Analysing Security Protocols using CSP

Gavin LOWE
Oxford University Computing Laboratory

1. Introduction

In this chapter we describe how security protocols can be analysed using the process algebra CSP and the model checker FDR. The basic technique is to build a CSP model of a small system running the protocol, together with the most general intruder who can interact with that protocol, and then to use the model checker FDR to explore the state space, looking for insecure behaviours.

We will base our explanation of the technique upon the book's running example:

Message *1*. $a \rightarrow b \ : \ \{|[k]_{SK(a)}|\}^{\mathsf{a}}_{PK(b)}$
Message *2*. $b \rightarrow a \ : \ \{|s|\}^{\mathsf{s}}_{k}$

The initiator a creates a fresh session key k, signs it with her secret key $SK(a)$, encrypts it with the responder b's public key $PK(b)$, and sends it to b. The responder b then uses k to encrypt some secret value s to return to a.

We analyse the protocol in the context of the Dolev-Yao model [DY83]. We assume that the intruder has complete control of the network and so can: overhear messages passing on the network; intercept messages, to prevent them reaching their intended recipients; encrypt and decrypt messages using keys he knows, so as to learn new messages; and send messages that he knows, possibly using a false identity. However, we do not allow the intruder to perform cryptanalytic attacks: we effectively assume perfect cryptography.

In the next section we give a brief introduction to CSP. Then in Section 3 we produce a CSP model of the protocol: we build a model of a small system running the protocol, including a single initiator, Alice, and a single responder, Bob, together with the most general intruder who can interact with the protocol. In Section 4 we describe how the security properties of secrecy and authentication can be captured as CSP specifications, and how FDR can be used to find attacks against these properties. We then adapt the protocol to prevent these attacks. FDR finds no attacks when it is used to analyse a small system running the adapted protocol. However, this does not immediately imply that there is no attack against some larger system running the protocol. We tackle this issue in Section 5, by showing how to build a model that abstracts an arbitrary system running the protocol: any attack upon an arbitrary system running the protocol will be reflected in an attack upon this model. In Section 6 we describe Casper, a compiler that can be used to produce the CSP models from a more abstract description, and in Section 7 we give bibliographic notes and briefly discuss some extensions of the technique.

2. A brief overview of CSP

Communicating Sequential Processes (CSP) [Hoa85,Ros97] is a process algebra for describing programs or *processes* that interact with their environment by communication. That environment might be other processes running in parallel, or might be processes outside the system being modelled. In this section we give a brief overview of the fragment of the syntax that we use subsequently, and describe the trace semantics of CSP; for more details, see [Ros97].

CSP processes communicate via atomic events, from some set Σ. Typically, events correspond to communications over channels; for example the event $c.3$ represents the communication of the value 3 over the channel c. Each channel is declared to pass data of a particular type. The set of events over channel c is denoted $\{|c|\}$.

The simplest process is $STOP$, which represents a deadlocked process that cannot communicate with its environment. The process $a \rightarrow P$ offers its environment the event a; if the event is performed, it then acts like P. This is generalised by the process $c?x : X \rightarrow P(x)$, which is willing to do any event of the form $c.x$ for $x \in X$, and then acts like $P(x)$; this represents an input of x on channel c; if the set X is omitted, it is taken to be the type of the channel.

If b is a boolean and P is a process, then $b \ \& \ P$ represents the process that acts like P if b is $true$, and like $STOP$ if b is $false$; in other words, b acts as a boolean guard on P.

The process $P \ \Box \ Q$ can act like either P or Q: the environment is offered the choice between the initial events of P and Q. The process $\Box_{i:I} P(i)$ represents an indexed choice between the processes $P(i)$ for $i \in I$. $Chaos(A)$ is a totally nondeterministic process that can communicate arbitrary events from A.

The process $P \underset{A}{\parallel} Q$ runs P and Q in parallel, synchronising on events from A, i.e. events from A can occur only when both P and Q are willing to perform them. An indexed version is written as $\Big\|_{i:I}[\alpha_i]P_i$: each process P_i synchronises on events from its alphabet α_i. By contrast, $P \ ||| \ Q$ runs P and Q in parallel with no synchronisation.

The process $P \setminus A$ acts like P, except the events from A are hidden, i.e. turned into internal, invisible events, denoted τ. Finally, the process $P[\![a/b]\!]$ represents the process P but where every occurrence of the event b is renamed to a. An indexed version of this is written, for example, $P[\![c.x/d.x \mid x \in X]\!]$.

A *trace* is a sequence of events that a process can perform; we write $traces(P)$ for the set of traces of P. We say that P *is refined by* Q, written $P \sqsubseteq_T Q$, if all the traces of Q are traces of P:

$$P \sqsubseteq_T Q \quad \Leftrightarrow \quad traces(P) \supseteq traces(Q).$$

Typically, Q will be a model of a system and P will be a specification process that can perform those traces that meet some requirement; the refinement will hold if P can perform only traces that meet the requirement. The tool FDR [Ros94,For97] can be used to test such refinements automatically, for finite state processes.

3. Modelling the protocol

In this section we describe the basic technique of CSP model checking of security protocols. As noted in the introduction, we consider a small system running the protocol: we include a single initiator Alice, who will use the session key Ka, and a single responder Bob, who will use the secret Sb. We also include an intruder, Mallory, who has complete control over the network, as illustrated in Figure 1.

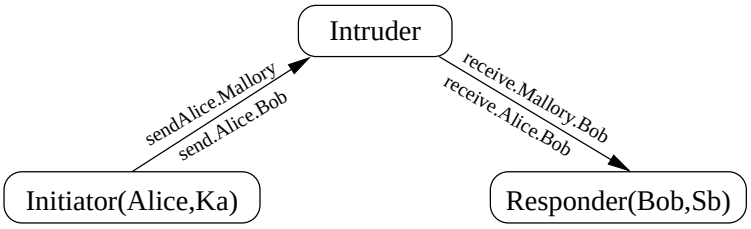

Figure 1. The system

In the next subsection we give some basic definitions of the underlying types, etc. In Subsection 3.2 we give CSP models for the honest participants in the protocol, and in Subsection 3.3 we give a model for the intruder. Finally, in Subsection 3.4 we put them together to form a model of the whole system.

We present the definitions using "blackboard CSP"; it is straightforward to translate this into machine-readable CSP, for use with FDR.

3.1. The basics

We begin by defining a datatype $Encryption$ representing the space of all possible messages. The datatype includes the atomic values indicated in Figure 1, together with a session key and a secret that the intruder will know initially. It includes constructors that correspond to concatenation and encryption; we do not distinguish between symmetric encryption, asymmetric encryption and signing, although it would be straightforward to include separate constructors for each of these. The datatype also includes constructors to give the public keys and secret keys of agents.

$$\begin{aligned}&\text{datatype } Encryption = \\ &\quad Alice \mid Bob \mid Mallory \mid Ka \mid Km \mid Sb \mid Sm \mid PK\_.Agent \mid SK\_.Agent \mid \\ &\quad Sq.Seq(Encryption) \mid Encrypt.(AllKeys, Encryption),\end{aligned}$$

where the set $AllKeys$ is defined below. For example, a typical instance of message 1 of the form $\{|[k]_{SK(a)}|\}^{\mathsf{a}}_{PK(b)}$ would be represented by the element $Encrypt.(PK\_.b, Encrypt.(SK\_.a, k))$ of $Encryption$.

