol once. Also, allow Bob's nonce to be interpreted as an agent's identity. It is reasonable to assume that the key server knows the identities of all genuine agents, so would not accept Bob's nonce as being an agent's identity; how can we model this? FDR should find an attack upon the resulting system. ♣

Note that the attack found in the above exercise would not work if message 3 included the sender's identity as plaintext: the intruder would not be able to create message $\beta.3$, because he does not know the identity of the apparent sender Nb. When not considering type flaws it is normal to omit plaintext sender and receiver fields, because it is normal to assume that the intruder knows all the agents' identities; however, when considering type flaws, it can be important to include these identities.

Note that giving data items multiple types can greatly increase the state space to be checked, so this technique should be used with some caution.

More than a little guess work is required in deciding which data items should be given multiple types, in order to have a good chance of finding an attack. This question certainly requires more research, but a good rule of thumb would be to try and spot whether part of a message may be interpreted as coming from a different message when a data item is interpreted as being of a different type. For example, in the attack on the Needham-Schroeder Public Key Protocol discussed in question 4, a message 6 of the form $\{Nm, Nb\}_{PK(A)}$ was interpreted as being part of a message 3 when the nonce Nb was interpreted as being an agent's identity.

5.9 Algebraic equivalences

Many cryptographic functions have interesting algebraic properties. These are defined in a separate section, under the `#Equivalences` heading.

As an example, the lines:

```
#Equivalences
forall k1, k2, m . {{m}{k1}}{k2} = {{m}{k2}}{k1}
```

would express the property that the encryption property used is commutative, written mathematically as:

$$\forall k1, k2, m \bullet \{\{m\}_{k1}\}_{k2} = \{\{m\}_{k2}\}_{k1}.$$

Within such lines, bound variables for which no type is given—each of $k1$, $k2$ and m in this case—are interpreted as ranging over all messages. If such an

equivalence only applies for some of the variables being of a particular type, then such variables should be typed within the quantification, for example:

```
forall k1, k2 : SomeKeyType; m . {{m}{k1}}{k2} = {{m}{k2}}{k1}
```

Exercise 5.5 *Investigate the following protocol:*

1. $a \rightarrow b \ : \ \{\{a, kab\}_{PK(b)}\}_{SK(a)}$
2. $b \rightarrow a \ : \ \{\{b, kab\}_{PK(a)}\}_{SK(b)}$

where PK and SK return an agent's public and secret key, respectively, and where the intention is to establish a shared session key kab, in a setting where the encryption used is commutative. ♣

5.10 Intruder deductions

It is sometimes useful to be able to define additional ways in which the intruder can deduce new messages. Each deduction defines a way in which the intruder can deduce some message from some collection of other messages that he knows. Each deduction is defined around a turnstile (`|-`), with the message to the right of the turnstile represents the new message that the intruder can deduce, and the messages to the left represent the messages needed for this deduction. For example:

```
forall k1, k2: SessionKey; pks : ServerPublicKey . \
  {k1}{pks}, {k2}{pks} |- k1 (+) k2
```

Exercise 5.6 *The above deduction is relevant to the TMN protocol (see Exercise 3 of this section); if the intruder knows two distinct session keys both encrypted with the same server's public key, then he can replay them at the server, and use the server as an oracle to learn the Vernam encryption of the session keys. Investigate the effect of modelling the TMN protocol, using a system without a server, but where the intruder is given the above extra deduction.* ♣

5.11 Key compromise

Some protocols are subject to *key compromise attacks*, where a key is compromised—either through cryptographic techniques, or through the key being stolen—and then used to lead to a failure of authentication in a subsequent session.

All keys of a particular type can be declared to be compromisable by including a line like the following in the `#Intruder Information` section:

```
Crackable = SessionKey
```

The key will be compromised and passed to the intruder when all agents whose runs overlap in time with any agent using that key have finished their runs.

Exercise 5.7 *Use* Casper *to model the following slightly simplified version of the Needham-Schroeder Shared Key Protocol:*

$$\begin{array}{ll} 1. & A \rightarrow S \ : \ A, B, na \\ 2. & S \rightarrow A \ : \ \{na, B, kab\}_{SKey(A)} \\ 3. & S \rightarrow B \ : \ \{kab, A\}_{SKey(B)} \\ 4. & B \rightarrow A \ : \ \{nb\}_{kab} \\ 5. & A \rightarrow B \ : \ \{nb, nb\}_{kab} \end{array}$$

Specify that session key, such as kab, are crackable. You should find an attack upon this protocol. ♣

In protocols using timestamps, it is convenient to specify that keys are compromised after a particular period of time. This can be done using a line such as:

```
Crackable = SessionKey (3)
```

which specifies that the key is compromised after 3 time units.

Exercise 5.8 *Adapt the TMN Protocol script from Section 4.5 to specify that session keys can be compromised after three time units. You should find an attack upon the protocol.* ♣

5.12 Password guessing attacks

Some protocols make use of poorly-chosen secrets, such as passwords, which might be guessed by an intruder, who is then able to verify that guess. For example, such a verification might be by the intruder using the guessed value to decrypt a message to find a value that he has already seen. See [Low02] for a description of other ways in which such guesses can be verified.

Values of certain types can be specified to be guessable by including a line like the following in the `#Intruder Information` section:

```
Guessable = Password
```

When dealing with guessable types, Casper produces an extra refinement assertion to find whether the intruder can guess a value and then verify his guess. In most protocols, such a correct guess will also lead to further security failures.

Exercise 5.9 *The Encrypted Key Exchange Protocol (EKE) seeks to achieve key establishment and mutual authentication using a poorly-chosen password:*

$$
\begin{array}{lll}
1.\ a \to b & : & a, \{pk\}_{passwd(a,b)} \\
2.\ b \to a & : & \{\{k\}_{pk}\}_{passwd(a,b)} \\
3.\ a \to b & : & \{na\}_k \\
4.\ b \to a & : & \{na, nb\}_k \\
5.\ a \to b & : & \{nb\}_k
\end{array}
$$

Here $passwd(a, b)$ is a potentially poorly-chosen password shared between a and b, pk is an asymmetric key created by a, and k is a symmetric key created by b.

Model and analyse this protocol.

Then adapt the protocol so that pk is replaced by a symmetric key; you should find an attack. ♣

6 Secure channels

Some security protocols are designed to be layered on top of a secure transport protocol, such as TLS, which provides additional security services, acting as some kind of secure channel over which the messages are passed.

There are two distinct way to specify the services provided by such secure channels, each within a `#Channels` section.

6.1 Old-style secure channels

The original style of designating the properties of channels allowed the following two properties to be specified:

`secret:` the intruder cannot learn the contents of a message sent over the channel;

`authenticated:` if agent b receives messages on the channel apparently from a, then a really did send the messages, intended for b, and b receives a prefix of the messages sent by a.

For example

```
#Channels

authenticated
secret
```

All channels in the system are given these properties. See [BL03] for more details of these properties.

Under this style is is also possible to use the following designator:

`direct:` messsages may be sent directly between hoenst agents, rather than having to pass via the intruder; this gives shorter, clearer counter-example traces in some cases, but leads to a larger state space.

6.2 New-style secure channels

The more recent style of designating secure channels allows each message to be designated as being over a secure channel that provides one or more of the following services:

Confidentiality (C): The contents of the message are kept confidential from the intruder;

No faking (NF): The intruder cannot fake messages on this channel;

No redirecting (NR): The intruder cannot redirect a message so that it is received by an agent other than that for whom it was originally intended;

No honest redirecting (NR^-); The intruder cannot redirect messages, except when the message was originally intended for him;

No reascribing (NRA): The intruder cannot reascribe a message, so that it appears to come from an agent other than the original sender;

No honest reascribing (NRA^-): The intruder cannot reascribe a message, other than so that it appears to come from himself.

See [Dil08, DL08] for further description of these properties.

For each message in the protocol, the user should list the properties of the channel that the message is carried on. For example, the following channels section specifies the use of $C \wedge NR^-$ on the channel for message 2, and $NF \wedge NRA^-$ on the channel for messages 1 and 3.

```
#Channels

1 NF NRA-
2 C NR-
3 NF NRA-
```

The channel properties must be listed for each message individually, and must be listed in the order `C`, `NF`, `NRA(-)`, `NR(-)`.

