Casper: A Compiler for the Analysis of Security Protocols

Gavin Lowe*

July 6, 1998

Abstract

In recent years, a method for analyzing security protocols using the process algebra CSP [8] and its model checker FDR [24] has been developed. This technique has proved remarkably successful, and has been used to discover a number of attacks upon protocols. However, the technique has required producing a CSP description of the protocol by hand; this has proved tedious and error-prone. In this paper we describe Casper, a program that automatically produces the CSP description from a more abstract description, thus greatly simplifying the modelling and analysis process.

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 [8] and its model checker FDR [24] has been developed [25, 14, 18]. 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;

*Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK; e-mail gavin.lowe@mcs.le.ac.uk

- 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 [14, 18]. 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.

In the next section we describe the syntax used to specify protocols. In Section 3 we briefly discuss how Casper models protocols in CSP, and describe how the output from FDR can be interpreted. In Section 4 we illustrate the tool, by showing how to apply it to the Wide-mouthed-frog Protocol [3]: in so doing, we discover three different attacks. We sum up in Section 5.

2 The Casper syntax

In this section we present the syntax for Casper scripts. We use EBNF, writing literals in teletype font within quotes; linebreaks are significant within Casper scripts, and are represented by "↵". Logical lines may be split across two or more lines by ending lines with "\". Lines beginning with "--" are comments.

A complete example script is included in Appendix A.

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, via some kind of inductive proof, is the topic of current research; see [14, 17] 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, and a specification of what the protocol is supposed to achieve.
- 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.

These two parts are further split into four sections each, giving eight sections in total:

$$\begin{array}{rl} \textit{script} ::= & \textit{free-vars-section processes-section} \\ & \textit{prot-desc-section spec-section} \\ & \textit{act-var-section } [\textit{functions-section}] \\ & \textit{system-section intruder-section} \end{array}$$

We give a brief overview of these sections here; they are described in more detail below.

Free variables section declares the types of the free variables and functions used in the definition of the protocol;

Processes section declares the agents taking part in the protocol, and gives information about their initial state;

Protocol description section defines the protocol itself, giving the sequence of messages that constitutes a normal protocol run;

Specification section defines the requirements of the protocol;

Actual variable section declares the atomic datatypes used in the system to be checked with FDR;

Functions section gives definitions for the functions used in the protocol;

System section defines the system that is to be checked, in terms of the number and types of agents;

Intruder section gives the identity and initial knowledge of the intruder.

Each section is headed by a line beginning with "`#`".

2.2 Free variables section

The free variables section of the input file defines the types of all variables and functions used in the protocol, along with information about which keys are inverses of one another.

$$\begin{aligned} \textit{free-vars-section} ::= {} & \text{`}\texttt{\#Free variables}\text{'} \hookleftarrow \\ & \{\textit{var-dec} \mid \textit{inv-keys-dec}\} \end{aligned}$$

Variables and functions are declared using Pascal-like syntax, extended to allow the declaration of functions:

$$\begin{aligned} \textit{var-dec} &::= \textit{identifier}\ \{\text{`}\texttt{,}\text{'}\ \textit{identifier}\}\ \text{`}\texttt{:}\text{'}\ \textit{type-expr} \hookleftarrow \\ \textit{identifier} &::= \textit{letter}\ \{\textit{letter} \mid \textit{digit}\} \\ \textit{type-expr} &::= \textit{type-name} \mid \\ & \qquad \textit{type-name}\ \{\text{`}\texttt{x}\text{'}\ \textit{type-name}\}\ \text{`}\texttt{->}\text{'}\ \textit{type-name} \end{aligned}$$

For example:

```
#Free variables
A, B : Agent
sharedKey : Agent x Agent -> SKey
```

With two exceptions, the names given to the different types are not significant: the CSP model of the system does not depend upon any special properties of nonces, public keys, etc., so there is no need for Casper to "know" about these types; this means that the user can choose his own names for the types. The exceptions to this rule are:

- The type `TimeStamp` represents timestamps. Timestamps are modelled in Casper by natural numbers. In order to be as general as possible, we make no assumptions about the size of a time unit compared with the time taken to send a message: several messages may occur within the same time unit, or several time units may elapse between consecutive messages. We discuss particular aspects of timestamps as they crop up throughout this section, and draw the threads together in Section 3.6.

- The type `HashFunction` represents cryptographic hash functions.

Inverse keys are declared by listing pairs representing keys that are inverses of one another:

$$\begin{aligned}
\textit{inv-keys-dec} &::= \text{`}\texttt{InverseKeys =}\text{'}\ \textit{inv-key-pair}\ \{\text{`}\texttt{,}\text{'}\ \textit{inv-key-pair}\}\ \hookleftarrow \\
\textit{inv-key-pair} &::= \text{`}\texttt{(}\text{'}\ \textit{identifier}\ \text{`}\texttt{,}\text{'}\ \textit{identifier}\ \text{`}\texttt{)}\text{'}
\end{aligned}$$

Each pair may be either:

- a pair of variables, representing that the variables themselves are inverses of one another; or
- a pair of function identifiers, representing that when the functions are applied to the same argument, they return keys that are inverses of one another; for example, a pair `(PK, SK)` would represent that $PK(A)$ and $SK(A)$ are inverses of one another, for every agent A.

In a system using symmetric key cryptography, symmetric keys would be defined as being self-inverse.