For convenience, we define functions to return the public and secret keys of agents, and a number of subtypes of $Encryption$.

$$PK(a) = PK\_.a, \qquad SK(a) = SK\_.a,$$
$$Agent = \{Alice, Bob, Mallory\}, \qquad Honest = Agent - \{Mallory\},$$
$$Secret = \{Sb, Sm\}, \qquad SessionKey = \{Ka, Km\},$$
$$PublicKey = \{PK(a) \mid a \in Agent\}, \qquad SecretKey = \{SK(a) \mid a \in Agent\},$$
$$AllKeys = SessionKey \cup PublicKey \cup Secretkey.$$

We also define a function to return the inverse key of a given key:

$$inverse(Ka) = Ka, \qquad inverse(Km) = Km,$$
$$inverse(PK\_.a) = SK\_.a, \qquad inverse(SK\_.a) = PK\_.a.$$

We will represent each message by a $(Label, Encryption)$ pair (the $Label$ simply helps with interpretation of the output from FDR). We define Msg to be the set of all such pairs that are used in the protocol:

$$\text{datatype } Label = Msg1 \mid Msg2 \mid Env0,$$
$$Msg = \{(Msg1, Encrypt.(PK(b), Encrypt.(SK(a), k))) \mid k \in SessionKey, a \in Agent, b \in Agent\} \cup \{(Msg2, Encrypt.(k, s)) \mid k \in SessionKey, s \in Secret\}.$$

And we define channels to represent the sending and receiving of messages:

$$\text{channel } send, receive : Agent.Agent.Msg.$$

For example, agent a sending a message 1 of the protocol intended for agent b will be represented by the event $send.a.b.(Msg1, Encrypt.(PK(b), Encrypt.(SK(a), k)))$; agent b receiving the message will be represented by the event $receive.a.b.(Msg1, Encrypt.(PK(b), Encrypt.(SK(a), k)))$.

It is useful to model interactions between the protocol and its environment (e.g. commands from a user to run the protocol). Here we want to model the protocol for the initiator receiving a message from its environment, telling it to run the protocol with a particular agent b.

$$EnvMsg = \{(Env0, b) \mid b \in Agent\},$$
$$\text{channel } env : EnvMsg.$$

3.2. Modelling the honest agents

We now describe how we can model the honest agents running the protocol as CSP processes. We give a parametrised process $Initiator(a, k)$ to represent an agent a running the protocol as initiator, and using session key k. The process starts by receiving a message from the environment, telling it with whom to run the protocol. It then sends an appropriate message 1, and receives back an appropriate message 2 containing an arbitrary value for s.

$$Initiator(a, k) =$$
$$\Box_{b:Agent}\, env.a.(Env0, b) \rightarrow$$
$$send.a.b.(Msg1, Encrypt.(PK(b), Encrypt.(SK(a), k))) \rightarrow$$
$$\Box_{s:Secret}\, receive.b.a.(Msg2, Encrypt.(k, s)) \rightarrow STOP.$$

The definition of the responder is similar: the process $Responder(b, s)$ represents agent b running the protocol as responder using secret s. The responder starts by receiving a message 1, from an arbitrary agent a and containing an arbitrary session key k. It then sends back the corresponding message 2.

$$\begin{aligned}&Responder(b, s) = \\ &\quad \Box_{a:Agent, k:SessionKey} \\ &\qquad receive.a.b.(Msg1, Encrypt.(PK(b), Encrypt.(SK(a), k))) \rightarrow \\ &\quad send.b.a.(Msg2, Encrypt.(k, s)) \rightarrow STOP.\end{aligned}$$

As noted above, we consider a small system, comprising Alice acting as initiator, using key Ka, and Bob acting as responder, using secret Sb. The two agents do not communicate directly: we arrange below for all communications to go via the intruder. We model this as an interleaving.

$$System_0 = Initiator(Alice, Ka) \;|||\; Responder(Bob, Sb).$$

Of course, it is straightforward to consider larger systems, with more agents, or with particular agents running the protocol multiple times, perhaps with different roles.

3.3. Modelling the intruder

We now describe how we can model the intruder. The main issue is modelling which messages the intruder is able to understand and to create. We need to keep track, therefore, of which submessages of protocol messages the intruder knows; we term these *facts*:

$$\begin{aligned}Fact = \{&Encrypt.(PK(b), Encrypt.(SK(a), k)) \mid \\ &k \in SessionKey, a \in Agent, b \in Agent\} \cup \\ \{&Encrypt.(k, s) \mid k \in SessionKey, s \in Secret\} \cup \\ \{&Encrypt.(SK(a), k) \mid k \in SessionKey, a \in Agent\} \cup \\ &Agent \cup SessionKey \cup Secret \cup SecretKey \cup PublicKey.\end{aligned}$$

What we do is define a deduction system to capture the intruder's capabilities: we write $X \vdash f$ if, given set of facts X, he is able to create the fact f. The relation $\vdash$ is defined by the following four rules:

$$\begin{aligned}\{f, k\} &\vdash Encrypt.(k, f), && \text{for } k \in AllKeys, \\ \{Encrypt.(k, f), inverse(k)\} &\vdash f, && \text{for } k \in AllKeys, \\ \{f_1, \ldots, f_n\} &\vdash Sq.\langle f_1, \ldots, f_n\rangle, && \\ \{Sq.\langle f_1, \ldots, f_n\rangle\} &\vdash f_i, && \text{for } i = 1, \ldots, n.\end{aligned}$$

If the intruder knows a fact f and a key k then he can encrypt f with k; if he knows an encrypted message and the corresponding decryption key, he can perform the decryption to obtain the body; if he knows a collection of facts, he can concatenate them together; if he knows a concatenation, he can split it up into the individual components.

We will create a definition for the intruder that can hear messages sent across the network, deduce new facts from what he knows (as defined by the $\vdash$ relation), and then send facts he knows to other agents (possibly using identities other than his own).

We define $MsgBody$ to be the bodies of messages (i.e., without the label), and declare two channels on which the intruder can hear and say message bodies; later we will arrange for these events to synchronise with $send$ and $receive$ events of honest agents. We also declare a channel to capture inferences, and a channel on which the intruder can signal that he knows a particular secret.

$$
\begin{array}{l}
MsgBody = \{m \mid \exists\, l \bullet (l, m) \in Msg\},\\
\text{channel}\, hear, say : MsgBody,\\
\text{channel}\, infer : \{(f, X) \mid X \vdash f\},\\
\text{channel}\, leak : Secret.
\end{array}
$$

One way to define the intruder is as follows; the parameter S represents the intruder's current knowledge.

$$
\begin{array}{l}
Intruder_0(S) =\\
\quad hear?f : MsgBody \to Intruder_0(S \cup \{f\})\\
\quad \Box\\
\quad say?f : S \cap MsgBody \to Intruder_0(S)\\
\quad \Box\\
\quad leak?f : S \cap Secret \to Intruder_0(S)\\
\quad \Box\\
\quad \Box_{f:Fact,\, X \subseteq S,\, X \vdash f,\, f \notin S}\, infer.(f, X) \to Intruder_0(S \cup \{f\}).
\end{array}
$$

The intruder can: hear a message sent on the network, and add it to his knowledge; say a message that he knows; signal that he knows a particular secret; or infer a new fact f if for some subset X of his current knowledge, $X \vdash f$, and then add f to his knowledge.

Below we will instantiate S with a set IIK that represents the intruder's initial knowledge; we assume that the intruder knows all the agent's identities, all the public keys, his own secret key, a session key, and a secret:

$$IIK = Agent \cup \{PK(a) \mid a \in Agent\} \cup \{SK(Mallory), Km, Sm\}.$$

The definition of the intruder ensures[1]

$$
\begin{array}{l}
\forall\, tr' \frown \langle say.f \rangle \in traces(Intruder_0(IIK)) \bullet\\
\quad \exists\, S \subseteq IIK \cup \{f' \mid hear.f' \text{ in } tr'\} \bullet S \vdash f.
\end{array}
$$

The intruder can say a message only if he can deduce it from his initial knowledge and what he has heard.

The above design of the intruder process works in theory, but is very inefficient in practice, because of the way FDR works. FDR produces explicit state machines for sequential processes, such as $Intruder_0(IIK)$ above. If there are N facts that the intruder might learn, then the $Intruder_0$ process has 2^N states. It is typical for N to be of the order of several thousand; clearly it is infeasible to construct the intruder process explicitly in such cases.