The script below illustrates the technique on a version of the SAML Single Sign On protocol (note that this is not a realistic model of the protocol). A user wants to authenticate herself to a service provider, with the help of an identity provider. The user and identity provider have pre-existing shared secrets, so are able to establish confidential authenticated channels (that satisfy $C \wedge NF \wedge NRA^{-} \wedge NR^{-}$) in each direction (e.g. bilateral TLS); the service provider and identity provider each has some means of authenticating himself (e.g. a public key certificate) so are likewise able to establish confidential authenticated channels in each direction; the service provider has some means of authenticating himself (e.g. a public key certificate) so is able to establish a one-way authenticated channel (that satisfies $C \wedge NR^{-}$; e.g. unilateral TLS) to the user, but the user cannot directly authenticate herself to the service provider.

```
-- SAML Single Sign On (SSO) protocol

#Free variables
u : User
sp : ServiceProvider
idp : IdProvider
m : Message
nidp : Nonce
a : Artifact

PK : IdProvider -> PublicKey
SK : IdProvider -> SecretKey

InverseKeys = (PK, SK)

#Protocol description
0.     -> u   : sp, idp

-- Request for a resource: I am 'u', I want to access 'sp'
1. u   -> idp : u, sp

-- <AuthnResponse>
2. idp -> u   : a, idp, sp, u
3. u   -> sp  : a, sp, idp, u
4. sp  -> idp : a, u, sp, idp
5. idp -> sp  : {u, sp, idp, nidp}{SK(idp)}
6. sp  -> u   : m
```

```
#Processes
USER(u) knows PK
SERVICEPROVIDER(sp, m) knows PK
IDPROVIDER(idp, a, nidp) knows PK, SK(idp)

#Channels

1 C NF NRA- NR-
2 C NF NRA- NR-
3 C NR-
4 C NF NRA- NR-
5 C NF NRA- NR-
6 C NF NRA- NR-

#Specification
Secret(sp, m, [u])
Secret(u, m, [sp])

Agreement(sp, idp, [u, a])
Agreement(idp, sp, [u, a, nidp])
Agreement(sp, u, [m, idp])
Agreement(u, sp, [idp])
Agreement(idp, u, [sp])
Agreement(u, idp, [sp])

#Actual variables
Alice, Mallory : User
Dustin, Mallory : ServiceProvider
Sam : IdProvider
M, M', Mm : Message
A, A', Am : Artifact
Nidp, Nidp', Nm : Nonce

#Functions
symbolic PK, SK

#System
USER(Alice); USER(Alice)
SERVICEPROVIDER(Dustin, M); SERVICEPROVIDER(Dustin, M')
IDPROVIDER(Sam, A, Nidp); IDPROVIDER(Sam, A', Nidp')

#Intruder Information
Intruder = Mallory
```

```
IntruderKnowledge = {Alice, Dustin, Mallory, Sam, Am, Nm, Mm, PK}
```

Some secure transport protocols link messages together into *sessions*, so that the intruder cannot combine messages sent in different sessions so that they are received in the same session. In some cases, the relationship between sessions is *injective*: each session of one agent corresponds to a *single* session of another. Further, some secure transport protocols provide *symmetric* sessions: if messages sent by A in session s_A are received by B in session s_B, then messages sent by B in session s_B are receives by A in session s_A. Finally, some secure transport protocols ensure that messages are received in the same order in which they are sent, for example by including an authenticated message number; we call this the *stream* property. Again, see [Dil08, DL08] for more details.

In order to use the session and stream properties the user should use the keywords `Session` and `Stream`, optionally followed by one of the keywords `injective` or `symmetric`, followed by a list of the message numbers that should be joined into a single session. When two agents communicate on session channels (even on non-symmetric session channels), it is not necessary to create different sessions for each agent; the list of all messages sent by both agents should be included in the session. For example, the following session description specifies that messages 1, 2 and 3 are sent in a single, injective, session channel.

```
Session injective 1,2,3
```

The script below illustrates the use of session channels within the OpenID protocol.

```
-- OpenID authentication protocol

#Free variables
u  : User
rp : RelyingParty
op : OpenID
m  : Message
nk : Nonce
k  : HashKey
h  : HashFunction
k1 : NullKey

PK : OpenID -> NullKey
SK : OpenID -> NullKey
```

```
InverseKeys = (PK, SK)

#Protocol description
0.    -> u  : rp
1. u  -> op : rp
2. op ->  u : rp,nk,h(k,u) % hash
3.  u -> rp : op,nk,(hash % h(k,u)) % hashrp
4. rp -> u  : u
5. u  -> rp : op
6. rp -> op : u,nk,hashrp % h(k,u)
7. op -> rp : u
8. rp -> u  : m

#Processes
USER(u, op)
RELYINGPARTY(rp, m)
OPENID(op, nk, k)

#Channels

Session symmetric 1,2
Session symmetric 3,8
Session symmetric 6,7
Session 4,5

1 C NF NRA NR
2 C NF NRA NR
3 C NR-
6 C NR-
7 NF NRA-
8 NF NRA-

#Specification
Agreement(u, rp, [op, nk])
Agreement(rp, u, [m, op, nk])
Agreement(u, op, [rp, nk])
Agreement(op, u, [rp, nk])
Agreement(op, rp, [u, nk])
Agreement(rp, op, [u, nk])

#Actual variables
Alice, Mallory : User
Richard, Mallory : RelyingParty
```

```
Olive, Mallory : OpenID
M1, M2, Mm : Message
NK1, NK2, Nm : Nonce
K1, K2, Km : HashKey

#Functions
symbolic PK, SK

#System
USER(Alice, Olive); USER(Alice, Olive)
RELYINGPARTY(Richard, M1); RELYINGPARTY(Richard, M2)
OPENID(Olive, NK1, K1); OPENID(Olive, NK2, K2)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Richard, Olive, Mallory, Nm, Mm, Km}
```

The old-style designation of `secret` is equivalent to `C NRA NR-`; the old-style designation of `authenticated` is equivalent to `NF NRA NR` and `Stream`.

7 Using data independence techniques

In [Ros98], Roscoe showed how techniques borrowed from data independence and related fields can be used to achieve the illusion that agents can call upon an infinite supply of different nonces, keys, etc., even though the actual types used for these values remain finite. It is thus possible to create models of protocols in which agents do not have to stop after a small number of sequential runs.

In [Kle08] Kleiner used data independence methods to extend the work of Roscoe to allow an unbounded number of parallel sessions with the same agent to occur. Informally the idea is that agents in the system are *internalised* into the intruder; these agents are referred to as *internal agents*. This effectively means that the intruder is given an oracle of the behaviour of the internal agents.

In this section, we describe how these techniques can be used from within Casper.

7.1 How to use data independence techniques

A complete input script using data independence techniques is given in Appendix A.2. We describe below the essential data independence features.

The first change comes within the `#Processes` section. Variables of data independent types are indicated using the `generates` keyword: values for such variables will be freshly generated. For example, within the script of Appendix A.2, this section is as follows:

```
#Processes
INITIATOR(A) knows SKey(A) generates na
RESPONDER(B,S) knows SKey(B) generates nb
SERVER(S) knows SKey generates kab
```

The values `na`, `nb` and `kab` are of data independent types, and so are freshly generated for each run. By contrast, if this protocol was being modelled without the use of data independence techniques then this section would be defined as follows:

```
INITIATOR(A, na) knows SKey(A)
RESPONDER(B, nb) knows SKey(B)
SERVER(S, kab) knows SKey
```

Use of data independence techniques also affects the `#Free variables` section. Each free variable can now be specified with a *subtype*; if this is done then internal agents will only use actual variables that have the same subtype for that free variable. The `#Actual variables` section is also altered with each data independent variable not only receiving a subtype but also a *status* that is one of the following:

`InternalKnown` The variable should be used by internalised agents when communicating with an agent they think is dishonest (i.e. `Known` as in known to the intruder).

`InternalUnknown` The variable should be used by internalised agents when communicating with an agent they think is honest.

`External` The variable should be used by external agents only.

For example, the nonces in the Needham-Schroeder Public Key protocol could be declared as follows:

```
#Free variables
...
na: Nonce [NonceNA]
nb : Nonce [NonceNB]
...
```

```
#Actual variables
...
Np : Nonce (InternalKnown)
Na1, Na2 : Nonce (InternalUnknown) [NonceNA]
Nb1, Nb2 : Nonce (InternalUnknown) [NonceNB]
Ne1, Ne2 : Nonce (External)
...
```

Note that if a variable (either actual or free) has no subtype then it is assumed that it is a member of every subtype. For details of how many variables of each type are required see Section 7.2.

The last required alteration that is required is to add the flag `UnboundParallel = True` to the `#Intruder information` section of the input script.

7.2 Assumptions and Threshold Theorems

In this section we detail the assumptions that the data-independent model makes and the minimum number of actual variables required to prove a protocol is correct. In particular, if no attacks are found against a system that satisfies these assumptions and has sufficient actual variables then no attack exists on a system that contains an unbounded number of agents running this protocol using infinite types.

The data-independent model makes one assumption in addition to the Dolev-Yao assumptions; namely that there exist no inequality tests in the protocol. For example, it is frequently assumed that an agent will not open a session with itself and thus tests such as `[a != b]` are included. However, in the data-independent model a test of this sort is prohibited.

The minimum number of actual variables that are required depends on the type of specification being tested. We first consider secrecy specifications such as `Secret(b, v, as)` for which the following are required:

- One external instance of b;
- Two agent identities, one honest and one dishonest;
- Two actual variables, S and P that are `InternalUnknown` and `InternalKnown` respectively; S should be of the type and subtype of v. P should be of every type and subtype that is not the type or subtype of v;
- Sufficient `External` values such that external agents do not generate the same values for data-independent variables;

We illustrate the minimum number of required variables by considering the simplified Yahalom protocol, as presented below:

$$
\begin{aligned}
&1.\ \ a \rightarrow b\ :\ na\\
&2.\ \ b \rightarrow s\ :\ \{a, na, nb\}_{ServerKey(b)}\\
&3a.\ s \rightarrow a\ :\ \{b, kab, na, nb\}_{ServerKey(a)}\\
&3b.\ s \rightarrow b\ :\ \{a, kab\}_{ServerKey(b)}\\
&4.\ \ a \rightarrow b\ :\ \{nb\}_{kab}.
\end{aligned}
$$

Below is the the system setup and minimum number of variables as dictated by the above thresholds for the specification `Secret(b, kab, [a, s])`.

```
#Free variables
a, b : Agent
s : Server
na: Nonce [NonceNA]
nb : Nonce [NonceNB]
kab : SessionKey
ServerKey : Agent -> ServerKeys
InverseKeys = (kab, kab), (ServerKey, ServerKey)

#Processes
INITIATOR(a,na) knows ServerKey(a) generates na
RESPONDER(b,s,nb) knows ServerKey(b) generates nb
SERVER(s,kab) knows ServerKey generates kab

#Actual variables
Alice, Mallory : Agent
Alice : Server
vSecret : SessionKey (InternalUnknown)
vPublic : SessionKey (InternalKnown)
vPublic : Nonce (InternalKnown)
vPublic : Nonce (InternalUnknown)
Ne1 : Nonce (External)
InverseKeys = (vSecret,vSecret)

#Inline functions
symbolic ServerKey

#System
RESPONDER(Alice, Alice, Ne1)
```

We now present the thresholds for the authentication specifications. Consider specifications of the form `NonInjectiveAgreement(a,b,vs)`, `Agreement(a,b,vs)`, `WeakAgreement(a,b)` and `Aliveness(a,b)`; the system should be constructed as follows:

- For non-injective, aliveness and weak agreement specifications one external b should be constructed and for injective specifications two external b's should be constructed (the arguments to the external b's should differ only in data independent values);
- Two agent identities, one honest and one dishonest;
- Sufficient `External` values such that external agents do not generate the same values for data-independent variables;
- For every variable v in vs a unique `InternalUnknown` value v_s;
- A variable p that is of every type and subtype and has status `InternalUnknown` and `InternalKnown`.