2.3 Processes section

The processes section contains lines giving information about each agent taking part in the protocol:

$$\begin{aligned}
\textit{processes-section} &::= \text{`}\texttt{\#Processes}\text{'}\ \hookleftarrow\ \{\textit{process-def}\} \\
\textit{process-def} &::= \textit{process-name}\ \text{`}\texttt{(}\text{'}\ \textit{identifier}\ \{\text{`}\texttt{,}\text{'}\ \textit{identifier}\}\ \text{`}\texttt{)}\text{'} \\
&\qquad [\textit{knows-statement}]\ \hookleftarrow \\
\textit{knows-statement} &::= \text{`}\texttt{knows}\text{'}\ \textit{atom}\ \{\text{`}\texttt{,}\text{'}\ \textit{atom}\} \\
\textit{atom} &::= \textit{identifier}\ \mid\ \textit{fn-app} \\
\textit{fn-app} &::= \textit{identifier}\ \text{`}\texttt{(}\text{'}\ \textit{identifier}\ \{\text{`}\texttt{,}\text{'}\ \textit{identifier}\}\ \text{`}\texttt{)}\text{'}
\end{aligned}$$

For example:

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

The first parameter of each process (here `A` and `B`) should represent agent identities used in the protocol description (see below); the process line gives information about that agent, as follows:

- 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.
- The parenthesized 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. In the example above, the initiator A is expected to know his own identity A, the nonce na, the public

```
#Protocol description
0.    -> A : B
1. A -> S : {na, B}{shared(A,S)}
2. S -> B : {kab, na, A, ts}{shared(B,S)}, \
            {f(na), kab}{shared(A,S)} % v
[ts <= now and now <= ts+2]
3. B -> A : v % {f(na), kab}{shared(A,S)}, \
            {na (+) nb}{kab}
4. A -> B : {m, f(nb)}{kab}
5. B ->   : m
```

Figure 1: An example protocol description

key function PK (i.e. he can look up public keys in some table), and his own secret key $SK(A)$. 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 (made precise below).

- Later (under the `#System` heading, below), we will instantiate the parenthesized parameters with actual values, so as to define a system, and define the data values that each agent should use in their runs.

The fields that follow the `knows` keyword are intended to represent variables and function identifiers for which the agent uses the same value in every run; such values might be a function of the agent's identity, as is the case with $SK(A)$, above. On the other hand, the parameters inside the parentheses—with the exception of the identity itself—are intended to represent data items that might be instantiated differently in different runs; for example, the agents will use different nonces in different runs.

2.4 Protocol description section

The protocol itself is defined by listing the steps in order:

prot-desc-section ::= '`#Protocol description`' ↵
{*prot-msg* | *env-msg-send* | *env-msg-rec*}

A (somewhat contrived, so as to illustrate several features) example is in Figure 1.

Most of the steps will correspond to protocol messages, optionally preceded by an assignment and/or followed by a test (see below); the protocol message itself is written using an ascii version of the standard notation.

prot-msg ::= [*assignment-line* ↵]
line-no '`.`' *identifier* '`->`' *identifier* '`:`' *msg* ↵
[*test-line* ↵]

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.

Such messages may be preceded by one or more assignments, written between angle brackets; these represent cases where the sender of the message assigns values to variables, which he uses in the subsequent message. (Use of assignments has largely been superseded by the %-notation—see below—but they are still useful in a few cases.)

$$
\begin{aligned}
\textit{assignment-line} &::= \text{`}\texttt{<}\text{'}\ \textit{assignment}\ \{\text{`}\texttt{;}\text{'}\ \textit{assignment}\}\ \text{`}\texttt{>}\text{'} \\
\textit{asssignment} &::= \textit{identifier}\ \text{`}\texttt{:=}\text{'}\ \textit{fdr-expression}
\end{aligned}
$$

The type *fdr-expression* represents all expressions accepted by FDR[1]; the syntax is basically that of a simple functional language; see [6] for details.

Messages may be followed by tests, written between square brackets. These represent boolean tests performed by the receiver of the message; if the test fails (evaluates to *false*) then the receiver aborts the run.

$$
\begin{aligned}
\textit{test-line} &::= \text{`}\texttt{[}\text{'}\ \textit{test}\ \text{`}\texttt{]}\text{'} \\
\textit{test} &::= \textit{fdr-expression}
\end{aligned}
$$

Within tests, the distinguished variable `now` represents the current time, and is used in comparisons with timestamps; for example, the test following message 2 of Figure 1 is a test that the timestamp received is no more than two time units old.

Some steps in the protocol description correspond to messages received from a user or the environment; these are written analogously to normal protocol messages, except without a sender field; they may be followed by a test that is to be performed by the recipient of the message.

$$
\begin{aligned}
\textit{env-msg-rec} ::= {} & \textit{line-no}\ \text{`}\texttt{.}\text{'}\ \text{`}\texttt{->}\text{'}\ \textit{identifier}\ \text{`}\texttt{:}\text{'}\ \textit{msg} \hookleftarrow \\
& [\textit{test-line} \hookleftarrow]
\end{aligned}
$$

For example, message 0 of Figure 1 is used to start the protocol off, and tells A the identity of the agent with whom he should run the protocol.

Similarly, steps in the protocol description can correspond to messages sent to a user or the environment (for example, message 5 of Figure 1); these are written without a receiver field, and may be preceded by an assignment.

$$
\begin{aligned}
\textit{env-msg-send} ::= {} & [\textit{assignment-line} \hookleftarrow] \\
& \textit{line-no}\ \text{`}\texttt{.}\text{'}\ \textit{identifier}\ \text{`}\texttt{->}\text{'}\ \text{`}\texttt{:}\text{'}\ \textit{msg} \hookleftarrow
\end{aligned}
$$

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

The syntax of the body of messages is defined as follows:

$$\begin{aligned} msg ::= {} & identifier \mid \\ & identifier \text{ `(' } msg \text{ `)' } \mid \\ & msg \text{ `,' } msg \mid \\ & \text{`\{' } msg \text{ `\}\{' } atom \text{ `\}' } \mid \\ & msg \text{ `(+)' } msg \mid \\ & msg \text{ `\%' } identifier \mid \\ & identifier \text{ `\%' } msg \mid \\ & \text{`(' } msg \text{ `)'} \end{aligned}$$

We discuss each of the terms below, and explain how each is treated by the sender and receiver. First, we need a definition; we define the *agent's knowledge* at a particular point in the protocol run to be the set of variables the agent has seen so far: those variables included in the initial state, those received in subsequent messages, or those obtained as the result of an assignment. Casper performs various sanity checks on the protocol description, to ensure that the sender's knowledge includes all the variables needed to produce the message, and that the receiver's knowledge contains all the information needed to understand the message (e.g. appropriate decrypting keys).