Observe, though, that most states of the intruder will not be reachable in the context of a particular system — otherwise model checking would be infeasible. What we want

[1] "$\frown$" denotes concatenation of traces.

to do is avoid constructing the entire intruder process, but simply to explore the parts that are reachable in the context of the given system.

The way we do this is to construct the intruder in a different way. Rather than build the intruder as a single sequential process with 2^N states, we build it out of N component processes, one for each fact f, each with two states corresponding to whether the intruder does or does not know f.

$$
\begin{aligned}
&Ignorant(f) = \\
&\quad f \in MsgBody \;\&\; hear.f \rightarrow Knows(f) \\
&\quad \Box \\
&\quad \Box_{X \subseteq Fact,\, X \vdash f} \; infer.(f, X) \rightarrow Knows(f), \\
&Knows(f) = \\
&\quad f \in MsgBody \;\&\; hear.f \rightarrow Knows(f) \\
&\quad \Box \\
&\quad f \in MsgBody \;\&\; say.f \rightarrow Knows(f) \\
&\quad \Box \\
&\quad f \in Secret \;\&\; leak.f \rightarrow Knows(f) \\
&\quad \Box \\
&\quad \Box_{f' : Fact,\, X \subseteq Fact,\, f \in X,\, X \vdash f'} \; infer.(f', X) \rightarrow Knows(f).
\end{aligned}
$$

If the intruder doesn't know f, he can hear f on the network, or deduce it from some set X (that he does know), at which point he will know f. When he knows f he can: hear it on the network again; say it; signal that he knows it (if it is a secret); or use it within the inference of another fact.

We build the intruder by composing the component processes together in parallel, where the component for fact f synchronises on the events in its alphabet $alpha(f)$.

$$
\begin{aligned}
Intruder_0 &= \Big\|_{f : Fact} \, [alpha(f)] \text{ if } f \in IIK \text{ then } Knows(f) \text{ else } Ignorant(f), \\
alpha(f) &= \{hear.f, say.f, leak.f\} \cup \\
&\qquad \{infer.(f, X) \mid X \subseteq Fact,\; X \vdash f\} \cup \\
&\qquad \{infer.(f', X) \mid f' \in Fact,\; X \subseteq Fact,\; f \in X,\; X \vdash f'\}.
\end{aligned}
$$

Note in particular that an inference of the form $infer.(f, X)$ can occur precisely when the component for fact f is in the $Ignorant$ state, and for each $f' \in X$, the component for f' is in the $Knows$ state. The above highly parallel process is, then, equivalent to the original sequential definition, but FDR can construct it much more quickly.

We can make the definition more efficient, still, in order to reduce the size of the state space explored by FDR. When the intruder learns one fact, he can often use it to deduce many more facts; thus deductions often come together. The more deductions that are made, the more the intruder can do, so it makes sense to force the intruder to make all possible inferences, rather than allowing FDR to explore states where he makes a subset of those deductions. Further, if the intruder can make k independent such new deductions, then there are $k!$ orders in which those deductions can be made. But all different orders of the deductions reach the same state, so it doesn't matter what order they are made in: we should force FDR to consider just a *single* order of those deductions.

The FDR function $chase$ will do both of these for us: it forces internal τ events to occur, but picking an arbitrary order, and performing as many as possible. We therefore hide the inferences and apply $chase$, to force the intruder to perform as many inferences as possible, but in an arbitrary order.

$$Intruder_1 = chase(Intruder_0 \setminus \{|infer|\}).$$

Finally, we rename the $hear$ and say events within the intruder to $send$ and $receive$ events, ready to synchronise with the honest agents.

$$\begin{aligned} Intruder = Intruder_1 \\ &[\![send.a.b.(l,m)/hear.m \mid \\ &\quad a \in Agent - \{Mallory\}, b \in Agent, (l,m) \in Msg]\!] \\ &[\![receive.a.b.(l,m)/say.m \mid \\ &\quad a \in Agent, b \in Agent - \{Mallory\}, (l,m) \in Msg]\!] \end{aligned}$$

3.4. Putting it together

Finally, we construct our model of the complete system by composing the honest agents and the intruder together in parallel, synchronising on all the $send$ and $receive$ events to reflect the fact that the intruder has complete control over the network:

$$System = System_0 \underset{\{|send, receive|\}}{\parallel} Intruder.$$

4. Testing for security properties

We now describe how to test for various security properties: first secrecy properties, then authentication properties.

4.1. Secrecy properties

There are two properties concerning the secret s that we want to test for:

1. If an initiator a receives a value s, apparently from b, and b is not the intruder, then the intruder does not know s;
2. If a responder b sends a value s, intended for a, and a is not the intruder, then the intruder does not know s.

We can test for both of these together.

What we do is introduce new events of the form $claimSecret.a.s.b$ to indicate that a thinks that s is a secret that only b should know, and similar events with a and b swapped. We transform the system so that these events occur when a and b receive or send (respectively) a message 2, corresponding to items 1 and 2 above. We then hide all events other than the $claimSecret$ and $leak$ events.

$$\begin{aligned} SecretSystem = \\ &System [\![claimSecret.a.s.b/receive.b.a.(Msg2, Encrypt.(k,s)), \\ &\quad claimSecret.b.s.a/send.b.a.(Msg2, Encrypt.(k,s)) \mid \\ &\quad\quad a \in Agent, b \in Agent, s \in Secret, k \in SessionKey]\!] \\ &\setminus (\Sigma - \{|claimSecret, leak|\}). \end{aligned}$$

We want to test that, whenever an honest agent a performs a $claimSecret.a.s.b$ event, with b not the intruder, the intruder does not subsequently perform $leak.s$ indicating that he has learnt s. The process $SecretSpec$ below allows precisely those traces that satisfy this property; the parameter of the subsidiary process $SecretSpec'$ records those secrets for which there has been a corresponding $claimSecret$, and so must not be leaked by the intruder.

$$
\begin{array}{l}
SecretSpec = SecretSpec'(\{\}), \\
SecretSpec'(secs) = \\
\quad claimSecret?a?s?b \rightarrow \\
\quad\quad \text{if } b \in Honest \text{ then } SecretSpec'(secs \cup \{s\}) \text{ else } SecretSpec'(secs) \\
\quad \Box \\
\quad leak?s : Secret - secs \rightarrow SecretSpec'(secs).
\end{array}
$$

We can then test our secrecy properties by checking that every trace of $SecretSystem$ is also a trace of $SecretSpec$; we can do this by using FDR to test the refinement

$$SecretSpec \sqsubseteq_T SecretSystem.$$

When we perform this check, FDR finds that the refinement does not hold, and gives a witness trace of $SecretSystem$ that is not a trace of $SecretSpec$; this correspond to an attack against item 2 above. It turns out, though, that there is no attack against item 1.

The witness trace returned by FDR is

$$\langle claimSecret.Bob.Sb.Alice,\ leak.Sb \rangle.$$

Bob believes that Sb is a secret that only Alice should learn, yet the intruder does learn it. The FDR debugger can be used to find that the corresponding trace of $System$ is:

$$
\begin{array}{l}
\langle\ env.Alice.(Env0, Mallory), \\
\ send.Alice.Mallory. \\
\quad\quad (Msg1, Encrypt.(PK\_.Mallory, Encrypt.(SK\_.Alice, Ka))), \\
\ receive.Alice.Bob.(Msg1, Encrypt.(PK\_.Bob, Encrypt.(SK\_.Alice, Ka))), \\
\ send.Bob.Alice.(Msg2, Encrypt.(Ka, Sb)), \\
\ leak.Sb\ \rangle.
\end{array}
$$

This can be described in more standard notation as follows (the notation I_{Alice} represents the intruder faking a message, pretending to be Alice, or intercepting a message intended for Alice):

$$
\begin{array}{lrcl}
\text{Message } 0. & & \rightarrow Alice & : Mallory \\
\text{Message } 1. & Alice & \rightarrow Mallory & : \{\!|[Ka]_{SK(Alice)}|\!\}^{\mathsf{a}}_{PK(Mallory)} \\
\text{Message } 1. & I_{Alice} & \rightarrow Bob & : \{\!|[Ka]_{SK(Alice)}|\!\}^{\mathsf{a}}_{PK(Bob)} \\
\text{Message } 2. & Bob & \rightarrow I_{Alice} & : \{\!|Sb|\!\}^{\mathsf{s}}_{Ka}
\end{array}
$$

The intruder knows Sb.