As an example we again consider the simplified Yahalom protocol for the specification `Agreement(a,b,[kab])`.

```
#Free variables
a, b : Agent
s : Server
na: Nonce [NonceNA]
nb : Nonce [NonceNB]
kab : SessionKey
ServerKey : Agent -> ServerKeys
InverseKeys = (kab, kab), (ServerKey, ServerKey)

#Processes
INITIATOR(a,na) knows ServerKey(a) generates na
RESPONDER(b,s,nb) knows ServerKey(b) generates nb
SERVER(s,kab) knows ServerKey generates kab

#Actual variables
Alice, Mallory : Agent
Alice : Server
vSecret : SessionKey (InternalUnknown)
* vPublic : SessionKey (InternalUnknown)
vPublic : SessionKey (InternalKnown)
```

```
vPublic : Nonce (InternalKnown)
vPublic : Nonce (InternalUnknown)
* Ne1, Ne2 : Nonce (External)
InverseKeys = (vSecret,vSecret)

#Inline functions
symbolic ServerKey

#System
RESPONDER(Alice, Sam, Ne1)
* RESPONDER(Alice, Sam, Ne2)
```

Lines that have been changed or added compared to the secrecy specification above have been marked with a `*`.

Whilst the above represents the minimum number of variables required it is recommended that more values are provided in order to avoid type-flaw and certain secrecy attacks. For example, in the above `vPublic` is both a session key and a nonce and therefore some false attacks may be introduced. Also, above there is an overlap between `InternalUnknown` and `InternalKnown` session keys. This can lead to false attacks on protocols such as the one below:

$$
\begin{array}{lll}
1. & a \rightarrow b & : \ na \\
2. & b \rightarrow s & : \ \{a, na, nb\}_{ServerKey(b)} \\
3a. & s \rightarrow a & : \ nb, \{b, kab, na\}_{ServerKey(a)} \\
3b. & s \rightarrow b & : \ \{a, kab\}_{ServerKey(b)} \\
4. & a \rightarrow b & : \ \{nb\}_{kab}.
\end{array}
$$

Since nb is not transmitted encrypted if kab were an `InternalKnown` value the intruder could generate message 4 thus leading to a non-injective authentication attack. However, this attack should be regarded as false as, providing kab is freshly generated by the server, kab would not be known to the intruder.

To avoid the problems above the following system would be recommended for analysing the simplified Yahalom protocol.

```
#Actual variables
Alice, Mallory : Agent
Alice : Server
Kabs1, Kabs2 : SessionKey (InternalUnknown)
Kabp : SessionKey (InternalKnown)
Ni1 : Nonce (InternalKnown)
Ni1 : Nonce (InternalUnknown)
```

```
Ne1, Ne2 : Nonce (External)
InverseKeys = (vSecret,vSecret)

#Inline functions
symbolic ServerKey

#System
RESPONDER(Alice, Sam, Ne1)
RESPONDER(Alice, Sam, Ne2)
```

Note that the types now share no variables in common and there is no overlap between `InternalUnknown` and `InternalKnown` values for *kab*.

7.3 System Generation

The task of specifying the actual variables and external agents can be onerous and therefore Casper has the ability to automatically generate the actual variables and the external agent definitions. To do this, the `#Actual variables` section should be left blank aside from the type of the intruder. The flag `GenerateSystem = True` should then be specified in the `#System` section. An example is show below for the simplified Yahalom protocol above:

```
#Actual variables
Mallory : Agent

#Inline functions
symbolic ServerKey

#System
GenerateSystem = True

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {ServerKey(Mallory)}

UnboundParallel = True
```

Note that if multiple specifications are in the same file then they all must require the same agent to be external. Hence, all specifications must be of the form `Agreement(_,b,_)` or `Secret(b,_,_)` for some `b`.

The system that is generated is designed in such a way to avoid any type flaw attacks as mentioned above. However, it does not, by default, eliminate

false attacks that occur as a result of an overlap between `InternalUnknown` and `InternalKnown` values. To eliminate false attacks due to a variable `v` that overlaps place a secrecy specification of the form `Secret(a,v,as)` for appropriate `a` and `as` in the same file. This will cause the resulting generated system to have no overlap between the `InternalUnknown` and `InternalKnown` values.

7.4 Repeat Sections

A repeat section within a protocol is a section that is repeated a potentially unbounded number of times. For example, consider the following modified version of the KSL protocol (u is a user, as is the authentication server and s is a server that u wishes to be authenticated to):

$$
\begin{array}{lrcl}
1. & u \rightarrow as & : & s, ma \\
2a. & as \rightarrow u & : & \{ma, ts, s, kab\}_{UKey(u)} \\
2b. & as \rightarrow u & : & \{ma, ts, kab, u\}_{SKey(s)} \\
4. & u \rightarrow s & : & \{ma, ts, kab, u\}_{SKey(s)} \\
5. & s \rightarrow u & : & nb, \{ma, u\}_{kab} \\
6. & u \rightarrow s & : & \{s, nb\}_{kab}.
\end{array}
$$

Messages 4–6 are intended to be performed repeatedly until the ticket expires. If you were to analyse the protocol using the standard data independence methods you would find no attacks against the protocol. However, there exists an injective authentication attack against the repeated authentication section as shown below:

$$
\begin{array}{lrcl}
\alpha.1. & Alice \rightarrow Sam & : & Bob, Ma \\
\alpha.2a. & Sam \rightarrow Alice & : & \{Ma, 0, Bob, Kab\}_{UKey(Alice)} \\
\alpha.2b. & Sam \rightarrow Alice & : & \{Ma, 0, Kab, Alice\}_{SKey(Bob)} \\
\alpha.4. & Alice \rightarrow Bob & : & \{Ma, 0, Kab, Alice\}_{SKey(Bob)} \\
\alpha.5. & Bob \rightarrow Alice & : & Nb1, \{Ma, Alice\}_{kab} \\
\alpha.6. & Alice \rightarrow Bob & : & \{Bob, Nb1\}_{kab} \\
\beta.4. & Alice \rightarrow I_{Bob} & : & \{Ma, 0, Kab, Alice\}_{SKey(Bob)} \\
\beta.5. & I_{Bob} \rightarrow Alice & : & Nb1, \{Ma, Alice\}_{kab} \\
\beta.6. & Alice \rightarrow I_{Bob} & : & \{Bob, Nb1\}_{kab}.
\end{array}
$$

Hence, `Agreement(s, u, [kab])` does not hold in this protocol.

Casper can detect this attack if `GenerateSystemForRepeatSection = 4 to 6` is specified instead of `GenerateSystem = True`. When this flag is specified the system construction is altered; rather than ensuring that all data-independent arguments to external processes are disjoint it instead does the following. If a data-independent argument to the external process is first

used in the non-repeat section then an `InternalUnknown` value is picked (and the same value is picked for all external instances); otherwise, an external value is picked as before.

By forcing the external processes to use `InternalUnknown` values for variables introduced outside of the repeat section we ensure that the internal agents are capable of performing the same repeat section with variables differing only when they are introduced in the repeat section. Therefore, as the internal agents simulate an unbounded number of runs this simulates an unbounded number of repeat sessions.

7.5 Minimising state space explosion

When applying these techniques, one runs into problems with the size of the state space of the protocol model. We describe now several techniques for managing this state space explosion within data independence scripts.

Authentication specifications that test for agreement on many variables can be very slow and demanding on memory to test. To speed up the checks a specification such as `Agreement(a,b,vs)` can be split up into two different specifications; `Agreement(a,b,vs1)` and `Agreement(a,b,vs2)` where `vs = vs1++vs2`. Also, the use of % notation should be avoided as much as possible as the use of it dramatically increases the amount of memory that is required.

Testing protocols that contain timed specifications such as `TimedAgreement(a,b,t,vs)` can be very slow, especially if `t` is greater than 2. It is recommended that this specification is instead split up into two different checks; `Agreement(a,b,vs)` and `TimedAgreement(a,b,t,[])`. The `Agreement` specification can then be split up further as mentioned above.

Close inspection of the threshold theorems should show that as the number of authentication specifications in a single file rises so to does the number of values that are required to show the protocol is secure. Since increasing the number of values in the system can cause the checking time to rise greatly it can help to split specifications into several input files that can be run independently.

Lastly, whilst the `GenerateSystem` flag is extremely useful and makes using the data-independent model substantially easier it does not produce the minimum number of variables. It may be possible to reduce the number of variables required by collapsing some of the types which must be done by hand. It is recommended that the user starts off with the minimum number of values as dictated by the above threshold theorems and then adds variables as necessary to remove the false attacks.

7.6 Current limitations

The current limitations upon the models using the data independence techniques are as follows:

- No algebraic equivalences can be defined;
- Secure channels cannot be used;
- Guessing attacks cannot be modelled;
- Crackables are not allowed.

