On the Specification and Analysis of Secure Transport Layers

Christopher Dilloway

Keble College

University of Oxford Computing Laboratory

Trinity Term 2008

This thesis is submitted to the University of Oxford Computing Laboratory for the degree of Doctor of Philosophy.

Acknowledgements

I would like to thank my supervisor, Gavin Lowe, for his advice and support over the last three years. Many parts of the work described in this thesis have benefitted from discussions with him, and much of the presentation is clearer because of his suggestions.

I would also like to thank Bill Edwards, Jason Crampton, Michael Goldsmith, Joshua Guttman, Allaa Kamil, Eldar Kleiner, Toby Murray, Kenny Paterson, Bill Roscoe and my examiners Steve Schneider and Andy Simpson for many invaluable discussions about the work, and for their comments and questions that have helped me to refine my ideas. Thanks are also due to the anonymous referees from WITS'07, CSF'08 and FCS-ARSPA-WITS'08 for their useful comments on papers based on some of the work in this thesis.

I am grateful to Bill, Dan, Ozzy and James for helping to make the Comlab a friendly place to work, and to my friends for being able to distract me from my work, particularly when I most needed distracting!

I could not have completed this thesis without the support of my family and, in particular, none of this would have been possible without the love and support of my fiancée, Joanna.

This work was supported by a grant from the Office of Naval Research.

On the Specification and Analysis of Secure Transport Layers

Christopher Dilloway

Keble College

A thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford, Trinity Term 2008.

Abstract

The world is becoming strongly dependent on computers, and on distributed communication between computers. As a result of this, communication security is important, sometimes critically so, to many day-to-day activities. Finding strategies for discovering attacks against security protocols and for proving security protocols correct is an important area of research.

An increasingly popular technique that is used to simplify the design of security protocols is to rely on a secure transport layer to protect messages on the network, and to provide protection against attackers. In order to make the right decision about which secure transport layer protocols to use, and to compare and contrast different secure transport protocols, it is important that we have a good understanding of the properties that they can provide. To do this, we require a means to specify these properties precisely.

The aim of this thesis is to improve our understanding of the security guarantees that can be provided by secure transport protocols. We define a framework in which one can capture security properties. We describe a simulation relation over specifications based on the events performed by honest agents. This simulation relation allows us to compare channels; it also allows us to specify the same property in different ways, and to conclude that the specifications are equivalent.

We describe a hierarchy of confidentiality, authentication, session and stream properties. We present example protocols that we believe satisfy these specifications, and we describe which properties we believe that the various modes of TLS satisfy. We investigate the effects of chaining our channel properties through a trusted third party, and we prove an invariance theorem for the secure channel properties.

We describe how one can build abstract CSP models of the secure transport protocol properties. We use these models to analyse two single sign-on protocols for the internet that rely on SSL and TLS connections to function securely. We present a new methodology for designing security protocols which is based on our secure channel properties. This new approach to protocol design simplifies the design process and results in a simpler protocol.

Contents

Chapter 1

Introduction

Computer and communication security is important, sometimes critically so, to everyone. However, many people do not realise the role that security protocols play in protecting their personal information, and many people take security for granted. It is only when something goes wrong that we begin to appreciate the need for reliable and verifiable security.

The world is becoming strongly dependent on computers, and on distributed communication between remote devices. Large amounts of money are regularly moved between banks, merchants and customers electronically; without suitable security procedures and policies it would be easy for a malicious attacker to steal money, and to defraud these banks, merchants and customers. Many companies collect and store vast amounts of information about their customers; for example, a customer purchasing a life insurance policy has to provide most of their personal details to the policy provider. These personal details are precisely what an attacker needs to steal a person's identity, so businesses must be able to persuade their customers that their personal information is kept securely. Businesses and governments use email to send highly sensitive data over insecure networks; this sort of communication is infeasible without security protocols to protect the confidentiality and integrity of the data en-route. E-commerce is becoming more and more widely used: many merchants trade solely online, and today's e-commerce customers are willing to spend large amounts of money on the internet; without carefully designed security protocols the risks involved for customers, merchants and banks would be too great.

All of these day-to-day activities, and many more, rely on security protocols in order to achieve authentication, confidentiality and data integrity between remote computers communicating over distributed, and inherently insecure networks. Finding strategies for discovering attacks against security protocols and for proving security protocols correct is an extremely important area of research.