Alice runs the protocol with Mallory, sending him a key Ka, signed with her public key. But Mallory can then use this signed key to send a fake message to Bob, pretending to be Alice. Bob accepts this key as having come from Alice, and so uses it to try to send Alice the secret Sb. However, the intruder knows Ka so can learn Sb.

4.2. Authentication

We now consider authentication of the responder to the initiator, and vice versa. More precisely, we consider the following questions:

1. If an initiator a completes a run of the protocol, apparently with b, then has b been running the protocol, apparently with a, and do they agree upon the value of the secret s and the session key k?
2. If a responder b completes a run of the protocol, apparently with a, then has a been running the protocol, apparently with b, and do they agree upon the value of the session key k? (Note that b can receive no guarantee that he and a agree upon s, because he cannot be sure that a even receives message 2.)

We describe how to test for the latter property; the test for the former is very similar.

We introduce new events, as follows:

- The event $Running.InitiatorRole.a.b.k$ indicates that a thinks that she is running the protocol as initiator, apparently with b, using session key k;
- The event $Complete.ResponderRole.b.a.k$ indicates that b thinks he has completed a run of the protocol as responder, apparently with a, using session key k.

We will then check that whenever the latter event occurs, the former event has previously occurred.

We arrange for initiator a to perform the $Running$ event when she sends message 1, and we arrange for responder b to perform the $Complete$ event when he sends message 2; we hide all other events.

$$
\begin{array}{l}
AuthSystem_0 = \\
\quad System[\![Running.InitiatorRole.a.b.k/ \\
\qquad\qquad send.a.b.(Msg1, Encrypt.(PK(b), Encrypt.(SK(a), k))), \\
\qquad\quad Complete.ResponderRole.b.a.k/ \\
\qquad\qquad send.b.a.(Msg2, Encrypt.(k, s)) \mid \\
\qquad\qquad\quad a \in Agent, b \in Agent, k \in SessionKey, s \in Secret]\!] \\
\qquad \setminus (\Sigma - alphaAuthSystem),
\end{array}
$$

$$
\begin{array}{l}
alphaAuthSystem = \\
\quad \{\!|Running.InitiatorRole.a.b, Complete.ResponderRole.b.a \mid \\
\qquad a \in Honest, b \in Honest|\!\}.
\end{array}
$$

(More generally, the $Complete$ event is performed at the last step in the protocol taken by that agent, and the $Running$ event is performed when the agent sends a message that should be causally linked to the other agent receiving a message.)

Recall that we want to check that whenever a responder b performs a $Complete$ event concerning initiator a, then a has previously performed a corresponding $Running$ event concerning b. We therefore consider the following specification process, which allows only such traces

$$
\begin{array}{l}
AuthSpec = Running.InitiatorRole?a.b.k \rightarrow \\
\qquad\qquad Chaos(\{Complete.ResponderRole.b.a.k\}).
\end{array}
$$

Note that this specification allows b to perform an arbitrary number of $Complete$ events corresponding to a single $Running$ event, and so does not insist that there is a one-one relationship between the runs of a and the runs of b. We could test for such a relationship by replacing the $Chaos(\{Complete.ResponderRole.b.a.k\})$ by $Complete.ResponderRole.b.a.k \rightarrow STOP$.

We can use FDR to test the refinement

$$AuthSpec \sqsubseteq_T AuthSystem.$$

(The above refinement test is appropriate since $AuthSystem$ performs at most a ***single*** $Running$ event; for a system that could perform n such events, we would replace the left hand side of the refinement test by an interleaving of n copies of $AuthSpec$.) FDR finds that this refinement does not hold, and returns the following witness trace:

$$\langle Complete.ResponderRole.Bob.Alice.Ka \rangle.$$

Bob thinks he has completed a run of the protocol with Alice, but Alice did not think that she was running the protocol with Bob. We can again use the FDR debugger to find the corresponding trace of $System$:

$$\begin{array}{l}\langle\ env.Alice.(Env0, Mallory) \\ \quad send.Alice.Mallory. \\ \qquad (Msg1, Encrypt.(PK\_.Mallory, Encrypt.(SK\_.Alice, Ka))) \\ \quad receive.Alice.Bob.(Msg1, Encrypt.(PK\_.Bob, Encrypt.(SK\_.Alice, Ka))) \\ \quad send.Bob.Alice.(Msg2, Encrypt.(Ka1, Sb))\ \rangle.\end{array}$$

This is the same as the attack against secrecy.

We can test whether the responder is authenticated to the initiator (item 1 above) in a similar way. FDR finds no attack in this case.

It is interesting to consider what guarantees the responder does receive from the protocol. We claim that if responder b completes a run of the protocol, apparently with a, then a has been running the protocol, and that they agree upon the value of the session key k. Note though that a might have been running the protocol with some agent c other than b, and so performed a $Running.InitiatorRole.Alice.c.k$ event. We can test this condition using the refinement check

$$AlivenessSpec \sqsubseteq_T SystemAliveness,$$

where

$$\begin{array}{l}AlivenessSpec = Running.InitiatorRole.Alice?c?k \rightarrow \\ \qquad\qquad Chaos(\{Complete.ResponderRole.b.Alice.k \mid b \in Agent\}),\end{array}$$

$$SystemAliveness = AuthSystem_0 \setminus (\Sigma - alphaSystemAliveness),$$

$$\begin{array}{l}alphaSystemAliveness = \\ \quad \{Running.InitiatorRole.a.b, Complete.ResponderRole.b.a \mid \\ \qquad a \in Honest,\ b \in Agent\}.\end{array}$$

4.3. Correcting the protocol

It is fairly straightforward to correct the protocol to prevent the attacks identified above. Each attack was caused by Bob accepting a key signed by Alice that had been intended for Mallory rather than himself. The obvious way to prevent this attack, then, is to include the intended recipient's identity within this signed message. We also include the sender's identity, although this is not actually necessary:

$$\text{Message } 1. \quad a \rightarrow b \ : \ \{\!|[a, b, k]_{SK(a)}|\!\}^{\mathsf{a}}_{PK(b)}$$
$$\text{Message } 2. \quad b \rightarrow a \ : \ \{\!|s|\!\}^{\mathsf{s}}_{k}$$

It is straightforward to adapt the CSP model to reflect this change in the protocol: it simply requires a change to the definitions of the honest agents, the sets Msg and $Fact$, and the renamings performed to the final system, all restricted to the parts that deal with message 1.

When we analyse this version of the protocol using FDR, we find no attacks against the main security properties discussed above. If we use the version of the authentication check that tests whether there is a one-one relationship between the runs of a and the runs of b, we find that this does not hold: clearly the intruder can replay a message 1 sent by a multiple times so that b thinks he's performed multiple runs corresponding to a single run of a; we consider this a limitation of the protocol rather than a serious flaw.

We should be clear, though, about the limitations of this analysis. When we model check a particular system running the protocol and find no attacks, that does not necessarily imply that there is no attack upon some other system running the protocol (although in practice most attacks can be found by considering fairly small systems). We address this problem in the following section.

5. Model checking arbitrary systems

In this section we show how to build a model that abstracts an *arbitrary* system running the protocol: if no attack is found on this model, then no attack exists on the protocol.