Note also that the counterexamples given by `casperFDR` are not very informative because any interaction with an internal agent will appear as `I(Agent)`. Therefore, if a protocol is found to be insecure then it is recommended that the user attempts to build a small system using the standard model that exhibits the problem.

8 Specifying Properties using Temporal Logic

Whilst most secrecy and authentication properties of security protocols can be captured using Casper's `Secret` and `Agreement` specifications there are some properties that cannot. For example, the specification *if Bob receives a message 3 then previously Alice sends a message 2 or Sam sends a message 2* is not testable because of the disjunction. Furthermore, the authentication properties apply only to complete runs of the protocol which is not true of the temporal specifications; since any message may be specified partial runs of the protocol are also considered. Casper therefore allows specifications to be entered in a restricted temporal logic consisting of just one temporal operator, *previously* (written as $\diamond$ below) along with conjunction, disjunction and implication.

Here we provide a brief description of the semantics, for the full details of the input syntax see Section C.6. A temporal logic specification is a formula of the form $p \Rightarrow q$ where:

- p is any formula built from $\wedge$, $\vee$ and $\diamond$ such that p is not a disjunction (for specifications such as $p \vee q \Rightarrow r$ the specification should be split into two specifications, $p \Rightarrow r$ and $q \Rightarrow r$);
- q is any formula built from $\wedge$, $\vee$ and $\diamond$.

Also, in the model that we consider the sending and receiving of messages is considered atomic and therefore no more than one atomic event can occur at any given time unit. Therefore, specifications such as $\diamond(p \wedge q)$ are not allowed (instead users can write $\diamond p \wedge \diamond q$). Lastly, $\diamond$ binds weakest of all meaning that $\diamond p \vee q$ is equal to $\diamond(p \vee q)$.

The atomic events that we allow are the sending or receiving of a message along with a list of the variables that should be bound. Some example events include:

```
a sends message 2 containing na, nb
A sends message 2 to b containing Na for na, nb
A receives message 2 from B containing Na for Na, Nb for Nb
```

Note that the sender and receiver of a message and any variable in the `containing` clause may be either an actual variable or a free variable. Whilst most of the time free variables will be preferable it can result in checks taking too long; if this is the case it is recommended that the user instead substitutes some free variables for actual variables.

The list of bound variables in the events allows the user to force messages to contain the same data item; in particular any data items in the same scope are required to match. For example, in the specification:

```
if a receives message 2 from b containing na then
    previously b sends message 1 containing na, a
```

the `na` and `a` are required to be the same. However, in the specification

```
if a receives message 3 from b containing na then
    (previously b sends message 1 containing na, nb, a)
    and (previously a sends message 2 containing nb)
```

the `nb` is not required to be the same. The scope of the variables can be defined as follows:

- Any free variable on the left hand side of an implication is in scope on the right hand side;
- In a formula such as $p \wedge \diamond q$ any free variable in p is in scope within q.

An example input file is shown below for the Needham-Schroeder Public Key protocol containing some temporal logic specifications:

```
-- Needham Schroeder Public Key Protocol, 3 message version

#Free variables
```

```
a, b : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)

#Processes

INITIATOR(a,na) knows PK, SK(a)
RESPONDER(b,nb) knows PK, SK(b)

#Protocol description

0.    -> a : b
1.  a -> b : {na, a}{PK(b)}
2.  b -> a : {na, nb}{PK(a)}
3.  a -> b : {nb}{PK(b)}

#Specification
-- PASS
if B receives message 3 from A containing Nb for nb then
  (previously A receives message 2 containing Nb for nb) and
    (previously A sends message 3 containing Nb for nb)

-- PASS
if A receives message 2 from B containing Na for na,Nb for nb then
  previously (
    B sends message 2 to A containing Na for na,Nb for nb,A for a
    and previously A receives message 0 containing B for b
  )

#Actual variables

A, B, I : Agent
Na, Nb, Nm : Nonce

#Functions

symbolic PK, SK

#System
```

```
INITIATOR(A, Na)
RESPONDER(B, Nb)

#Intruder Information

Intruder = I
IntruderKnowledge = {A, B, I, Nm, PK, SK(I)}
```

Temporal logic specifications may be used either with the normal models or in conjunction with the data independent models. Note that when using it with the data independent models the normal threshold theorems do not apply and therefore the `GenerateSystem` flag may not be used. The external agent should be picked to be the agent who sends or receives the event on the left hand side of the if that is not contained within any `previously` statement.

9 Simplifying transformations

Casper has been extended to implement the simplifying transformations described in [HL99]. These transformations can be used to simplify a protocol to make it easier to analyze. They have the property that if there is an attack upon the original protocol, then there is also an attack on the simplified version. We will assume some familiarity with the material in [HL99] in this section.

The simplifying transformation extension to Casper can be used to define a sequence of simplifications, which are then applied automatically to the protocol. The input script contains the first four sections of a standard Casper script, together with an additional section, `#Simplifications`, containing a list of the simplifications to be performed. An example `#Simplifications` section is below, which illustrates the seven types of simplifying transformation currently supported:

```
#Simplifications

RemoveFields [Nonce, TimeStamp]
RemoveHashedFields f(Nonce, Agent)
RemoveEncryption {Nonce, Agent}{PublicKey}
RemoveHash f(Nonce, Nonce, TimeStamp)
SwapPairs (Nonce, Agent)
Coalese (Nonce, Nonce)
RemovePlainAndEnc
```

All the simplifications are defined in terms of the type of the messages to be simplified. Each simplification is applied uniformly to all the messages of the protocol.

We describe each simplification in turn:

`RemoveFields` This simplification takes a list of types, and remove all fields of those types from the protocol description. For example, the simplification `RemoveFields [Nonce, TimeStamp]` will remove every instance of a nonce or timestamp.

`RemoveHashedFields` This simplification removes some hashed messages from the protocol description. For example, `RemoveHashedFields f(Nonce, Agent)` will remove all messages of the form $f(N, A)$ for $N \in Nonce$, $A \in Agent$.

`RemoveEncryption` This simplification strips off encryptions, leaving just the body of a message. For example, `RemoveEncryption {Nonce, Agent}{PublicKey}` will simplify any message of the form $\{N, A\}_K$, for $N \in Nonce$, $A \in Agent$, $K \in PublicKey$, replacing it by N, A.

`RemoveHash` This simplification strips off applications of hash functions, leaving just the body. The example `RemoveHash f(Nonce, Nonce, TimeStamp)` will simplify any message of the form $f(N1, N2, T)$ for $N1, N2 \in Nonce$, $T \in Timestamp$, replacing it by $N1, N2, T$.

`Coalesce` This simplification takes a pair of types as an argument, and coalesces pairs of adjacent atoms of those types, replacing them by just the first element. For example, `Coalesce (Nonce, Nonce)` will coalesce adjacent pairs of nonces, replacing them by the first element of the pair. This simplification should only be used (if the results of [HL99] are to be applied) if the field removed is either in the intruder's initial knowledge, or equal to the value removed; it is up to the user to check this condition.

`SwapPairs` This simplification takes a pair of types as an argument, and swaps pairs of adjacent atoms of those types. For example, `SwapPairs (Nonce,Agent)` swaps adjacent pairs of nonces and atoms.

`RemovePlainAndEnc` This simplification will simplify signed messages of the form $m, \{m\}_k$, for k a signature key, replacing them by just $\{m\}_k$. It is up to the user to ensure that all messages of this form are encrypted with signature keys.

A full sample input file can be found in Appendix A.3.

When Casper has been loaded, the command `simplify "`*filename*`"` will apply the simplifications defined in the file *filename*`.spl`, and print the new protocol description on the screen.

10 Conclusion

The Casper compiler has revolutionized our approach to analyzing protocols. Previously, when we produced the CSP by hand, it would take about a day to code up a protocol; now it takes only a few minutes. In particular, making small changes to the protocol or the system to be checked typically requires changes to only a couple of lines of the input file; when editing the CSP code by hand, the changes necessary were spread throughout the file, and it was hard to know whether you had remembered them all.

Also, it was easy to make mistakes when producing the CSP by hand, and these mistakes were hard to spot. When using Casper, errors are less common, most get caught by the compiler, and those errors that do get through are easier to spot because the file is so much shorter.

I anticipate that the Casper compiler will continue to evolve in the future. In particular, the following features are anticipated:

- Key compromise will be modelled;
- Non-atomic keys will be supported;
- A richer set of agent actions will be supported.

I would be interested to receive requests for further features.

Acknowledgements

Much of the code of Casper was written by Philippa Broadfoot. The simplifying transformations code was written by Mei Lin Hui.

I would also like to thank Bill Roscoe, Peter Ryan, Irfan Zakiuddan, Steve Schneider, Joshua Guttman, Paul Gardiner, Michael Goldsmith, Bryan Scattergood, Simon Ambler, Mark Reilly, Paul Norris and Ben Donovan for useful comments and discussions about Casper.

This work was partially funded by the US Office of Naval Research, the UK Defence Evaluation and Research Agency, and the UK EPSRC.

References

[AN95] Ross Anderson and Roger Needham. Programming Satan's computer. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, 1989.

[BL03] Philippa Broadfoot and Gavin Lowe. On distributed security transactions that use secure transport protocols. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 141–151, 2003.

[BLR00] Philippa Broadfoot, Gavin Lowe, and Bill Roscoe. Automating data independence. In *Proceedings of ESORICS 2000*, pages 175–190, 2000.

[Cas98] Casper example library, 1998. Available via URL `http://www.comlab.ox.ac.uk/people/gavin.lowe/Security/Casper/library.tar.gz`.