The terms of the datatype of messages are as follows:

- A variable `v` simply represents the sender sending the value that he currently associates with that variable. If the variable is not declared to be a timestamp, then the variable should already be in the sender's knowledge; if the variable is declared as a timestamp, then it will be instantiated with the current time.

  If the variable is already in the receiver's knowledge, then he will only accept the corresponding value; otherwise he will accept any value (and store this value for subsequent use).

- In f is declared to be a hash function in the free variables section, then `f(m)` represents the application of f to message m. In this case, 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 (for example, in message 4 of Figure 1, B will only accept a message containing a hash of the nonce he sent out in message 3). It is assumed that all hash functions are known to all agents.

  If f is not declared to be a hash function, then f should only be applied to tuples of variables whose types match the type of the domain of f as declared in the free variables section. Both the sender and receiver should be able to perform this function application—i.e. f and each of the argument variables should be in their knowledge. The receiver will only accept a value that matches the value he calculates himself.

- `m , m'` represents the sequencing of messages m and m'.

- `{m}{k}` represents message m encrypted with key k, i.e. $\{m\}_k$; k may be either a variable or a function application. The receiver's knowledge should contain the corresponding decryption key, as defined in the free variables section; the receiver will only accept a message that is encrypted with the expected key.

- `m (+) m'` represents the bit-wise exclusive-or (also known as Vernam encryption) of m and m'. The receiver should be able to create at least one of m and m' so as to obtain the other (for example, in Message 3 of Figure 1, A knows na so can extract nb).

- In some protocols, the sender and receiver of a message treat the message somewhat differently. For example, some protocols contain an encrypted message that the recipient isn't supposed to decrypt, but instead forward to a third party in a subsequent message (for example, the component encrypted with $shared(A, S)$ in messages 2 and 3 of Figure 1). Also, in a public key distribution protocol, if key server S wants to send A's public key to B, then S will produce the key by application of an appropriate "public key" function to A's identity, whereas B will accept any value; thus S treats the key as the result of a function application, but B treats it as a variable.

  Such situations are handled using the %-notation. If m is a message and v is a variable, then $m\%v$ represents that the sender creates and sends m, but the receiver simply stores this message in variable v, without trying to interpret it. Similarly, $v\%m$ represents that the sender sends the contents of variable v, but the receiver interprets this as being of the form given by m.

  In fact, in the $m\%v$ case, above, the receiver should be willing to accept any message of the appropriate number of bits. We model this in Casper by allowing the agent to accept either any message of the same type as m, or a special value `Garbage` representing a random sequence of bits invented by the intruder.

  For example, the situation concerning public keys, described above, can be modelled by:

  ```
  1. B -> S : A
  2. S -> B : {PK(A) % pka}{SK(S)}
  ```

  where PK is the "public key" function, and the encryption with $SK(S)$ acts as a signature; in subsequent messages, B will use pka. Note that the above does not guarantee that the value B receives for pka will necessarily be the same as $PK(A)$: B will be willing to accept any value (of the appropriate type); indeed, there is an attack upon the above protocol where B will end up accepting the wrong key.

- Parentheses may be used to group messages.

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

Finally, the line numbers may be any alphanumeric strings (not necessarily numbers), with different line numbers for different messages.

2.5 Specification section

The specification section of the input script is used to specify the requirements of the protocol; each line represents a particular requirement.

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

There are ten different forms of specification available:

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 ::= $\mathbf{N}$

We discuss each of the specification forms in turn.

- `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 [17].)

- `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, [v`$_1$`,...,v`$_\mathrm{n}$`])` 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, [v`$_1$`,...,v`$_\mathrm{n}$`])` is a timed version of `Agreement(A, B, [v`$_1$`,...,v`$_\mathrm{n}$`])` 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.

- The specifications `TimedNonInjectiveAgreement(A, B, t, [v`$_1$`,..., v`$_\mathrm{n}$`])`, `TimedWeakAgreement(A, B, t)` and `TimedAliveness(A, B, t)` are similarly timed versions of `NonInjectiveAgreement(A, B, [v`$_1$`,...,v`$_\mathrm{n}$`])`, `WeakAgreement(A, B)` and `Aliveness(A, B)`.

The different authentication specifications are discussed in more detail in [16].

Our treatment of time is described in Section 3.6.

2.6 Actual variables section

The actual variables section contains information defining the actual datatypes to be used in the model checking of a system running the protocol.

$$\begin{aligned} \textit{act-var-section} &::= \text{`}\texttt{\#Actual variables}\text{'} \hookleftarrow \ \{\textit{act-dec}\} \\ \textit{act-dec} &::= \textit{act-var-dec} \mid \textit{timestamp-def} \mid \\ &\quad\ \ \textit{maxruntime-def} \mid \textit{act-inv-keys-dec} \end{aligned}$$

Every type that appears in the free variables section—except for timestamps, and those types that appear only as the result of functions that are declared symbolic (see below)—is defined by explicitly declaring values to be of that type:

$$\textit{act-var-dec} ::= \textit{identifier} \ \{\text{`}\texttt{,}\text{'} \ \textit{identifier}\} \ \text{`}\texttt{:}\text{'} \ \textit{type-name} \ \hookleftarrow$$

If timestamps are used in the protocol, then the range that these timestamps can take must be declared here; this range should be a subinterval of the natural numbers:

$$\textit{timestamp-def} ::= \text{`}\texttt{TimeStamp}\text{'} \ \text{`}\texttt{=}\text{'} \ \textit{time} \ \text{`}\texttt{..}\text{'} \ \textit{time} \ \hookleftarrow$$

It is often desirable to make an assumption about the implementation, namely that if any run lasts for a long time, then the agent involved will timeout and abort the run. This is necessary if any of the timed specifications are to be satisfied. The maximum allowable duration of any run is defined here as the variable `MaxRunTime`:

$$\textit{maxruntime-def} ::= \text{`}\texttt{MaxRunTime}\text{'} \ \text{`}\texttt{=}\text{'} \ \textit{time} \ \hookleftarrow$$

Key inverses are declared analogously to as in the free variable section, using a list of pairs of keys that are inverses of one another:

$$\begin{aligned} \textit{act-inv-keys-dec} &::= \textit{act-inv-key-pair} \ \{\text{`}\texttt{,}\text{'} \ \textit{act-inv-key-pair}\} \ \hookleftarrow \\ \textit{act-inv-key-pair} &::= \text{`}\texttt{(}\text{'} \ \textit{identifier} \ \text{`}\texttt{,}\text{'} \ \textit{identifier} \ \text{`}\texttt{)}\text{'} \end{aligned}$$

Such inverses need be declared only for values declared in this section; keys that are formed by the application of functions inherit their inverses from the declaration in the free variables section.