For ease of exposition, we consider only the question of attacks upon the responder. By symmetry, it is enough to consider attacks upon a *particular* instance of a *particular* responder, say the instance $Responder(Bob, Sb)$, which we call the *principal responder*. We want to consider all possible systems containing this principal responder, together with *arbitrary* other instances of honest agents (possibly including other instances of Bob, as either initiator or responder) and the intruder. Each such system can be written in the form

$$System_0 = (Responder(Bob, Sb) \ ||| \ Others) \underset{\{|send, receive|\}}{\|} Intruder,$$

where $Others$ models all the other honest instances. Of course, there are infinitely many such systems. However, we show how to construct a model that is guaranteed to find all attacks upon all such systems.

Our development proceeds in several steps. In the next subsection, we show how to build models that represent the effects of the other honest agents, $Others$, internally within the intruder. However, that still produces an infinite number of models, param-

eterised by the types used. In Subsection 5.2 we show how to reduce those models to a single finite model, using techniques from the area of data independence. Finally, in Subsection 5.3 we show how we can analyse the resulting model, and, if we find no attacks, deduce that there are no attacks upon an arbitrary system of the form of $System_0$, above.

5.1. Internalising agents

Recall that we are interested only in attacks against the principal responder. In so far as such attacks are concerned, there are two contributions that the other instances $Others$ can make:

1. An initiator instance within $Others$ can send a message 1, which the intruder can overhear and hence use;
2. A responder instance within $Others$ can send a message 2, which the intruder can overhear and hence use; note that this will happen only after that responder has received a corresponding message 1, which the intruder will know.

We build a new intruder process, $Intruder'$, that corresponds to the combination of $Intruder$ and $Others$, except it represents messages sent by the other honest agents in a different way. We represent those messages as deductions within the new intruder, rather than as explicit messages: in other words, these interactions are modelled internally to the intruder, rather than externally.

Consider, first, item 1 above, corresponding to an initiator instance, a say, sending a message 1. Our new model will represent this by a deduction of the corresponding message from the empty set. For later convenience, we partition the set $SessionKey$ into three subtypes: those keys that an honest initiator intends to share with the intruder, denoted $SessionKeyKnown$; those keys that an honest initiator intends to share with an honest agent, denoted $SessionKeyUnknown$; and those keys that the intruder knows initially, denoted $SessionKeyIntruder$; in the first two cases, the suffixes $Known$ and $Unknown$ indicate whether or not we expect the value to be known by the intruder. The intruder overhearing a message 1 intended for some honest agent b can then be represented by a deduction of the following form:

$$\{\} \vdash Encrypt.(PK(b), Encrypt.(SK(a), Sq.\langle a, b, k\rangle)), \quad \text{for } a, b \in Honest,\ k \in SessionKeyUnknown. \tag{1}$$

Similarly, the intruder receiving a message 1 intended for himself can be represented by a deduction of the following form:

$$\{\} \vdash Encrypt.(PK(Mallory), Encrypt.(SK(a), Sq.\langle a, Mallory, k\rangle)), \quad \text{for } a \in Honest,\ k \in SessionKeyKnown. \tag{2}$$

We now consider item 2 above, corresponding to the responder b sending a message 2 in response to receiving a message 1 that the intruder knows. We can represent this by a deduction of the message 2 from the message 1. We partition $Secret$ into four subtypes: the distinguished secret Sb used by the principal responder; those other secrets that an honest responder intends to share with the intruder, denoted $SecretKnown$; those other secrets that an honest responder intends to share with an honest agent,

denoted $SecretUnknown$; and those secrets that the intruder knows initially, denoted $SecretIntruder$. The intruder overhearing a message 2 intended for some honest agent a can then be represented by a deduction of the following form:

$$\{Encrypt.(PK(b), Encrypt.(SK(a), Sq.\langle a, b, k\rangle))\} \vdash Encrypt.(k, s), \quad \text{for } a, b \in Honest,\ s \in SecretUnknown,\ k \in SessionKey. \tag{3}$$

Similarly, the intruder receiving a message 2 intended for himself can be represented by a deduction of the following form:

$$\{Encrypt.(PK(b), Encrypt.(SK(Mallory), Sq.\langle Mallory, b, k\rangle))\} \vdash Encrypt.(k, s), \quad \text{for } b \in Honest,\ s \in SecretKnown,\ k \in SessionKey. \tag{4}$$

Note that in both cases we allow k to range over *all* of $SessionKey$: we do not want to make any assumptions about the set of keys for which the intruder can construct a valid message 1 that is accepted by an honest agent.

Let $Intruder'$ be constructed using this extended deduction relation, in the same way that $Intruder$ was constructed in Section 3.3, except with none of the $infer$ events corresponding to new deductions hidden. We can then construct a new system

$$System_{Int} = Responder(Bob, Sb) \underset{\{|send, receive|\}}{\|} Intruder'.$$

This is analogous to $System$, except that messages from the other agents have been replaced by corresponding $infer$ events.

In fact, $System_{Int}$ has slightly more behaviours than $System$. For example, it allows *multiple* deductions that produce messages containing the same fresh value: for instance, it would allow the intruder to perform *both* the deductions

$$\{\} \vdash Encrypt.(PK(Bob), Encrypt.(SK(Alice), Sq.\langle Alice, Bob, K1\rangle)),$$

$$\{\} \vdash Encrypt.(PK(Bill), Encrypt.(SK(Alison), Sq.\langle Alison, Bill, K1\rangle)),$$

using the same key $K1$, whereas within $System$ only a single message 1 containing $K1$ would be sent. Of course, these extra behaviours are safe: if we can verify $System_{Int}$ then we can deduce that $System$ is secure. There is a possibility of these extra behaviours leading to the discovery of *false attacks*: attacks against $System_{Int}$ where there is no corresponding attack on $System$. However, we have designed $System_{Int}$ so that these false attacks do not arise in practice: this is one reason why we partitioned $SessionKey$ and $Secret$ into those values intended for use with the intruder or with honest agents.

Note that we have, in fact, produced an *infinite family* of systems, one for each choice of the types $Agent$, $SessionKey$ and $Secret$ (and their subtypes); we sometimes write $System_{Int}(Agent, SessionKey, Secret)$ to make this dependence explicit. Note in particular that the intruder is also parameterised by these types, both in the definition of the set of deductions, and in the definition of his initial knowledge IIK as defined in Section 3.3. We cannot directly check all members of this family. In the next section we show how to perform a further reduction, in order to reduce the analysis of *all* systems of the form $System_{Int}(Agent, SessionKey, Secret)$ to the analysis of a *single* system $System_{Int}(Agent^{\dagger}, SessionKey^{\dagger}, Secret^{\dagger})$, for some fixed types $Agent^{\dagger}$, $SessionKey^{\dagger}$ and $Secret^{\dagger}$.

5.2. Reducing the system

A process $P(T)$ is said to be *data independent* in its type parameter T if it can input and output values from T, store them for later use, perform polymorphic operations such as tupling upon them, but not perform any computations that constrain what T may be. The processes we have defined to model the protocol have been data independent in the types $Agent$, $SessionKey$, and $Secret$. A number of interesting results have been proven that allow one to deduce results about a data independent process $P(T)$ for *all* choices of T from an analysis of $P(T^\dagger)$ for some fixed type $T^\dagger$; see, for example [Ros97, Chapter 15]. We apply some of the theory of data independence here.

Without loss of generality, we may assume that each type $Agent$, $SessionKey$ and $Secret$ contains at least as many elements as the corresponding types $Agent^\dagger$, $SessionKey^\dagger$ and $Secret^\dagger$, since increasing the sizes of the types does not remove any traces from $System_{Int}(Agent, SessionKey, Secret)$.