[Dil08] Christopher Dilloway. *On the Specification and Analysis of Secure Transport Layers*. DPhil, Oxford University, 2008. `http://web.comlab.ox.ac.uk/activities/security/papers/dilloway_thesis.pdf`.

[DL08] Christopher Dilloway and Gavin Lowe. Specifying secure transport layers. In *21st IEEE Computer Security Foundations Symposium (CSF 21)*, 2008. `http://www.comlab.ox.ac.uk/files/116/channels.pdf`.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2 User Manual*, 1997. Available via URL `http://www.formal.demon.co.uk/FDR2.html`.

[HL99] Mei Lin Hui and Gavin Lowe. Safe simplifying transformations for security protocols. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Kle08] Eldar Kleiner. A web services security study using Casper and FDR DPhil, Oxford University, 2008

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996. Also in *Software—Concepts and Tools*, 17:93–102, 1996.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997.

[Low98] Gavin Lowe. Towards a completeness result for model checking of security protocols (extended abstract). In *11th Computer Security Foundations Workshop*, 1998.

[Low02] Gavin Lowe. Analysing protocols subject to guessing attacks. In *Proceedings of WITS '02*, 2002.

[LR97] Gavin Lowe and Bill Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997.

[Mea96] Catherine A. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *ESORICS '96, LNCS 1146*, pages 351–364, 1996.

[NS78] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[RB99] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Ros95] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, 1995.

[Ros98] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *Proceedings of 11th IEEE Computer Security Foundations Workshop*, pages 84–95, 1998.

[TMN90] Makoto Tatebayashi, Natsume Matsuzaki, and David B. Newman, Jr. Key distribution protocol for digital mobile communication systems. In *Advances in Cryptology: Proceedings of Crypto '89*, volume 435 of *Lecture Notes in Computer Science*, pages 324–333. Springer-Verlag, 1990.

[YOM90] S. Yamaguchi, K. Okayama, and H. Miyahara. Design and implementation of an authentication system in WIDE Internet environment. In *Proc. 10th IEEE Region Conf. on Computer and Communication Systems*, 1990.

A Example input scripts

A.1 A standard input script

```
-- Needham Schroeder Public Key Protocol, 3 message version

#Free variables
A, B : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)

#Processes
INITIATOR(A,na) knows PK, SK(A)
RESPONDER(B,nb) knows PK, SK(B)

#Protocol description
0.    -> A : B
1.  A -> B : {na, A}{PK(B)}
2.  B -> A : {na, nb}{PK(A)}
```

```
3.  A -> B : {nb}{PK(B)}

#Specification
Secret(A, na, [B])
Secret(B, nb, [A])
Agreement(A,B,[na,nb])
Agreement(B,A,[na,nb])

#Actual variables
Alice, Bob, Mallory : Agent
Na, Nb, Nm : Nonce

#Functions
symbolic PK, SK

#System
INITIATOR(Alice, Na)
RESPONDER(Bob, Nb)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}
```

A.2 An input script using data independence techniques

```
-- Simplified version of Yahalom
#Free variables
a, b : Agent
s : Server
na: Nonce [NonceNA]
nb : Nonce [NonceNB]
kab : SessionKey
ServerKey : Agent -> ServerKeys
InverseKeys = (kab, kab), (ServerKey, ServerKey)

#Processes
INITIATOR(a,na) knows ServerKey(a) generates na
RESPONDER(b,s,nb) knows ServerKey(b) generates nb
SERVER(s,kab) knows ServerKey generates kab

#Protocol description
0.    -> a : b
```

```
1.  a -> b : na
2.  b -> s : {a, na, nb}{ServerKey(b)}
3a. s -> a : {b, kab, na, nb}{ServerKey(a)}
3b. s -> b : {a, kab}{ServerKey(b)}
4.  a -> b : {nb}{kab}

#Specification
Secret(b, kab, [a,s])
Agreement(a, b, [na,nb])
Agreement(a, b, [kab])

#Actual variables
Alice, Bob, Mallory : Agent
Sam : Server
Kabp : SessionKey (InternalKnown)
Kabs1, Kabs2 : SessionKey (InternalUnknown)
Np : Nonce (InternalKnown)
Na1, Na2 : Nonce (InternalUnknown) [NonceNA]
Nb1, Nb2 : Nonce (InternalUnknown) [NonceNB]
Ne1, Ne2 : Nonce (External)
InverseKeys = (Kabp, Kabp), (Kabs1, Kabs1), (Kabs2, Kabs2)

#Inline functions
symbolic ServerKey

#System
RESPONDER(Bob, Sam, Ne1)
RESPONDER(Bob, Sam, Ne2)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Sam, Np, ServerKey(Mallory)}

UnboundParallel = True
```

A.3 Simplifying transformations input script

```
#Free variables
A, B : Agent
na, nb : Nonce
PK : Agent -> PublicKey
SK : Agent -> SecretKey
kab : SessionKey
```

```
ts : TimeStamp
InverseKeys = (PK, SK), (kab,kab)
f : HashFunction

#Processes
INITIATOR(A,na, kab) knows PK, SK(A)
RESPONDER(B,nb,kab) knows PK, SK(B)

#Protocol description
0.    -> A : B
1.  A -> B : {na, A,ts}{PK(B)}
2.  B -> A : {na, nb, A,ts}{PK(A)}
3.  A -> B : f(na, A, f(B))
4.  B -> A : {na, A}{PK(A)}

#Specification
Secret(A, na, [B])
Agreement(A,B,[na,nb])
Agreement(B,A,[na,nb])

#Simplification
RemoveFields [TimeStamp]
SwapPairs (Nonce, Agent)
RemoveHash f(Agent)
```

B Answers to the exercises

B.1 Answers to exercises from Section 2

Answer to Exercise 1 The only necessary changes to the input file are in the `#Protocol description` section, which should be edited to the following:

```
#Protocol description
0.    -> A : B
1.  A -> B : {na}{PK(B)}
2.  B -> A : {na, nb, B}{PK(A)}
3.  A -> B : {nb}{PK(B)}
```

Answer to Exercise 2 Predictable nonces may be modelled by including those nonces within the intruder's initial knowledge. With the Needham-Schroeder protocol, the nonces *Na* and *Nb* may be made predictable by editing the `#Intruder Information` section to:

```
#Intruder Information
Intruder = Mallory
IntruderKnowledge = \
  {Alice, Bob, Mallory, Na, Nb, Nm, PK, SK(Mallory)}
```

Answer to Exercise 3 Having a line of the form `INITIATOR(Alice, Na, PKb)` in the `#System` section would have the effect that Alice would use Bob's public key *PKb* in *every* run of the protocol, including those runs with an agent other than Bob. This clearly isn't what we want.

In general, none of the parameters of a process should depend upon the identity of the other agent taking part in the protocol run.

B.2 Answers to exercises from Section 3

Answer to Exercise 1 When FDR is used to test the assertion dealing with secrecy, the test should fail. The attack can be interpreted as follows:

$0. \quad \rightarrow Alice \quad : Bob$
$1. \quad Alice \rightarrow I_{Bob} \quad : \{Na\}_{PK(Bob)}$
$1.\ I_{Mallory} \rightarrow Bob \quad : \{Na\}_{PK(Bob)}$
Bob believes he is running the protocol, taking role RESPONDER, with Mallory, using data items *Na*, *Nb*
$2. \quad Bob \rightarrow Mallory \ : \{Na, Nb, Bob\}_{PK(Mallory)}$
$2. \quad I_{Bob} \rightarrow Alice \quad : \{Na, Na, Bob\}_{PK(Alice)}$
Alice believes *Na* is a secret shared with Bob
The intruder knows *Na*

Alice tries running the protocol with Bob, but Mallory intercepts the first message 1. Mallory then forwards the encrypted component from the first message to Bob, but under his own identity (second message *1*). Bob sees nothing wrong with this message, so he responds with a message 2, from which Mallory can learn the value of *Na*. Finally, Mallory completes the protocol run by faking a message 2 back to Alice.

When FDR is used to test the assertion dealing with authentication of the responder Bob, the test should fail, giving an attack very similar to the one above.

B.3 Answers to exercises from Section 5

Answer to Exercise 1 The protocol decription should be changed to the following:

```
#Protocol description
0.    -> a : b
1.  a -> b : na
2.  b -> s : nb, {a, na}{ServerKey(b)}
3.  s -> a : nb, {b, kab, na}{ServerKey(a)}, \
             {a, kab, nb}{ServerKey(b)} % v
4.  a -> b : v % {a, kab, nb}{ServerKey(b)}, {nb}{kab}
```

Analyzing this protocol gives the following attack:

$$
\begin{array}{lcl}
0. & \rightarrow Alice & : Bob \\
1.\ Alice & \rightarrow I_{Bob} & : Na1 \\
1.\ I_{Bob} & \rightarrow Alice & : Na1 \\
2.\ Alice & \rightarrow I_{Sam} & : Na2, \{Bob, Na1\}_{ServerKey(Alice)} \\
2.\ I_{Alice} & \rightarrow Sam & : Na1, \{Bob, Na1\}_{ServerKey(Alice)} \\
3.\ Sam & \rightarrow I_{Bob} & : Na1, \{Alice, Kab, Na1\}_{ServerKey(Bob)}, \\
 & & \quad \{Bob, Kab, Na1\}_{ServerKey(Alice)} \\
3.\ I_{Sam} & \rightarrow Alice & : Na1, \{Bob, Kab, Na1\}_{ServerKey(Alice)}, \\
 & & \quad \{Bob, Kab, Na1\}_{ServerKey(Alice)} \\
4.\ Alice & \rightarrow I_{Bob} & : \{Bob, Kab, Na1\}_{ServerKey(Alice)}, \{Na1\}_{Kab}
\end{array}
$$

Answer to Exercise 2 You should find that in each case the intruder is able to change the identity on the public key requesting message so that the key server delivers the wrong public key, namely the intruder's.