When security researchers design security protocols, they take for

granted the fact that they can build their protocol on top of a reliable transport layer protocol. They do not concern themselves with routing issues (how best to get a message from A to B), with technical details of message transmission or reception, or with necessary details such as error checking. Naturally, the security protocol is much simpler once these properties can be assumed of the transport layer.

An increasingly popular technique that is being used to simplify further the design of security protocols is to rely on a secure transport layer to protect messages on the network, and to provide some protection against attackers; see e.g. [FRH+08, OAS05b, Vis06, WSF+03]. By depending on a lower level protocol to provide a secure channel between hosts, a security protocol can be designed in a more transparent way. This simplifies the design of the security architecture: the designer can use an off-the-shelf secure transport protocol, such as TLS [DA99], to provide secrecy and authentication guarantees; the architecture can then provide additional security guarantees in a higher layer.

In such circumstances it is important to understand what is required of the secure transport protocol, and, conversely, what services are provided by different protocols. TLS provides strong guarantees; however, it is computationally-expensive, and so in some circumstances, a simpler protocol might suffice. On the other hand, unilateral TLS does not authenticate the client to the server; if the client does not have a public-key certificate, then authentication of the client must take place in the application layer. In this case TLS might not provide all the security guarantees that are necessary. In order to make the right decision about which secure transport layer protocols to use, it is vitally important to have a good understanding of the properties that they each provide. In order to do this, we require a means to specify these properties precisely.

In 2003, Philippa Broadfoot and Gavin Lowe proposed a layered approach to designing and analysing security transactions [BL03]. With their approach, a security transaction is layered over a lower-level secure transport protocol. The properties of the transport protocol are specified formally, and they can then be modelled abstractly.

There are several advantages to adopting this abstract layered approach. It is traditional to abstract away the details of the lower levels in a protocol analysis; we do not build an explicit model of TCP or IP when we analyse a security protocol, we assume that there is a network capable of sending messages between agents. Similarly, security protocol designers do not design their protocols to replicate the features of a reliable transport protocol. It makes sense therefore to model the properties of the secure transport layer abstractly rather than modelling the transport layer protocol itself; in order to do this, we require a precise way of specifying these properties.

There are many secure transport protocols that provide the same or similar properties. If two protocols provide the same properties, then one ought

to be able to use them interchangeably. If one protocol is stronger than another, then the stronger protocol can be used in any scenario instead of the weaker one. If a particular application-layer security protocol requires certain properties from the secure transport protocol then any transport-layer protocol that provides these properties can be used. In order to compare and equate secure transport protocols in this way we require a framework in which we can specify their properties.

The aim of this thesis, therefore, is to improve our understanding of the security guarantees that can be provided by secure transport protocols. We define a general framework in which one can capture security properties using CSP-style trace specifications, building on the work of Broadfoot and Lowe [BL03]. We provide a simulation relation over secure transport protocol specifications based on the events performed by honest agents. This simulation relation allows us to compare channels; if an architecture is correct when it uses a particular secure channel, it will still be correct when it uses a stronger channel. Our formalism also allows us to specify the same property in different ways, and to conclude that the specifications are equivalent.

We describe a hierarchy of confidentiality, authentication, session and stream properties. The session property groups messages into sessions by specifying that all messages received in a single connection were sent in a single connection; the stream property extends the session property by specifying that messages are received in the same order as that in which they were sent. We present example (single-message) protocols that we believe satisfy these specifications, and we use the various modes of TLS as a running example, and describe which properties that we believe they satisfy. We use our simulation relation to justify some simplifications on the model of the intruder, and to justify situations in which a weaker protocol can be used instead of a stronger one.

We investigate the effects of chaining our confidentiality and authentication properties through a trusted third party (a proxy). We prove an invariance theorem for the secure channel properties, and we prove that, in some cases, two channels can be chained through a proxy to produce a stronger channel.

We describe how one can build abstract models of the secure transport protocol properties in Casper, and we use the simulation relation to prove the soundness and completeness of these models. We show how these models can be used to analyse layered architectures; we study two single sign-on protocols for the internet that rely on SSL and TLS connections to function securely. Finally, we present a new methodology for designing security protocols which is based on our secure channel properties; we describe a single sign-on protocol that was developed using these new techniques. This new approach to protocol design simplifies the design process and results in a simpler protocol.