In order to reduce the infinite family of systems of the form of $System_{Int}$, we will define a function $\phi : Fact \to Fact$, and (informally speaking) consider a system that uses fact $\phi(f)$ whenever $System_{Int}$ uses f. Given a particular choice of $Agent$, $SessionKey$ and $Secret$, we define ϕ as the homomorphism induced by three surjective functions on atomic types:

$$\phi_{Ag} : Agent \to Agent^\dagger,$$
$$\phi_{SK} : SessionKey \to SessionKey^\dagger,$$
$$\phi_{Sec} : Secret \to Secret^\dagger.$$

So, for example,

$$\phi(Encrypt.(SK(a),\ Sq.\langle a, b, k\rangle)) = \\ (Encrypt.(SK(\phi_{Ag}(a)),\ Sq.\langle \phi_{Ag}(a), \phi_{Ag}(b), \phi_{SK}(k)\rangle)).$$

We lift ϕ to events in the obvious way, for example $\phi(send.a.b.(l, m)) = send.\phi_{Ag}(a).\phi_{Ag}(b).(l, \phi(m))$. We lift ϕ to sets, traces, etc., by point-wise application.

The system we will consider, then, is

$$System^\dagger = System_{Int}(\phi_{Ag}(Agent),\ \phi_{SK}(SessionKey), \phi_{Sec}(Secret)) \\ = System_{Int}(Agent^\dagger,\ SessionKey^\dagger, Secret^\dagger).$$

We give the definitions of ϕ_{Ag}, etc., below (in fact, we'll use different definitions depending on the property we're checking). However, they will each map values onto small fixed ranges, $Agent^\dagger$, etc., so that the above process $System^\dagger$ is independent of the choices of $Agent$, $SessionKey$ and $Secret$. In other words, we will reduce arbitrary systems to some fixed finite system.

In order to deduce results about $System_{Int}$ from an analysis of $System^\dagger$, we would like to be able to relate their traces. The property we would like to deduce is

$$traces(System^\dagger) \supseteq \\ \{\phi(tr) \mid tr \in traces(System_{Int}(Agent, SessionKey, Secret))\}. \tag{5}$$

If we can do that, and prove some property of the traces of $System^\dagger$ (by performing a refinement check using FDR) we will be able to deduce a corresponding property about the traces of $System_{Int}$. Equation (5) does hold, but this requires some justification; indeed the generalisation to an arbitrary CSP process P parameterised by type T,

$$traces(P(\phi(T)) \supseteq \{\phi(tr) \mid tr \in traces(P(T))\}, \tag{6}$$

does not hold. To see why not, consider the process (taken from [RB99]):

$$P(T) = in?x : T \rightarrow in?y : T \rightarrow \text{if } x = y \text{ then } a \rightarrow STOP \text{ else } b \rightarrow STOP.$$

Suppose $T = \{0, 1\}$ and $\phi(0) = \phi(1) = 0$; then the trace $\langle in.0, in.0, b\rangle$ is contained in $\{\phi(tr) \mid tr \in traces(T)\}$, but not in $traces(P(\phi(T)))$.

Lazić [Laz99] shows that the following condition, known as **PosConjEqT** (*positive conjunction of equality tests*), is enough to ensure that equation (6) does hold (in fact, the two sides of the equation are equal in this case):

> A process P satisfies **PosConjEqT** precisely when the failure of each equality check in P results in the process $STOP$.

For example, the following process satisfies **PosConjEqT**:

$$P(T) = in?x : T \rightarrow in?y : T \rightarrow \text{if } x = y \text{ then } a \rightarrow STOP \text{ else } STOP.$$

The CSP processes we have defined to model the honest protocol participants satisfy **PosConjEqT**. However, this won't be the case in protocols where an agent performs inequality tests —say between two values she receives, or between the identity of her apparent partner and her own identity— before proceeding.

The process we have defined to model the intruder also satisfies equation (6). In [RB99], it is shown that this will be the case provided the underlying deduction relation $\vdash$ is *positive*, i.e., whenever $X \vdash f$, it is also the case that $\phi(X) \vdash \phi(f)$; in other words, if the intruder can make a particular deduction before the types are collapsed, he can make the corresponding deduction after the types are collapsed. This is true of the deduction system we are using here.[2]

We have shown that both components of $System^\dagger$ satisfy equation (6). One can show that this property is preserved by parallel composition. Hence the entire protocol model $System^\dagger$ satisfies equation (5).

5.3. Testing for secrecy and authentication

We now describe how to adapt the refinement test for secrecy from Section 4.1. Recall that we are only considering secrecy from the point of view of the principal responder.

We define the reduction functions ϕ_{Ag}, etc., as follows, to reduce each of the subtypes to singleton values:

[2] As an example of a deduction system that is not positive, consider the following, designed to capture the cracking of a one-time pad that has been used with two distinct texts ($\oplus$ denotes bit-wise exclusive-or):

$$\{k \oplus t_1, k \oplus t_2\} \vdash k, \qquad \text{for } k \in Key,\ t_1, t_2 \in Text,\ t_1 \neq t_2.$$

If $\phi(t_1) = \phi(t_2)$, for some $t_1 \neq t_2$, then the deduction system is not positive.

$$
\begin{aligned}
\phi_{Ag}(Bob) &= Bob,\\
\phi_{Ag}(Mallory) &= Mallory,\\
\phi_{Ag}(a) &= Alice, \quad \text{for } a \in Honest - \{Bob\},\\
\\
\phi_{Sec}(Sb) &= Sb,\\
\phi_{Sec}(s) &= Sm, \quad \text{for } s \in SecretKnown \cup SecretIntruder,\\
\phi_{Sec}(s) &= Sa, \quad \text{for } s \in SecretUnknown,\\
\\
\phi_{SK}(k) &= Km, \quad \text{for } k \in SessionKeyKnown \cup\\
&\qquad\qquad\qquad SessionKeyIntruder,\\
\phi_{SK}(k) &= Ka, \quad \text{for } k \in SessionKeyUnknown.
\end{aligned}
$$

This definition means that $Agent^\dagger = \{Alice, Bob, Mallory\}$, $Secret^\dagger = \{Sa, Sb, Sm\}$, and $SessionKey^\dagger = \{Ka, Km\}$. For convenience, write $Honest^\dagger$ for $\{Alice, Bob\}$.

In particular, the effect of this reduction is to reduce the intruder's initial knowledge in the new system to

$$
\begin{aligned}
&\{Alice, Bob, Mallory, Sm, Km, SK(Mallory),\\
&\quad PK(Alice), PK(Bob), PK(Mallory)\}.
\end{aligned}
$$

Further, it reduces the deductions corresponding to internalised agents (equations (1–4)) to the following, where a and b range over $Honest^\dagger$, and k ranges over $SessionKey^\dagger$:

$$\{\} \vdash Encrypt.(PK(b), Encrypt.(SK(a), Sq.\langle a, b, Ka\rangle)),$$

$$\{\} \vdash Encrypt.(PK(Mallory), Encrypt.(SK(a), Sq.\langle a, Mallory, Km\rangle)),$$

$$\{Encrypt.(PK(b), Encrypt.(SK(a), Sq.\langle a, b, k\rangle))\} \vdash Encrypt.(k, Sa),$$

$$
\begin{aligned}
&\{Encrypt.(PK(b), Encrypt.(SK(Mallory), Sq.\langle Mallory, b, k\rangle))\}\\
&\quad \vdash Encrypt.(k, Sm).
\end{aligned}
$$

In order to test whether the secrecy property is satisfied, we perform a renaming and hiding to $System^\dagger$, analogous to that which produced $SecretSystem$ (but with the renaming restricted to just the principal responder):

$$
\begin{aligned}
&SecretSystem^\dagger =\\
&\quad System^\dagger \llbracket claimSecret.Bob.Sb.a / send.Bob.a.(Msg2, Encrypt.(k, Sb)) \mid\\
&\qquad\qquad a \in Agent^\dagger, k \in SessionKey^\dagger \rrbracket\\
&\qquad \setminus (\Sigma - \{\!| claimSecret, leak.Sb |\!\}).
\end{aligned}
$$