Answer to Exercise 3 Your input script should look something like the following:

```
-- TMN protocol

#Free variables
A, B : Agent
S : Server
pks : PublicKey
sks : SecretKey
ka, kb : SessionKey
InverseKeys = (pks, sks)

#Processes
```

```
INITIATOR(A,S,pks,ka)
RESPONDER(B,S,pks,kb)
SERVER(S,sks)

#Protocol description
0.    -> A : B
[A != B]
1.  A -> S : B, {ka}{pks}
[A != B]
2.  S -> B : A
[A != B]
3.  B -> S : A, {kb}{pks}
4.  S -> A : kb (+) ka

#Specification
Secret(A, ka, [B,S])
Secret(A, kb, [B,S])
Secret(B, kb, [A,S])


#Actual variables
Alice, Bob, Mallory : Agent
Sam : Server
PKs : PublicKey
SKs : SecretKey
Ka, Kb, Km : SessionKey
InverseKeys = (PKs, SKs)

#System
INITIATOR(Alice, Sam, PKs, Ka)
RESPONDER(Bob, Sam, PKs, Kb)
SERVER(Sam, SKs)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Sam, PKs, Km}
```

This should detect an attack; which can be interpreted as follows:

1. $I_{Alice} \rightarrow Sam \ : \ Bob, \{Km\}_{PKs}$
2. $Sam \rightarrow Bob \ : \ Alice$
Bob believes Kb is a secret shared with Sam, Alice
3. $Bob \rightarrow Sam \ : \ Alice, \{Kb\}_{PKs}$
4. $Sam \rightarrow I_{Alice} \ : \ Km \oplus Kb$
The intruder knows Kb

This attack exploits the fact that message 1 is not authenticated. Removing the specification `Secret(B, kb, [A,S])` from the Casper script will lead to the following attack being discovered:

0. $\rightarrow Alice \ : \ Bob$
1. $Alice \rightarrow Sam \ : \ Bob, \{Ka\}_{PKs}$
2. $Sam \rightarrow I_{Bob} \ : \ Alice$
3. $I_{Bob} \rightarrow Sam \ : \ Alice, \{Km\}_{PKs}$
4. $Sam \rightarrow Alice \ : \ Km \oplus Ka$
Alice believes Ka is a secret shared with Bob, Sam
The intruder knows Ka

which exploits the fact that message 3 is not authenticated.

These attacks can be prevented by replacing the public key encryptions in messages 1 and 3 by encryptions using keys shared between the sender and the server. However, there are still attacks upon the resulting protocol. We leave it up to the reader to investigate and then prevent these attacks.

Answer to Exercise 4 To ensure that the key server accepts only genuine agents' identities, we will define a function *realAgent* which tests whether its argument really does represent an agent's identity; the server should apply this test to the identities it receives in messages 1 and 4.

We also want to restrict ourselves to searching for attacks where the user requests a connection with a genuine agent—rather than a random number that B happens to then choose as his nonce!—so this test is applied to the value received in message 0; note that this should be thought of more as a restriction on the environment than a test that A should actually perform.

The protocol and system definition is then straightforward:

```
-- Needham Schroeder Public Key Protocol,
-- 7 message version, with type flaw.

#Free variables
```

```
A, B : Agent
na, nb : Nonce
pka, pkb : PublicKey
PK : Agent -> PublicKey
SK : Agent -> SecretKey
PKS : Server -> ServerPublicKey
SKS : Server -> ServerSecretKey
realAgent : Agent -> Bool
S: Server
InverseKeys = (PK,SK), (PKS, SKS)

#Processes
INITIATOR(A,na,S) knows SK(A), PKS
RESPONDER(B,nb,S) knows SK(B), PKS
SERVER(S) knows PK, SKS(S)

#Protocol description
0.    -> A : B
[realAgent(B)]
1.  A -> S : B
[realAgent(B)]
2.  S -> A : {B, PK(B) % pkb}{SKS(S)}
3.  A -> B : {na, A}{pkb % PK(B)}
4.  B -> S : A
[realAgent(A)]
5.  S -> B : {A, PK(A) % pka}{SKS(S)}
6.  B -> A : {na, nb}{pka % PK(A)}
7.  A -> B : {nb}{pkb % PK(B)}

#Specification
Secret(A, na, [B])
Secret(B, nb, [A])
Agreement(A, B, [na,nb])
Agreement(B, A, [na,nb])

#Actual variables
Alice, Bob, Mallory, Nb : Agent
Na, Nb, Nm : Nonce
Sam : Server

#Functions
symbolic PK, SK, PKS, SKS
realAgent(Alice) = true
```

```
realAgent(Bob) = true
realAgent(Mallory) = true
realAgent(_) = false

#System
RESPONDER(Alice, Na, Sam)
RESPONDER(Bob, Nb, Sam)
SERVER(Sam)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = \
  {Alice, Bob, Mallory, Nm, Sam, PK, PKS, SK(Mallory)}
```

Note that we have defined *Nb* so that it can be interpreted as either a nonce or an agent's identity.

When FDR is used to check whether initiator Alice is correctly authenticated, it produces an attack which can be rewritten as:

$$
\begin{array}{lrcl}
\alpha.3. & Mallory_{Alice} & \rightarrow Bob & : \{Nm, Alice\}_{PK(Bob)} \\
\alpha.1. & Mallory_{Alice} & \rightarrow Sam & : Bob \\
\alpha.2. & Sam & \rightarrow Mallory_{Alice} & : \{PKb, Bob\}_{SKS(Sam)} \\
\alpha.4. & Bob & \rightarrow Sam & : Alice \\
\alpha.5. & Sam & \rightarrow Bob & : \{PKa, Alice\}_{SKS(Sam)} \\
\alpha.6. & Bob & \rightarrow Mallory_{Alice} & : \{Nm, Nb\}_{PK(Alice)} \\
\beta.3. & Mallory_{Nb} & \rightarrow Alice & : \{Nm, Nb\}_{PK(Alice)} \\
\beta.4. & Alice & \rightarrow Mallory_{Sam} & : Nb \\
\alpha.7. & Mallory_{Alice} & \rightarrow Bob & : \{Nb\}_{PK(Bob)}
\end{array}
$$

In run α, Mallory imitates initiator Alice in a run of the protocol with Bob. (Messages $\alpha.1$ and $\alpha.2$ serve only to get the server into the right state to accept a message 4; there is no reason why they have to be out of order as above.) When Bob sends a nonce challenge in message $\alpha.6$, Mallory sends the message to Alice as a message 3, which Alice interprets as coming from an agent *Nb*; Alice therefore requests *Nb*'s key in message $\beta.4$, but this allows Mallory to learn the value of *Nb* and so answer Bob's nonce challenge.

This attack was described by Meadows in [Mea96].

Answer to Exercise 5 The input script should look something like:

```
-- A protocol to illustrate the use of algebra.

#Free variables
```

```
a,b : Agent
kab : SessionKey
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK)

#Processes
INITIATOR(a,kab) knows PK, SK(a)
RESPONDER(b) knows PK, SK(b)

#Protocol description
0.    -> a : b
[a != b]
1.  a -> b : {{a, kab}{PK(b)}}{SK(a)}
[a != b]
2.  b -> a : {{b, kab}{PK(a)}}{SK(b)}

#Specification
Secret(b, kab, [a])

#Actual variables
A, B, M : Agent
K1, K2 : SessionKey

#Functions
symbolic PK, SK

#System
INITIATOR(A, K1)
RESPONDER(B)

#Intruder Information
Intruder = M
IntruderKnowledge = {A, B, M, PK, K2, SK(M)}

#Equivalences
-- This is the interesting bit.  The following specifies that the
-- encryption is commutative
forall k1, k2, m . {{m}{k1}}{k2} = {{m}{k2}}{k1}
```

The following attack can be discovered:

$0. \quad \rightarrow A \ : \ M$
$1. \ A \rightarrow M \ : \ \{\{A, K1\}_{SK(A)}\}_{PK(M)}$
$1. \ I_A \rightarrow B \ : \ \{\{A, K1\}_{PK(B)}\}_{SK(A)}$
B believes *K1* is a secret shared with A
The intruder knows *K1*

The first message 1 is somewhat surprising, as it doesn't seem to match the protocol description. However, it is equivalent, under the algebraic equivalence, to

$1. \ A \rightarrow M \ : \ \{\{A, K1\}_{PK(M)}\}_{SK(A))}$

which does match the protocol description. From the message in its first form, the intruder is able to derive $\{A, K1\}_{SK(A)}$, and hence $\{\{A, K1\}_{SK(A)}\}_{PK(B)}$, which is equivalent to the fakes message 1 in the attack.

Answer to Exercise 6 Taking the input script from Exercise 3, removing the server, and adding the given deduction will allow the following attack to be found:

$2. \ I_{Sam} \rightarrow Bob \ : \ Alice$
Bob believes *Kb* is a secret shared with Sam, Alice
$3. \ Bob \rightarrow I_{Sam} \ : \ Alice, \{Kb\}_{PKs}$
The intruder knows *Kb*

From $\{Kb\}_{PKs}$ and $\{Km\}_{PKs}$, the intruder is able to deduce $Km \oplus Kb$ (using the given deduction, or—in a system with a server—using the server as an oracle), and hence *Kb*.

Encoding oracle properties of an honest agent in this way can sometimes allow that agent to be left out of the system checked, but still allow attacks to be found, often searching a smaller state space.