Normally, the different types defined in this section will be disjoint. However, some protocols are susceptible to type flaw attacks, where data of one type is interpreted as being of a different type within an attack. Such attacks may be found by including the same value within two types, for example:

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

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.

2.7 Functions section

The functions section is used to give more information about each of the functions declared in the free variables section. This information can take one of two forms:

- An explicit definition can be given for the function, for example:

  ```
  #Functions
  PK(Alice) = PKa
  PK(Bob) = PKb
  PK(Mallory) = PKm
  ```

  In this case, each of the values appearing on the right hand sides should have been declared in the actual variables section. The arguments to the functions may optionally be an underscore character "_", acting as a wild-card.

- The function can be declared as *symbolic*. This means that Casper produces its own values to represent the results of function applications, rather than insisting that these values are supplied by the user. One effect is that all symbolic functions are injective.

  Formally:

$$
\begin{aligned}
\textit{functions-section} &::= \text{`}\texttt{\#Functions}\text{'} \hookleftarrow \{\textit{functions-line}\} \\
\textit{functions-line} &::= \textit{explicit-function-line} \mid \textit{symbolic-line} \\
\textit{explicit-function-line} &::= \textit{fn-def-lhs}\ \text{`}\texttt{=}\text{'}\ \textit{identifier} \hookleftarrow \\
\textit{fn-def-lhs} &::= \textit{identifier}\ \text{`}\texttt{(}\text{'}\ \textit{fn-def-arg}\ \{\text{`}\texttt{,}\text{'}\ \textit{fn-def-arg}\}\ \text{`}\texttt{)}\text{'} \\
\textit{fn-def-arg} &::= \textit{identifier} \mid \text{`}\_\text{'} \\
\textit{symbolic-line} &::= \text{`}\texttt{symbolic}\text{'}\ \textit{identifier}\ \{\text{`}\texttt{,}\text{'}\ \textit{identifier}\} \hookleftarrow
\end{aligned}
$$

Symbolic functions are used more frequently, because they do not require the user to explicitly declare the values that are returned by the functions, and so require less extra baggage. However, the functions should be declared explicitly when it is required that some extra relationships hold between the values returned. For example, the following defines a function that returns a key shared between two agents, where the function is symmetric in its arguments:

```
shared(Alice, Bob) = Kab
shared(Bob, Alice) = Kab
shared(Alice, Mallory) = Kam
shared(Mallory, Alice) = Kam
shared(Bob, Mallory) = Kbm
shared(Mallory, Bob) = Kbm
shared(_, _) = K
```

(The last line is a catch-all, to cover the cases where the two arguments are the same, and is included so as to make the function total.)

2.8 System section

The system section is the most important section of the system definition. It defines which agents should be present in the system to be checked, by listing the agents, possibly combined together sequentially, with the parameters suitably instantiated:

system-section ::= '`#System`' {*agent-dec*}
agent-dec ::= *instance-dec* {';' *instance-dec*} ↵
instance-dec ::= *process-name* '(' *identifier* {',' *identifier*} ')'

For example:

```
#System
INITIATOR(Alice, Na1) ; INITIATOR(Alice, Na2)
RESPONDER(Bob, Nb1) ; RESPONDER(Bob, Nb2)
```

Where an agent is declared as a sequential composition of two or more instances, then each of these instances must have the same process name (e.g. `INITIATOR`, above) and agent identity (e.g. `Alice`).

Agents defined on different lines run concurrently. Two or more such agents may have the same agent identity, representing that the agent can run two or more copies of the protocol concurrently.

The types of the arguments should match the types of the parameters as defined in the processes section.

2.9 Intruder section

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

intruder-section ::= '`#Intruder Information`' ↵
'`Intruder`' '`=`' *identifier* ↵
'`IntruderKnowledge`' '`=`' '{' *atom* {'`,`' *atom*} '}' ↵

The intruder's knowledge can contain function identifiers, in which case the intruder can apply that function to any other value (of the appropriate type) that he knows.

3 The operation of Casper

The program Casper takes an input file as described in Section 2, and produces an output file, containing a CSP description of the system in question, suitable for model checking using FDR. In this section we give a brief overview of the way Casper represents protocols in CSP, and then explain how the output from FDR can be interpreted. We assume some familiarity with CSP [8, 26].

3.1 The *Encryption* datatype

The CSP model is based around a datatype—traditionally called *Encryption*—representing the set of messages that can be formed from the atoms by sequencing, encryption, application of hash functions, and Vernam encryption. For the input file for the Needham-Schroeder Public Key Protocol in Appendix A this is defined by:

```
datatype Encryption =
  Alice | Bob | Mallory | Na | Nb | Nm |
  PK_.Agent | SK_.Agent |
  Garbage | Sq.Seq(Encryption) |
  Encrypt.(ALL_KEYS, Seq(Encryption)) |
  Hash.(HashFunction, Seq(Encryption)) |
  Xor.(Encryption, Encryption)
```

The first six terms correspond to the atomic values defined in the `#Actual variables` section of the script. A data item of the form $PK\_.A$ represents the public key of A, written as `PK(A)` in the script: this is how we treat applications of symbolic functions. The term `Garbage` represents a random sequence of bits invented by the intruder; this will be accepted by an honest agent who does not try to interpret a message he receives (as specified by the %-notation). A data item of the form $Sq.\langle m_1, \ldots, m_n \rangle$ represents the concatenation of $m_1, \ldots, m_n$ ($Seq(X)$ represents the set of all sequences over X). A data item of the form $Encrypt.(k, \langle m_1, \ldots, m_n \rangle)$ represents the encryption $\{m_1, \ldots, m_n\}_k$. A data item of the form $Hash.(h, \langle m_1, \ldots, m_n \rangle)$ represents the application of hash function h to message $\langle m_1, \ldots, m_n \rangle$. A data item of the form $Xor.(m, m')$ represents the Vernam encryption $m \oplus m'$.