We consider a specification analogous to $SecretSpec$, but restricted to just the principal responder and the principal responder's nonce, Sb:

$$
\begin{aligned}
SecretSpec^\dagger = {} & claimSecret.Bob.Sb?a \rightarrow\\
& \quad \text{if } a \in Honest^\dagger \text{ then } SecretSpec1^\dagger \text{ else } SecretSpec^\dagger\\
& \Box\\
& leak.Sb \rightarrow SecretSpec^\dagger,
\end{aligned}
$$

$$SecretSpec1^\dagger = claimSecret.Bob.Sb?a \rightarrow SecretSpec1^\dagger.$$

We can then use FDR to verify that

$$SecretSpec^{\dagger} \sqsubseteq_T SecretSystem^{\dagger}.$$

We now deduce a corresponding result for $System_{Int}$. The success of the above refinement test means that $SecretSystem^{\dagger}$ has no trace of the form

$$\langle \ldots, claimSecret.Bob.Sb.a, \ldots, leak.Sb \rangle,$$

for $a \in Honest^{\dagger}$. Hence $System^{\dagger}$ has no trace of the form

$$\langle \ldots, send.Bob.a.(Msg2, Encrypt.(k, Sb)), \ldots, leak.Sb \rangle,$$

for $k \in SessionKey^{\dagger}$. Therefore, by equation (5), and using the definition of ϕ_{Ag} etc., $System_{Int}(Agent, SessionKey, Secret)$ has no trace of the form

$$\langle \ldots, send.Bob.a.(Msg2, Encrypt.(k, Sb)), \ldots, leak.Sb \rangle,$$

for $a \in Honest$ and $k \in SessionKey$. We deduce that there is no secrecy attack against $System_{Int}(Agent, SessionKey, Secret)$, for all choices of the types.

Thus we have verified that, for an arbitrary system running the protocol, there is no secrecy attack against the principal responder, and hence, by symmetry, there is no secrecy attack against an arbitrary responder.

We now consider how to adapt the test for authentication from Section 4.2. Recall that we are considering only authentication attacks on authentication against the principal responder, and testing for agreement on the session key.

We let $Agent^{\dagger} = \{Alice, Bob, Mallory\}$, $SessionKey^{\dagger} = \{Ka, Kb, Km\}$ and $Secret^{\dagger} = \{Sa, Sb, Sm\}$. We let ϕ_{Sec} be as in the previous section; we define ϕ_{Ag} and ϕ_{SK} below.

In order to test for authentication, we produce a system similar to $AuthSystem$ from Section 4.2. However, we are only interested in checking for authentication to the principal responder Bob, so we need produce only the corresponding $Running$ and $Complete$ signals. Recall that in Section 4.2 the $Running$ event was produced by renaming the sending of a message 1. However, in $System^{\dagger}$ that send is modelled by an internalised inference, of the form

$$infer.(Encrypt.(PK(Bob), Encrypt.(SK(a), Sq.\langle a, Bob, k \rangle)), \{\}).$$

For ease of reference, we write $Msg1Inf(a, k)$ for the above event. We therefore need to produce the $Running$ event by renaming the corresponding $Msg1Inf(a, k)$ event:[3]

[3]In the interests of efficiency, we should hide and chase all $infer$ events other than $Msg1Inf(a, k)$ events when we construct the intruder process.

$$
\begin{array}{l}
AuthSystem^{\dagger} = \\
\quad System^{\dagger} \\
\quad [\![Running.InitiatorRole.a.Bob.k / Msg1Inf(a,k), \\
\quad\ Complete.ResponderRole.Bob.a.k / send.Bob.a.(Msg2, Encrypt.(k,s)) \mid \\
\quad\quad a \in Agent, k \in SessionKey, s \in Secret]\!] \\
\quad \setminus (\Sigma - alphaAuthSystem^{\dagger}),
\end{array}
$$

$$
\begin{array}{l}
alphaAuthSystem^{\dagger} = \\
\quad \{\!| Running.InitiatorRole.a.Bob, Complete.ResponderRole.Bob.a \mid \\
\quad\quad a \in Honest |\!\}.
\end{array}
$$

We can also produce a specification process, similar to $AuthSpec$ from Section 4.2, but specialised to the principal responder:

$$
\begin{array}{l}
AuthSpec^{\dagger} = Running.InitiatorRole?a!Bob?k \rightarrow \\
\qquad\qquad Chaos(\{Complete.ResponderRole.Bob.a.k\}).
\end{array}
$$

We can then use FDR to verify that

$$AuthSpec^{\dagger} \sqsubseteq_T AuthSystem^{\dagger}.$$

We can use this to deduce a corresponding result for $System_{Int}$. The success of the above refinement test means that for every trace tr of $System^{\dagger}$, and for all $a \in Honest^{\dagger}$ and $k \in SessionKey^{\dagger}$

$$
\begin{array}{l}
\text{if } tr = tr' \frown \langle send.Bob.a.(Msg2, Encrypt.(k,s)) \rangle \\
\text{then } Msg1Inf(a,k) \text{ in } tr'.
\end{array}
\tag{7}
$$

We claim that a similar result holds for $System_{Int}(Agent, SessionKey, Secret)$. Suppose, for a contradiction, that $System_{Int}(Agent, SessionKey, Secret)$ does have a trace tr such that for some $A \in Honest$ and $K \in SessionKey$

$$
\begin{array}{l}
tr = tr' \frown \langle send.Bob.A.(Msg2, Encrypt.(K,s)) \rangle \\
\text{but not } Msg1Inf(A,K) \text{ in } tr'.
\end{array}
\tag{8}
$$

Consider the collapsing functions ϕ_{Ag} and ϕ_{SK} below:

$$
\begin{array}{rll}
\phi_{Ag}(A) = Alice, & & \\
\phi_{Ag}(Bob) = Bob, & & \\
\phi_{Ag}(c) = Mallory, & \text{for } c \neq A, Bob, & \\
\phi_{SK}(K) = Kb, & & \\
\phi_{SK}(k) = Km, & \text{for } k \in SessionKeyKnown \cup & \\
 & \qquad SessionKeyIntruder - \{K\}, & \\
\phi_{SK}(k) = Ka, & \text{for } k \in SessionKeyUnknown - \{K\}. &
\end{array}
$$

Then

$$\phi(tr) = \phi(tr') \frown \langle send.Bob.Alice.(Msg2, Encrypt.(Kb, \phi_{Sec}(s))) \rangle.$$

However, $\phi(tr')$ has no corresponding $Msg1Inf(Alice, Kb)$ event, by the way we have constructed ϕ. But equation (5) implies that $\phi(tr) \in traces(System^{\dagger})$, which contradicts equation (7). Thus we have obtained a contradiction to equation (8). Hence $System_{Int}(Agent, SessionKey, Secret)$ satisfies the authentication property, for all choices of the types. Thus we have verified that, for an arbitrary system running the protocol, there is no authentication attack against the principal responder, and hence, by symmetry, no authentication attack against an arbitrary responder.

6. Casper

In the previous sections, we have described the CSP models of security protocols. However, creating these models by hand is time-consuming and error-prone. Casper is a compiler that automates this process, creating the CSP models from a more abstract description written by the user. In this section we briefly outline the Casper input syntax; for more details see [LBDH].

A complete input file, corresponding to the model in Sections 3 and 4, is given in Figure 2. The script is split into eight sections, each beginning with a header starting with "#". The first three sections describe the protocol; the fourth section defines the specifications for it to be tested against; and the last four sections describe the system to be checked.

The `#Free variables` section declares the variables to be used in the protocol description, together with their types; it also defines which keys are inverses of one another. The `#Processes` section defines names for the agents running in the protocol, together with parameters for them; Casper uses these names for the corresponding CSP processes. This section also defines the initial knowledge of the agents; Casper uses this information to check that the protocol is feasible, in that each agent has all the information necessary to run the protocol. The `#Protocol description` section defines the protocol itself, using an ascii representation of standard protocol notation. In particular, it includes a message 0 that `a` receives from its environment, telling it with whom to run the protocol, and corresponding to the environment message in the CSP model.