Answer to Exercise 7 The script should be similar to the following.

```
-- Needham Schroeder Shared Key Protocol, with key compromise

#Free variables
A, B   : Agent
S      : Server
na, nb : Nonce
SKey   : Agent -> ServerKey
```

```
kab   : SessionKey
InverseKeys = (SKey,SKey), (kab,kab)

#Processes
INITIATOR(A,S,na) knows SKey(A)
RESPONDER(B,nb) knows SKey(B)
SERVER(S,kab) knows SKey

#Protocol description
0.   -> A : B
[A != B]
1. A -> S : A, B, na
2. S -> A : {na, B, kab}{SKey(A)}
3. S -> B : {kab,A}{SKey(B)}
[A != B]
4. B -> A : {nb}{kab}
5. A -> B : {nb,nb}{kab}

#Specification
Agreement(B, A, [])
Agreement(A, B, [])
Secret(A, kab, [B])
Secret(B, kab, [A])

#Actual variables
Alice, Bob, Ivor  : Agent
Stan              : Server
Na, Nb, Ni        : Nonce
Kab               : SessionKey
InverseKeys = (Kab,Kab)

#Inline functions
symbolic SKey

#System
INITIATOR(Alice, Stan, Na)
RESPONDER(Bob, Nb)
SERVER(Stan,Kab)

#Intruder Information
Intruder = Ivor
IntruderKnowledge = {Alice, Bob, Ivor, Stan, Ni, SKey(Ivor)}
Crackable = SessionKey
```

Checking the resulting secrecy specification reveals the following attack:

1. $I_{Alice} \rightarrow Stan \ : \ Alice, Bob, Ni$
2. $Stan \rightarrow I_{Alice} \ : \ \{Ni, Bob, Kab\}_{SKey(Alice)}$
3. $Stan \rightarrow I_{Bob} \ : \ \{Kab, Alice\}_{SKey(Bob)}$
Stan withdraws from this run as SERVER
Kab has been compromised
3. $I_{Stan} \rightarrow Bob \ : \ \{Kab, Alice\}_{SKey(Bob)}$
4. $Bob \rightarrow I_{Alice} \ : \ \{Nb\}_{Kab}$
5. $I_{Alice} \rightarrow Bob \ : \ \{Nb, Nb\}_{Kab}$
The intruder knows Kab

Answer to Exercise 8 The script should look something like the following:

```
-- Wide Mouthed Frog Protocol, with key compromise

#Free variables
A, B : Agent
S : Server
kab : SessionKey
ts, ts' : TimeStamp
SKey : Agent -> ServerKey
InverseKeys = (SKey,SKey),(kab,kab)

#Processes
INITIATOR(A,S,kab) knows SKey(A)
RESPONDER(B) knows SKey(B)
SERVER(S) knows SKey

#Protocol description
0.    -> A : B
1.  A -> S : {B, ts, kab}{SKey(A)}
[(ts==now or ts+1==now) and A != B]
2.  S -> B : {A, ts', kab}{SKey(B)}
[(ts'==now or ts'+1==now) and A != B]

#Specification
TimedAgreement(A,B,2,[kab])
Secret(A,kab,[B])
Secret(B,kab,[A])

#Actual variables
```

```
Alice, Bob, Mallory : Agent
Sam : Server
Kab : SessionKey
TimeStamp = 0 .. 3
MaxRunTime = 0
InverseKeys = (Kab,Kab)

#Functions
symbolic SKey

#System
INITIATOR(Alice, Sam, Kab)
RESPONDER(Bob)
SERVER(Sam) ; SERVER(Sam) ; SERVER(Sam)
WithdrawOption  = True

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Alice, Bob, Mallory, Sam, SKey(Mallory)}
Crackable = SessionKey (3)
```

This produces the following attack on secrecy:

$0. \quad \rightarrow Alice \ : \ Bob$
$1.\ Alice \rightarrow I_{Sam} \ : \ \{Bob, 0, Kab\}_{SKey(Alice)}$
Time passes
$1.\ I_{Alice} \rightarrow Sam \ : \ \{Bob, 0, Kab\}_{SKey(Alice)}$
$2.\ Sam \rightarrow I_{Bob} \ : \ \{Alice, 1, Kab\}_{SKey(Bob)}$
Time passes
$1.\ I_{Bob} \rightarrow Sam \ : \ \{Alice, 1, Kab\}_{SKey(Bob)}$
$2.\ Sam \rightarrow I_{Alice} \ : \ \{Bob, 2, Kab\}_{SKey(Alice)}$
Time passes
Kab has been compromised
$1.\ I_{Alice} \rightarrow Sam \ : \ \{Bob, 2, Kab\}_{SKey(Alice)}$
$2.\ Sam \rightarrow I_{Bob} \ : \ \{Alice, 3, Kab\}_{SKey(Bob)}$
$2.\ I_{Sam} \rightarrow Bob \ : \ \{Alice, 3, Kab\}_{SKey(Bob)}$
The intruder knows Kab

The intruder plays ping pong with Sam, keeping the key delivery messages current, until he is able to crack the key.

Answer to Exercise 9 The original protocol can be modelled as follows:

```
-- Encrypted Key Exchange

#Free variables
a, b : Agent
na, nb : Nonce
k : SessionKey
pk : PubKey
sk : SecKey
passwd : Agent x Agent -> Password
InverseKeys = (passwd, passwd), (pk, sk), (k,k)

#Processes
INITIATOR(a, pk, sk, na) knows passwd(a,b)
RESPONDER(b, nb, k) knows passwd(a,b)

#Protocol description
0.    -> a : b
1.  a -> b : a, {pk}{passwd(a,b)}
2.  b -> a : { {k}{pk} }{passwd(a,b)}
3.  a -> b : {na}{k}
4.  b -> a : {na, nb}{k}
5.  a -> b : {nb}{k}

#Specification
Secret(a, passwd(a,b), [b])
Secret(a, k, [b])
Agreement(b,a,[na])

#Actual variables
Alice, Bob, Mallory : Agent
Na1, Nb1, Nm : Nonce
K1, Km : SessionKey
PK1, PKm : PubKey
SK1, SKm : SecKey
InverseKeys = (PK1, SK1), (PKm, SKm), (K1, K1), (Km, Km)

#Functions
symbolic passwd

#System
INITIATOR(Alice, PK1, SK1, Na1)
RESPONDER(Bob, Nb1, K1)
```

```
#Intruder Information
Intruder = Mallory
IntruderKnowledge = \
  {Alice, Bob, Mallory, Nm, PKm, SKm, \
   passwd(Mallory, Alice), passwd(Mallory, Bob), \
   passwd(Alice, Mallory), passwd(Bob, Mallory)}
Guessable = Password
```

This produces no attacks.

If the protocol is adapted so that *pk* and *sk* are replaced by a single symmetric key, then FDR discovers an attack where the intruder observes a run between A and B up to message 4, guesses the value $passwd(A, B)$, and then verifies the guess using verifier `Na`. Exploring the debug tree, one can find out that the intruder : decrypts the second message with $passwd(A, B)$ to obtain $\{K\}_{PK}$; decrypts the first message with $passwd(A, B)$ to obtain PK; decrypts $\{K\}_{PK}$ with PK to obtain K (this step is not possible if PK is an asymmetric key); decrypts the third message to obtain Na; and decrypts the fourth message to obtain Na again; the fact that he obtains the same value Na from both these decryptions verifies the guess. It is also possible to verify the guess by decrypting the fourth and fifth messages to obtain Nb in two different ways.

C The Casper syntax

In this section we present the formal syntax for Casper scripts. We use EBNF, writing literals in teletype font within quotes; terms within curly brackets may be repeated zero or more times; terms within square brackets are optional; linebreaks are significant within Casper scripts, and are represented by "↵". Logical lines may be split across two or more lines by ending lines with "\" or by indenting the following line. Lines beginning with "`--`" are comments.

C.1 Basic definitions

$$
\begin{aligned}
\textit{identifier} &::= \textit{letter}\ \{\textit{letter} \mid \textit{digit}\} \\
\textit{atom} &::= \textit{identifier} \mid \textit{fn-app} \\
\textit{fn-app} &::= \textit{identifier}\ \text{`(`}\ \textit{identifier}\ \{\text{`,`}\ \textit{identifier}\}\ \text{`)`}
\end{aligned}
$$

msg ::= *identifier* |
identifier '(' *msg* ')' |
msg ',' *msg* |
'{' *msg* '}{' *atom* '}' |
msg '(+)' *msg* |
msg '%' *identifier* |
identifier '%' *msg* |
'(' *msg* ')'

type-name ::= *identifier*

process-name ::= *identifier*

The bit-wise exclusive-or operator `(+)` binds tighter than the `%` operator, which binds tighter than the sequencing operator ",".

There are two distinguised type names in Casper: `TimeStamp` and `HashFunction`.

C.2 Syntax summary

script ::= *free-vars-section*
processes-section
prot-desc-section
spec-section
[*equivalences-section*]
act-var-section
[*functions-section*]
system-section
intruder-section
channels-section

C.3 Free variables section

free-vars-section ::= '`#Free variables`' ↵
{*var-dec* | *inv-keys-dec*}

var-dec ::= *identifier* {',' *identifier*} ':' *type-expr* [*subtype-expr*] ↵

subtype-expr ::= '[' *identifier* {',' *identifier*} ']'

type-expr ::= *type-name* |
type-name {'x' *type-name*} '->' *type-name*

inv-keys-dec ::= ‘`InverseKeys =`’ *inv-key-pair*
{‘`,`’ *inv-key-pair*} ↵
inv-key-pair ::= ‘`(`’ *identifier* ‘`,`’ *identifier* ‘`)`’

Each *inv-key-pair* should be either a pair of variable names, or a pair of function names.