3.2 Channels

The CSP model of a system running the protocol uses eight channels:

`comm` is used for standard messages between two agents;

`fake` is used for messages faked by the intruder and received by an honest agent;

`intercept` is used for messages sent by an honest agent, but intercepted by the intruder;

`env` is used for messages sent to, or received from, a user or the environment;

`signal` is used for signalling information about the states and beliefs of honest agents:

- events of the form `signal.Claim_Secret.A.s.Bs` represents that A believes that s is a secret that only the agents Bs should learn;
- events of the form `signal.Running`n`.r.A.B.vs` represents that A believes he is running the protocol, taking role r, with B, using data items vs;
- events of the form `signal.Commit`n`.r.A.B.vs` represents that A believes he has completed a run of the protocol, taking role r, with B, using data items vs;

`leak` is used to signal that the intruder has learnt a particular fact;

`tock` is used to represent the passage of one unit of time;

`time` is used by honest agents to obtain the current time from a shared clock.

3.3 The honest agents

Each agent in the #Processes list is modelled by a process (with appropriate free variables); this process sends and receives messages, as in the message list, and performs appropriate signal events to indicate the agent's beliefs. For example, Casper produces the following process to represent the initiator A in the running example:

```
INITIATOR_0(A, na) =
  [] B : Agent @ env.A.(Env0, B) ->
  comm.A.B.(Msg1, Encrypt.(PK(B), <na,A>)) ->
  [] nb : Nonce @ comm.B.A.(Msg2, Encrypt.(PK(A), <na,nb>)) ->
  signal.Claim_Secret.A.na.{B} ->
  signal.Running1.INITIATOR_role.A.B.na.nb ->
  comm.A.B.(Msg3, Encrypt.(PK(B), <nb>)) ->
  signal.Commit2.INITIATOR_role.A.B.na.nb ->
  SKIP
```

($PK(A)$ is defined as syntactic sugar for $PK\_.A$.)

The channel comm represents normal communications between two agents. A renaming is applied to such a process, so that messages that are sent can also be intercepted (represented by the channel intercept), and so that messages that are received can also be faked (represented by the channel fake).

These processes are then instantiated as in the #System section, and combined together in parallel.

3.4 The intruder

The definition of the intruder is based around the set of facts that it might, conceivable, be able to learn: this is the set of all atoms, along with the set of all encrypted, hashed, or Vernam encrypted components that are submessages of messages of the protocol.

The intruder can learn a new fact f by overhearing or intercepting a message that contains f as a submessage (with f not further encrypted). The intruder can send a (possibly fake) message consisting of a sequence of facts if he knows all the constituent facts.

The intruder can also derive new facts according to the following derivation rules; a rule of the form $S \vdash f$ means that the intruder can derive fact f from the set of facts S (S includes those functions that the intruder knows, as defined in the input file):

$$
\begin{array}{ll}
\{k, f_1, \ldots, f_n\} \vdash \mathit{Encrypt}.(k, \langle f_1, \ldots, f_n \rangle) & \\
\{\mathit{Encrypt}.(k, \langle f_1, \ldots, f_n \rangle), k^{-1}\} \vdash f_i, & i = 1, \ldots, n \\
\{f, f'\} \vdash f \oplus f' & \\
\{f \oplus f', f\} \vdash f' & \\
\{f \oplus f', f'\} \vdash f & \\
\{f \oplus f'\} \vdash f' \oplus f & \\
\{f_1, \ldots, f_n\} \vdash \mathit{Hash}.(h, \langle f_1, \ldots, f_n \rangle), & \text{for hash function } h \\
\{g, f\} \vdash g(f), & \text{for function } g
\end{array}
$$

The first rule represents that the intruder can encrypt a sequence of known facts with a known key to produce a new fact; the second rule represents that the intruder can decrypt known facts with an appropriate known key (k^{-1} is the decrypting key corresponding to encrypting key k) to learn new facts; the third rule represents Vernam encryption; the fourth and fifth rules represent Vernam decryption; the sixth rule models the commutativity property of Vernam encryption; the seventh rule represents that the intruder can apply hash functions; the eighth rule represents that the intruder can apply functions he knows.

The above set of rules does not capture all properties of Vernam encryption; a proper treatment would require the modelling of algebraic equivalences, which we briefly discuss in the concluding section.

If a particular fact is supposed to remain secret, and the intruder learns it, then he may leak it on the channel $leak$.

When FDR is used to analyze a system, it begins by building an explicit representation of each atomic process. In a system with n possible facts, the intruder process will have 2^n states. It is typical to deal with a system with a few hundred facts, so building such a large process is impractical. Instead, the intruder is built up from the parallel combination of n atomic processes, one process for each fact; each such process has two states, corresponding to the cases where the intruder does or does not know the fact. The process representing fact f will synchronize on several events:

- It will synchronize on the overhearing or interception of messages containing f, sent by other agents, so as to learn the fact f (events of the form $comm.A.B.(Msgn, f)$, $intercept.A.B.(Msgn, f)$, $comm.A.B.(Msgn, Sq.fs)$, or $intercept.A.B.(Msgn, Sq.fs)$ for f in fs);
- It will synchronize on the sending of messages that contain f as a submessage (events of the form $fake.A.B.(Msgn, f)$, or $fake.A.B.(Msgn, Sq.fs)$ for f in fs);
- It will synchronize on events representing the derivation of f from other facts, so as to learn f (events of the form $derive.(f, S)$ where $S \vdash f$);
- It will synchronize on events representing the derivation of other facts from f (events of the form $derive.(f', S)$ where $S \vdash f'$ and $f \in S$);

- If the fact is supposed to remain secret, and the intruder learns it, then it will signal this fact (via a $leak.f$ event).