The `#Specification` section defines the security properties to be tested. The two `Secret` specifications correspond to the two secrecy properties considered in Section 4.1: that `a` believes that `s` is a secret that only `b` should know, and vice versa. The two `Agreement` specifications correspond to the first two authentication properties considered in Section 4.2: that `b` is authenticated to `a` and that they agree upon the values of `s` and `k`; and that `a` is authenticated to `b` and that they agree upon the value of `k`. The `Aliveness` specification corresponds to the property considered at the end of Section 4.2: that if `b` completes a run of the protocol, apparently with `a`, then `a` has previously been running the protocol.

The `#Actual variables` section defines the atomic values to be used in the protocol model; this is used in the definition of the $Encryption$ datatype in the model. This section also defines which keys are inverses of one another. The `#Functions` section tells Casper to produce symbolic definitions for the functions `PK` and `SK`; alternatively, the user can provide his own definitions. The `#System` section defines the instances of honest agents in the system. Finally, the `#Intruder Information` section gives the name of the intruder, and his initial knowledge (corresponding to IIK in the CSP model).

```
#Free variables
a, b : Agent
k : SessionKey
s : Secret
PK : Agent -> PublicKey
SK : Agent -> SecretKey
InverseKeys = (PK, SK), (k, k)

#Processes
Initiator(a, k) knows PK, SK(a)
Responder(b, s) knows PK, SK(b)

#Protocol description
0.   -> a : b
1. a -> b : { {k}{SK(a)} }{PK(b)}
2. b -> a : {s}{k}

#Specification
Secret(a, s, [b])
Secret(b, s, [a])
Agreement(b, a, [s,k])
Agreement(a, b, [k])
Aliveness(a, b)

#Actual variables
Alice, Bob, Mallory : Agent
Ka, Km : SessionKey
Sb, Sm : Secret
InverseKeys = (Ka, Ka), (Km, Km)

#Functions
symbolic PK, SK

#System
Initiator(Alice, Ka)
Responder(Bob, Sb)

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

Figure 2. Casper script for the example protocol

Casper also provides support for analysing systems of unbounded size, as described in Section 5. Most sections of the Casper script are unchanged; Figure 3 gives those sections that are changed.

The `generates` clauses in the `#Processes` section indicates that the initiator and responder generate fresh values for `k` and `s`, respectively. The `#Intruder Information` section includes the directive `UnboundParallel = True` to indicate that unboundedly many instances of other agents should be modelled as internal agents.

```
#Processes
INITIATOR(a,k) knows PK, SK(a) generates k
RESPONDER(b,s) knows PK, SK(b) generates s

#Actual variables
Mallory : Agent

#System
GenerateSystem = True

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {PK, SK(Mallory)}
UnboundParallel = True
```

Figure 3. Changes to the Casper script to allow checking of arbitrary systems

Those types used in `generates` clauses (here `SessionKey` and `Secret`) and the types of agent identities (here `Agent`) are treated as being data independent.

The `#System` section includes the directive `GenerateSystem = True` to indicate that Casper should automatically generate a system containing just enough agents, generate types containing just enough values, and give the intruder just enough initial knowledge, in order for the security of this small system to imply the security of arbitrary systems, using an argument as in Section 5.3. The user has to declare only the intruder's identity, and to specify what initial knowledge from non-data-independent types the intruder should have. (It is also possible for the user to specify the system and types, as in a standard script.)

Analyses based on scripts of this form are less good for detecting attacks than those of the standard form, since counterexamples returned by FDR are harder to understand (many of the details of the attack are hidden within internalised deductions), but are appropriate for protocols that are believed to be correct.

7. Notes

In this section we give brief historical notes concerning the development of the techniques described in this chapter, and pointers to other extensions.

The first uses of model checkers to analyse security protocols are due to Millen et al. [MCF87] and Meadows [Mea96]. Roscoe was the first to consider the use of CSP and FDR in [Ros95], but the ideas gained more prominence in [Low96] and [LR97]. The highly-parallel model of the intruder from Section 3.3 was proposed in [RG97].

Casper was first described in [Low97a,Low98]; Casper itself is available from [Low09]. Different types of authentication requirements, and how to model them in CSP, were discussed in [Low97b]. The book [RSG$^+$00] includes a more detailed description of the CSP approach to modelling and analysis of security protocols.

A number of other extensions have been made to the approach. Some cryptographic systems have interesting algebraic properties; for example, Vernam encryption (bit-wise exclusive-or) is associative and commutative, and Diffie-Hellman encryption depends upon commutativity of exponentiation. In [RG97], Roscoe and Goldsmith describe how

to model such algebraic properties, by renaming each fact to a representative member of its equivalence class. Some protocols make use of passwords, which might be poorly-chosen and so appear in an on-line dictionary; some such protocols may be subject to *guessing attacks*, where the intruder is able to test in some way whether a particular guess of the password is correct, and to iterate this process through all words in the dictionary; [Low04] describes how to analyse protocols for such guessing attacks. In some protocols (such as the running example in this book!), if an old session key is compromised, the intruder can replay it in order to launch subsequent attacks; [Hui01] describes how to find such attacks.

The theory of data independence, as applied to CSP, was developed in [Laz99]. Roscoe first applied data independence to security protocols in [Ros98], using a technique to "recycle" fresh values in a protocol, to give a model that simulated a system with an unbounded number of sequential runs (but with a limited number of concurrent runs). Roscoe and Broadfoot extended these ideas in [RB99], and introduced the idea of internalising agents, although there it was applied to secondary agents such as servers, and was intended as a technique for reducing the size of the state space rather than to provide for a general proof. These ideas were extended in [BR02,Bro01] to show that internalising agents can be used to provide a general proof; however, in that development, considerable machinery was necessary to avoid false attacks. The models we have presented here are due to Kleiner [Kle08]; Kleiner also presents some extensions we have not discussed, and presents several theorems giving appropriate choices for the collapsed datatypes for checking different security properties.

Acknowledgments

I would like to thank Bill Roscoe for discussions about security protocols and their analysis using CSP, stretching over the last 15 years. I would also like to thank Eldar Kleiner for discussions about his techniques for internalising agents and applying data independence techniques. I received useful comments on this chapter from Tom Gibson-Robinson, Steve Kremer, Tomasz Mazur, Toby Murray and Bernard Sufrin.

References

[BR02] P. J. Broadfoot and A. W. Roscoe. Capturing parallel attacks within the data independence framework. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[Bro01] Philippa Broadfoot. *Data independence in the model checking of security protocols*. DPhil thesis, Oxford University, 2001.

[DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.

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

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

[Hui01] Mei Lin Hui. *A CSP Appraoch to the Analysis of Security Protocols*. PhD thesis, University of Leicester, 2001.

[Kle08] Eldar Kleiner. *A Web Services Security Study using* Casper *and FDR*. DPhil thesis, Oxford University, 2008.

[Laz99] Ranko Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. DPhil thesis, Oxford University, 1999.

[LBDH] Gavin Lowe, Philippa Broadfoot, Christopher Dilloway, and Mei Lin Hui. Casper*: A Compiler for the Analysis of Security Protocols, User Manual and Tutorial*. Oxford University Computing Laboratory. Available via `http://www.comlab.ox.ac.uk/people/gavin.lowe/Security/Casper/index.html%`.

[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.

[Low97a] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, pages 18–30, 1997.

[Low97b] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[Low04] Gavin Lowe. Analysing protocols subject to guessing attacks. *Journal of Computer Security*, 12(1):83–98, 2004.

[Low09] Gavin Lowe. Casper: A compiler for the analysis of security protocols, 1996–2009. World Wide Web home page at URL `http://www.comlab.ox.ac.uk/people/gavin.lowe/Security/Casper/index.html%`.

[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.

[MCF87] J. K. Millen, S. C. Clark, and S. B. Freedman. The interrogator: Protocol security analysis. *IEEE Transactions on software Engineering*, 13(2), 1987.

[Mea96] Catherine Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[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.

[RG97] 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`.

[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.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

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

[RSG+00] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Pearson Education, 2000.