C.4 Processes section

processes-section ::= ‘`#Processes`’ ↵ {*process-def*}
process-def ::= *process-name*
‘`(`’ *identifier* {‘`,`’ *identifier*} ‘`)`’
[*knows-statement*]
[*generates-statement*] ↵
knows-statement ::= ‘`knows`’ *atom* {‘`,`’ *atom*}
generates-statement ::= ‘`generates`’ *identifier* {‘`,`’ *identifier*}

The first parameter of each process should represent agent identities used in the protocol description.

C.5 Protocol description section

prot-desc-section ::= ‘`#Protocol description`’ ↵
{*prot-msg* | *env-msg-send* | *env-msg-rec*}
prot-msg ::= [*assignment-line* ↵]
line-no ‘`.`’ *identifier* ‘`->`’ *identifier* ‘`:`’
msg ↵
[*test-line* ↵]
assignment-line ::= ‘`<`’ *assignment* {‘`;`’ *assignment*} ‘`>`’
asssignment ::= *identifier* ‘`:=`’ *fdr-expression*
test-line ::= ‘`[`’ *test* ‘`]`’
test ::= *fdr-expression*
env-msg-rec ::= *line-no* ‘`.`’ ‘`->`’ *identifier* ‘`:`’ *msg* ↵
[*test-line* ↵]
env-msg-send ::= [*assignment-line* ↵]
line-no ‘`.`’ *identifier* ‘`->`’ ‘`:`’ *msg* ↵
line-no ::= (*letter* | *digit*) {*letter* | *digit*}

The sender and receiver fields for each message (i.e. the fields preceding and following the '`->`') should both be declared as identities of agents in the `#Processes` section.

The type *fdr-expression* represents all expressions accepted by FDR[2]; the syntax is basically that of a simple functional language; see [For97] for details.

C.6 Specification section

spec-section ::= '`#Specification`' ↵ {*spec* | *temporal-spec*}

spec ::= '`Secret`'
'`(`' *identifier* '`,`' *atom* '`,`' *agents* '`)`' ↵ |
'`StrongSecret`'
'`(`' *identifier* '`,`' *atom* '`,`' *agents* '`)`' ↵ |
'`Agreement`' '`(`' *identifier* '`,`' *identifier* '`,`'
fields '`)`' ↵ |
'`NonInjectiveAgreement`' '`(`' *identifier*
'`,`' *identifier* '`,`' *fields* '`)`' ↵ |
'`WeakAgreement`'
'`(`' *identifier* '`,`' *identifier* '`)`' ↵ |
'`Aliveness`'
'`(`' *identifier* '`,`' *identifier* '`)`' ↵ |
'`TimedAgreement`'
'`(`' *identifier* '`,`' *identifier* '`,`'
time '`,`' *fields* '`)`' ↵ |
'`TimedNonInjectiveAgreement`'
'`(`' *identifier* '`,`' *identifier* '`,`'
time '`,`' *fields* '`)`' ↵ |
'`TimedWeakAgreement`' '`(`' *identifier* '`,`'
identifier '`,`' *time* '`)`' ↵ |
'`TimedAliveness`'
'`(`' *identifier* '`,`' *identifier* '`,`' *time* '`)`' ↵

agents ::= '`[`' *identifier* {'`,`' *identifier*} '`]`'

fields ::= '`[`' [*identifier* {'`,`' *identifier*}] '`]`'

time ::= ℕ

temporal-spec ::= '`if`' *temporal-formula* '`then`' *temporal-formula*

[2]In fact, Casper doesn't parse such expressions itself, but simply copies them directly into the output file, for FDR to subsequently parse.

temporal-formula ::= *temporal-formula* ‘`and`’ *temporal-formula*
| *temporal-formula* ‘`or`’ *temporal-formula*
| ‘`(`’ *temporal-formula* ‘`)`’
| ‘`previously`’ *temporal-formula*
| *temporal-event*

temporal-event ::= *identifier* (‘`sends`’ | ‘`receives`’) ‘`message`’
line-no [(‘`from`’ | ‘`to`’) *identifier*]
[‘`containing`’ *substitution* {‘`,`’ *substitution*}]

substitution ::= *identifier* | (*identifier* ‘`for`’ *identifier*)

C.7 Algebraic equivalences section

equivalences-section ::= ‘`#Equivalences`’ ↵ {*equiv-dec*}

equiv-dec ::= ‘`forall`’ *quants* ‘`.`’ *msg* ‘`=`’ *msg* ↵

quants ::= *quant* {‘`;`’ *quant*}

quant ::= *identifier* {‘`,`’ *identifier*}
[‘`:`’ *type-name*]

C.8 Actual variables section

act-var-section ::= ‘`#Actual variables`’ ↵ {*act-dec*}

act-dec ::= *act-var-dec* | *timestamp-def* |
maxruntime-def | *act-inv-keys-dec*

act-var-dec ::= *identifier* {‘`,`’ *identifier*} ‘`:`’
type-name [*di-tag*] [*subtype-expr*] ↵

subtype-expr ::= ‘`[`’ *identifier* {‘`,`’ *identifier*} ‘`]`’

di-tag ::= ‘`External`’ |
‘`InternalKnown`’ |
‘`InternalUnknown`’

timestamp-def ::= ‘`TimeStamp`’ ‘`=`’ *time* ‘`..`’ *time* ↵

maxruntime-def ::= ‘`MaxRunTime`’ ‘`=`’ *time* ↵

act-inv-keys-dec ::= *act-inv-key-pair* {‘`,`’ *act-inv-key-pair*} ↵

act-inv-key-pair ::= ‘`(`’ *identifier* ‘`,`’ *identifier* ‘`)`’

C.9 Functions section

functions-section ::= '`#Functions`' ↵ {*functions-line*}
functions-line ::= *explicit-function-line* | *symbolic-line*
explicit-function-line ::= *fn-def-lhs* '`=`' *identifier* ↵
fn-def-lhs ::= *identifier* '`(`' *fn-def-arg*
{'`,`' *fn-def-arg*} '`)`'
fn-def-arg ::= *identifier* | '`_`'
symbolic-line ::= '`symbolic`' *identifier*
{'`,`' *identifier*} ↵

C.10 System section

system-section ::= '`#System`' {*agent-dec*} [*withdraw-dec*][*generate-system*]

agent-dec ::= *instance-dec* {'`;`' *instance-dec*} ↵
instance-dec ::= *process-name* '`(`' *identifier*
{'`,`' *identifier*} '`)`'
withdraw-dec ::= '`WithdrawOption`' '`=`' ('`True`' | '`False`') ↵
generate-system ::= '`GenerateSystem = True`'
| '`GenerateSystemForRepeatSection=`'
line-no '`to`' *line-no*

C.11 Intruder section

intruder-section ::= '`#Intruder Information`' ↵
'`Intruder`' '`=`' *identifier* ↵
'`IntruderKnowledge`' '`=`'
'`{`' *atom* {'`,`' *atom*} '`}`' ↵
[*internal-proc-dec*]
[*stale-knowledge-dec*]
[*crackable-dec*]
[*guessable-dec*]
['`UnboundParallel = True`']
{*deduction*}
deduction ::= '`forall`' *quants* '`.`'
msg {'`,`' *msg*} '`|-`' *msg* ↵

internal-proc-dec ::= ‘`IntruderProcesses`’ ‘`=`’
process-name {‘`,`’ *process-name*} ↵

stale-knowledge-dec ::= ‘`StaleKnowledge`’ ‘`=`’
(‘`True`’ | ‘`False`’) ↵

crackable-dec ::= ‘`Crackable`’ ‘`=`’ *crackable-type*
{‘`,`’ *crackable-type*} ↵

crackable-type ::= *type-name* [‘`(`’ *time* ‘`)`’]

guessable-dec ::= ‘`Guessable`’ ‘`=`’ *type-name*
{‘`,`’ *type-name*} ↵

C.12 Secure channels section

channels-section ::= ‘`#Channels`’ ↵
({*old-channel-spec*} | {*channel-spec*})

old-channel-spec ::= ‘`authenticated`’ ↵ |
‘`secret`’ ↵ |
‘`direct`’ ↵

channel-spec ::= *msg-channel-spec* | *session-channel-spec*

msg-channel-spec ::= *line-no msg-channel-prop* ↵

msg-channel-prop ::= [‘`C`’] [‘`NF`’] [‘`NRA`’ | ‘`NRA-`’] [‘`NR`’ | ‘`NR-`’]

session-channel-spec ::= (‘`Session`’ | ‘`Stream`’)
[‘`injective`’ | ‘`symmetric`’]
line-no {‘`,`’ *line-no*} ↵

C.13 Simplifying transformations script

simplifying-script ::= *free-vars-section*
processes-section
prot-desc-section
spec-section
simplifying-section

C.14 Simplifications section

simpl-section ::= ‘`#Simplifications`’ ↵ {*simpl*}

simpl ::= ‘`RemoveFields`’ ‘`[`’ *type-name* {‘`,`’ *type-name*}
‘`]`’ ↵ |
‘`RemoveHashedFields`’ *hash-type* ↵ |
‘`RemoveEncryption`’ *encrypt-type* ↵ |
‘`RemoveHash`’ *hash-type* ↵ |
‘`SwapPairs`’
‘`(`’ *type-name* ‘`,`’ *type-name* ‘`)`’ ↵ |
‘`Coalesce`’
‘`(`’ *type-name* ‘`,`’ *type-name* ‘`)`’ ↵ |
‘`RemovePlainAndEnc`’ ↵

encrypt-type ::= ‘`{`’ *type* ‘`}{`’ *type-name* ‘`}`’

hash-type ::= *identifier* ‘`(`’ *type* ‘`)`’

type ::= *type-name* | *encrypt-type* |
hash-type | *type* ‘`(+)`’ *type*