By synchronizing the components representing different facts appropriately, we can ensure that the intruder "learns" new facts according to the above deduction rules. This technique for modelling the intruder is due to A. W. Roscoe [27, 26].

3.5 Specifications and assertions

Casper generates one refinement assertion to check all the secret specifications in the input file: the specification process specifies that if an agent signals, via a $signal.Claim\_Secret$ event, that he believes that a particular fact f will remain secret from the intruder, then the intruder will never learn f, i.e. will never perform a $leak.f$ event (the $signal.Claim\_Secret$ follows the first message containing f for `StrongSecret` specifications, and follows the last message the agent receives for `Secret` specifications, thus distinguishing these specifications).

Casper also generates a refinement assertion for each authentication specification in the input file, for each agent of the appropriate type: the specification process specifies that if an agent believes he has completed a protocol run (i.e. he performs a $signal.Commit$ event), then the other agent involved previously thought he was running the protocol (i.e. he performed a $signal.Running$ event). Further details are in [16].

3.6 About time

A treatment of time is necessary to handle two features of security protocols: timestamps and timed authentication specifications. We give a brief overview of our treatment of time, here.

We assume that the honest agents have access to a reliable, secure clock. We therefore do not consider attacks caused by the intruder changing the honest agents' clocks in some way; see [7] for a discussion of such attacks. We assume the clock is shared between the agents, although equally we could give them individual, synchronized clocks.

We model the clock as a CSP process, with an integer valued parameter representing the current time. The process can:

- perform the event $tock$, representing the passage of one unit of time, and update the current time appropriately, up to the maximum time specified in the `#Actual variables` section;
- synchronize on any protocol message that contains a timestamp representing the current time;
- pass the current time to any honest agent on the channel $time$.

When an honest agent sends a message containing a timestamp, this event will be synchronized with the clock, thus ensuring that only messages containing the current time can be sent.

When an honest agent performs a test that uses the distinguished variable *now*, representing the current time, the process obtains this time from the clock on the channel *time*.

In order to make the implementation assumption about no run lasting more than $MaxRunTime$ time units, the untimed definitions of the honest agents are lifted to timed versions in such a way that if more than $MaxRunTime$ tocks occur then the process is interrupted, representing that the run is aborted.

The timed authentication specifications are defined similarly to the untimed ones, above, except that the $signal.Commit$ event must occur within the specified number of *tock*s of the corresponding $signal.Running$ event.

3.7 Interpreting the output from FDR

The CSP model of the protocol is such that most events are hidden at the top level; the only events visible are the `signal` and `leak` events that are necessary for specifying the security property being tested for. When an attack is found by FDR, the debugger can be used to explore the process tree (see the FDR manual [6] for details) to reveal the sequence of communications.

For example, when Casper is used to compile the running example script into CSP, and FDR is used to test whether the protocol correctly authenticates initiator Alice, then it finds that the check fails, giving the following trace:

```
<signal.Running1.INITIATOR_role.Alice.Mallory.Na.Nb
 signal.Commit1.RESPONDER_role.Bob.Alice.Na.Nb>
```

The first event signifies that the initiator Alice thought she was running the protocol with Mallory, using nonces Na and Nb; the second event signifies that the responder Bob thought he had completed a run of the protocol with Alice, using nonces Na and Nb, and so commits to a subsequent session. Clearly the cause of the error is this second event: Bob thought he had completed a run of the protocol with Alice, but Alice didn't think she had been running the protocol with Bob.

Using the debugger reveals that the attack took the following form:

```
<env.Alice.(Env0, Mallory)
 comm.Alice.Mallory.(Msg1, Encrypt.(PKm, <Na, Alice>))
 fake.Alice.Bob.(Msg1, Encrypt.(PKb, <Na, Alice>))
 signal.Running2.RESPONDER_role.Bob.Alice.Na.Nb
 intercept.Bob.Alice.(Msg2, Encrypt.(PKa, <Na, Nb>))
 fake.Mallory.Alice.(Msg2, Encrypt.(PKa, <Na, Nb>))
 signal.Claim_Secret.Alice.Na.{Mallory}
 signal.Running1.INITIATOR_role.Alice.Mallory.Na.Nb
 comm.Alice.Mallory.(Msg3, Encrypt.(PKm, <Nb>))
 fake.Alice.Bob.(Msg3, Encrypt.(PKb, <Nb>))
 signal.Claim_Secret.Bob.Nb.{Alice}
 signal.Commit1.RESPONDER_role.Bob.Alice.Na.Nb>
```

The above attack can be re-written as follows, stripping out the signals and environmental communications:

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

where M_{Alice} represents the intruder Mallory imitating Alice; the attack uses two runs, which we have denoted α and β, writing for example $\alpha.2$ for message 2 of run α. This is the attack presented in [13, 14]. Briefly: in run β the intruder imitates Alice in a run of the protocol with Bob; in run α the intruder uses Alice as an oracle.

4 Case study: The Wide-mouthed-frog Protocol

In this section we illustrate some of the issues involved in modelling protocols by considering the example of the Wide-mouthed-frog Protocol of [3]. The protocol can be described as follows:

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

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; we discuss a few points below:

```
#Free variables
A, B : Agent
```

```
S : Server
SKey : Agent -> ServerKey
kab : SessionKey
ts1, ts2 : TimeStamp
InverseKeys = (SKey, SKey)

#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, ts1, kab}{SKey(A)}
[ts1==now or ts1+1==now]
2.  S -> B : {A, ts2, kab}{SKey(B)}
[ts2==now or ts2+1==now]

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

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.

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.

The specification is 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 specification assumes that there is negligible delay between S checking message 1 and sending message 2; we formalize this assumption by setting the `MaxRunTime` variable to 0, below). Further, the two agents should agree on the value of kab, and there should be a one-one relationship between the runs of A and the runs 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; we take all the datatypes to be as small as possible, consistent with this system.

```
#Actual variables
Alice, Bob, Mallory : Agent
```

```
Sam : Server
Kab : SessionKey
TimeStamp = 0 .. 0
MaxRunTime = 0

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

#Functions
symbolic SKey

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

We choose a very small (one point!) time domain, so as to make checking as fast as possible. We define the maximum length of any run—given by the variable MaxRunTime—to be 0 time units, so if any run lasts for longer than this time, then the agent involved will timeout and abort the run.

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

When the above file is compiled using Casper, and then checked using FDR, FDR fails to find any attack upon this small system. This test takes FDR less than five seconds.[2]

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. FDR finds that the system can perform the trace

```
<signal.Commit1.RESPONDER_role.Alice.Bob>
```

[2]All the timings in this section refer to running FDR under Free-BSD on a Pentium P-133 with 32Mb of RAM and 200Mb of swap space.

Alice commits to a session with Bob, even though Bob has not been running the protocol—indeed, Bob is completely absent.

By exploring the debug tree, we can find that this was caused by the following sequence of events:

```
env.Alice.(Env0, Bob)
signal.Running1.INITIATOR_role.Alice.Bob
comm.Alice.Sam.(Msg1, Encrypt.(Kas, <Bob, Timestamp.0, Kab>))
fake.Sam.Alice.(Msg2, Encrypt.(Kas, <Bob, Timestamp.0, Kab>))
time.0
signal.Commit1.RESPONDER_role.Alice.Bob
```

which can be rewritten in more conventional form:

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

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

FDR takes less than five seconds to find this attack.

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. FDR gives the following trace:

```
<signal.Running1.INITIATOR_role.Alice.Bob.Kab
 signal.Commit1.RESPONDER_role.Bob.Alice.Kab
 signal.Commit1.RESPONDER_role.Bob.Alice.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.

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

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

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

FDR takes about five seconds to find the above attack.

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. FDR produces the following trace:

```
<signal.Running1.INITIATOR_role.Alice.Bob.Kab
 tock
 tock
 tock
 signal.Commit1.RESPONDER_role.Bob.Alice.Kab>
```

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:

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

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 [1].

The above test takes approximately 30 seconds; this is longer than the previous tests, because the larger time domain makes for a larger message space, and so a more complicated intruder process, and a larger state space.

5 Conclusions

In this paper we have given an overview of the Casper compiler, for producing CSP descriptions of protocols. The 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.

Casper has proved easy enough to use that it has been the subject of several undergraduate projects.

The approach described in this paper has been applied to a number of other protocols, including the Andrew Protocol [3], the Kerberos Protocol [20, 3], the CCITT X.509 Protocol [3], the Yahalom Protocol [3], a number of ISO protocols [9, 10], the TMN Protocol [30], the Denning-Sacco public key protocol [4], the Woo and Lam protocols [31], and the SPLICE Protocol [32]. Some of these case studies are available via the Casper World Wide Web page [15]. The techniques seem to scale well to medium sized protocols, albeit with a reduction in the size of the system that can be studied.

We are currently applying these techniques to a large commercial protocol produced by Cybercash. This protocol contains a very large number of fields, and several levels of nested encryption; these do not seem to be necessary for the security of the protocol. We are investigating how we may analyze a simpler protocol, and still be sure that our results apply to the original protocol.

Casper has been designed so that the output is suitable for checking using FDR; however, it should prove reasonably straightforward to adapt it to produce output suitable for other protocol analysis tools. Similarly, the input language syntax, and much of the program's code, should prove useful in building completely different protocol analysis tools.

Casper does not yet cover all the features found in security protocols. However, it does provide a useful framework for investigating new ideas, and considering how to model new features within CSP. For example, when I first investigated how to model timestamps within protocols, I used Casper to produce a CSP model of the Wide-mouthed-frog Protocol, but where the $TimeStamp$ type was treated symbolically, the same as any other type; I then edited the CSP by hand so as to treat timestamps correctly. Similarly, when considering the different forms of authentication in [16], it was useful to be able to try them out by coding them up within a CSP protocol description produced by Casper. This approach is clearly easier than producing each CSP file from scratch.

5.1 Comparisons with CAPSL

Recently, a language called CAPSL [19] (Common Authentication Protocol Specification Language) has been suggested, to formally define the operation of protocols, and to act as the basis for input languages for protocol analysis tools. In this section we compare CAPSL with the Casper input language.

The main differences between the two languages result from the fact that a Casper input file has to define now only the protocol itself, but also the system to be checked; CAPSL defines just the protocol. This means that a CAPSL protocol description roughly corresponds to the first half of a Casper input file. One consequence of having to define the system to be checked is that when designing our input syntax we had to provide a mechanism for defining which agents were taking part in the system, which roles they played, and what data items they used: we did this by giving names to the roles and the free variables used (in the `#Processes` section), and then later instantiating these variables and the agents' identities (in the `#System` section). CAPSL does not provide such a mechanism, and it was not clear how to extend CAPSL appropriately.

It is fairly common to assume that a particular agent can adopt only some proper subset of the roles. For example, it is common to assume that a particular agent may run the protocol sometimes as initiator and sometimes as responder, but never as a key server. Casper allows one to make such assumptions, in the `#System` section, whereas there is no mechanism for doing this in CAPSL.

The Casper notation is more oriented towards trapping user errors than the CAPSL notation. For example, in a CAPSL protocol description, if an agent receives an encrypted message that it cannot decrypt, then it is assumed that this was the intention. This contrasts with the situation in Casper where an error message is generated; if it really were the intention that the message not be decrypted, then the %-notation should be used; my experience with using Casper has been that this has helped to catch many user errors.

There are various minor differences in concrete syntax. For example, a message such as $\{A, Kab\}_{Kb}$ is written as `{A, Kab}{Kb}` in Casper, but as `se(Kb, cat(A, Kab))` in CAPSL. Further, CAPSL differentiates between symmetric and asymmetric key encryption in the syntax for messages, whereas we differentiate between them in declaring the inverses of keys.

5.2 Forthcoming features

At present, Casper is able to deal with a very large class of protocols, including, for example, all the protocols in [3]. Below we discuss those aspects of security protocols that Casper cannot currently deal with, and discuss the prospects for including them.

One major class of protocol that Casper cannot currently deal with are those where there is not a simple linear sequence of messages. For example, some protocols such as those in [22, 11] are in two parts: an initial authentication phase and a repeated authentication phase, which may be repeated several times. Similarly, some commercial protocols, such as the Visa and Mastercard

Secure Electronic Transaction Protocol [29], allow a number of different operations; these different operations can almost be thought of as different protocols running on the same system. Modelling such protocols presents no particular difficulties: at the CSP level, it simply involves modelling each operation as a separate CSP process; at the Casper level, the biggest difficulty is finding an appropriate input format.

Algebra arises in various ways within security protocols. For example, the bit-wise exclusive-or operator has a number of algebraic properties, not all of which are caught by our current treatment. Similarly, RSA encryption [23] satisfies the identity $\{\{m\}_k\}_{k'} = \{\{m\}_{k'}\}_k$; as a result, some protocols can be attacked if RSA is used. Also, Diffie-Hellman [5] style key establishment involves the exchange of integers, after which both agents should agree on the same key; proving this fact requires the use of algebraic properties of the integers. Roscoe discusses how to model such algebraic properties in [26, 27].

In a similar vein, sometimes we would like to be able to define extra derivations that the intruder can make. For example, one might want to specify that the cryptosystem used is such that if the intruder knows an encrypted message and the corresponding plain text, then he can deduce the encrypting key. This could be specified by adding the following rule to the definition of the derivation relation in Section 3.4:

$$\{Encrypt.(k, \langle f_1, \ldots, f_n \rangle),\ f_1, \ldots, f_n\} \vdash k$$

I have recently added the ability to define such extra derivations, as a prototype feature.

Some forms of attacks cannot currently be found using Casper. In Section 2.6 we discussed how to detect some type flaws. The flaws discussed there were *simple*, in the sense that a single atom of one type is interpreted as being an atom of a different type. We would also like to be able to detect *compound* type flaws, where the concatenation of several atoms is interpreted as being a single atom, or vice versa; for example, there is an attack on the Otway-Rees protocol where the concatenation of a nonce and two agents' identities (all known to the intruder) could be accepted in the place of a key.

Ever since Denning and Sacco's attacks [4] on the Needham-Schroeder protocols [21], researchers in the security community have been interested in the question of whether the compromise of an old session key can lead to new attacks upon a protocol; this was the main reason for the development of BAN Logic [3]. It would therefore be useful to allow Casper to model key compromise. The way to do this would be to allow certain keys to be defined as compromisable: the definition of the intruder would have to be adapted so that if the intruder saw such a key being used, he would learn the value of that key, but only after some (user defined) delay.

So far, we have concentrated on protocols whose aims are establishing shared secrets or achieving entity authentication. Schneider [28] has shown how to formalize a number of other security properties, such as anonymity and non-repudiation, in CSP; it would be useful to incorporate these within Casper.

Some protocols aim to be resistant to attacks where an intruder is able to guess a poorly chosen password, for example [12, 2]. To analyze such protocols would involve altering the model of the intruder; doing this correctly presents an interesting challenge.

The other main topic of future research concerning the CSP/FDR analysis of security protocols is the question of which systems we should check in order to be able to deduce that a particular system is secure when run on an arbitrarily sized system; approaches to this question appear in [14, 17].

Acknowledgements

I would like to thank Bill Roscoe, Peter Ryan, Irfan Zakiuddan, Steve Schneider, Josh Guttman, Paul Gardiner, Michael Goldsmith, Simon Ambler and Mark Reilly for useful comments and discussions about Casper. I would also like to thank the anonymous referees for their comments and suggestions.

References

[1] 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.

[2] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Security Conference on Research in Security and Privacy*, pages 72–84, 1992.

[3] 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.

[4] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

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

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

[7] Li Gong. A security risk of depending on synchronized clocks. *ACM Operating Systems Review*, 26(1):49–53, 1992.

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

[9] ISO/IEC. Information tech-no-lo-gy—security techniques—entity authentication mechanisms part 2: Entity authentication using symmetric techniques, 1993.

[10] ISO/IEC. Information tech-no-lo-gy—security techniques—entity authentication mechanisms part 3: Entity authentication using a public key algorithm, 1995.

[11] A. Kehne, J. Schönwälder, and H. Landendörfer. A nonce-based protocol for multiple authentications. *Operating Systems Review*, 26(4):84–89, 1992.

[12] T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham. Reducing risks from poorly chosen keys. *Operating Systems Review*, 23(5):14–18, 1989. Proceedings of the Twelfth ACM Symposium on Operating Systems Principles.

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

[14] 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.

[15] Gavin Lowe. Casper: A compiler for the analysis of security protocols, 1996. World Wide Web home page at URL `http://www.mcs.le.ac.uk/~glowe/Security/Casper/index.html`.

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

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

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

[19] Jonathan Millen. CAPSL: Common authentication protocol specification language. Available via `http://www.mitre.org/research/capsl/`, 1996.

[20] S. P. Miller, C. Neumann, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Project Athena Technical Plan Section E.2.1, MIT, 1987. Available from URL `ftp://athena-dist.mit.edu/pub/kerberos/doc/techplan.PS`.

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

[22] B. Clifford Neuman and Stuart G. Stubblebine. A note on the use of timestamps as nonces. *Operating Systems Review*, 27(2):10–14, 1993.

[23] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

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

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

[26] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[27] A. W. Roscoe and M. H. Goldsmith. The perfect "spy" for model-checking cryptoprotocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Available via URL: `http://dimacs.rutgers.edu/Workshops/Security/program2/program.html`.

[28] Steve Schneider. Modelling security properties with CSP. Technical Report CSD-TR-96-04, Royal Holloway, University of London, 1996.

[29] Secure Electronic Transaction (SET) specification. Available from `http://www.visa.com/cgi-bin/vee/sf/set/intro.html?2+0`, 1996.

[30] 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.

[31] Thomas Y. C. Woo and Simon S. Lam. A lesson on authenticated protocol design. *Operating Systems Review*, 28(3):24–37, 1994.

[32] 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 script: the Needham-Schroeder Public Key Protocol

In this appendix we give an example Casper script, namely for the three message version of the well-known Needham-Schroeder Public Key Protocol. This script is sufficient to find the attack from [13].

```
-- 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)}
```