Thesis overview

This thesis is split into eight chapters. In the next chapter, we give an overview of the necessary background material for the later chapters. We describe what a security protocol is, how attacks against security protocols can arise, and how they can be fixed. We outline the formal notation that we use to define security protocols, and the symbolic encryption and active intruder model that we use when we analyse security protocols. We then describe some basic CSP notation, and give a brief overview of the traces semantic model for CSP. We outline the CSP model of a security protocol, developed by Lowe [Low96], and the method of using this model to look for attacks against a protocol. We introduce the extension to this model proposed (and implemented) by Broadfoot and Lowe [BL03]; we build upon many of their ideas in the later chapters of this thesis. We next study several different secure transport protocols, and discuss some of the properties that they provide to higher layers. We conclude the chapter with a brief consideration of alternative approaches for analysing and studying security protocols, and for proving protocols correct.

In Chapter 3 we describe our research into specifying secure channel properties. We first formalise our model of a set of honest agents who communicate over an insecure network. We then define several confidentiality and authentication properties, and we investigate the ways in which they can be combined. We identify several collapsing cases in which a combination of the channel properties simulates (i.e. allows the same attacks as) another combination. We enumerate all possible collapsing cases, and we define a hierarchy of specifications of properties that do not collapse. These properties are specified by placing restrictions on the activity of the intruder, and they are formulated as trace specifications parameterised by the channel under consideration. We discuss some of the points in the hierarchy in depth, and give examples of simple (single-message) protocols that we believe implement them. We then describe several session and stream properties that are independent of the confidential and authenticated properties.

In Chapter 4 we describe several useful results about secure channel properties. We first define a simulation relation based on the traces of systems as they are seen by honest agents; this simulation relation examines the effect of the intruder's behaviour rather than the events that he performs. We use this simulation relation to define an equivalence relation. We then use this relation to prove the equivalence of alternative specifications of our authentication properties; these alternative forms are more conducive to proving properties about secure channels. We show that every combination of the channel primitives introduced in Chapter 3 collapses uniquely to a point in the hierarchy, and we prove that some combinations of events that the intruder can perform can safely be blocked. We define a sufficient condition of an application-layer protocol for a particular weaker channel to be used

instead of a stronger one. Finally we compare our specification framework and our channel properties to properties developed by other researchers.

In Chapter 5 we examine the possibilities for chaining secure channels. We consider chaining channels in two different ways: firstly, though a set of dedicated intermediaries (which we refer to as simple proxies), and secondly, through a smaller, more general set of multiplexing proxies. We prove that in both cases the secure channel properties from Chapter 3 are invariant under chaining. We prove that the overall channel established through a proxy is at least as strong as the greatest lower bound of channels to and from the proxy, and we show that some combinations of channels can be chained to provide a stronger overall channel. We conclude this chapter by comparing our proxy results to similar results discovered by other researchers.

In Chapter 6 we present our abstract CSP models of the secure channel properties from Chapter 3. We first describe and characterise the existing Casper model [Low98], and we prove that it is equivalent to the network we described in Chapter 3, modulo the application-layer protocol being modelled. We then describe the new models that we have implemented in Casper and we prove that they are equivalent to the formal channel properties, again, modulo the application-layer protocol being modelled. These new models use the existing Casper structure, and so only small changes were necessary to Casper. Lastly, we describe the syntax for using the new channel models in a Casper protocol analysis.

In Chapter 7 we report on our analysis of two single sign-on protocols for the internet. We first introduce the single sign-on paradigm, and we describe the channel properties we use to model unilateral and bilateral TLS connections. We then describe our model of the SAML Single Sign-On protocols [OAS05b], and we describe several possible attacks that we found when the protocol is not implemented precisely according to the specification. Next we describe our model of the OpenID Authentication protocols [FRH+08], and we describe a serious attack that is possible when users choose insecure identities. We also describe several other attacks that are possible when the protocol is not implemented exactly as described (and sometimes implied) by the specification. Finally we present our own single sign-on protocol that we designed to be as concise as possible. We discuss the exact requirements of the secure transport layer, and we analyse this protocol for attacks. This new design methodology (using the channel properties from Chapter 3) simplifies the design process, and results in a much simpler protocol.

Finally, in Chapter 8 we conclude by summarising the contributions of this thesis. We describe several possible extensions to the specification and modelling work, and we discuss how this future work might be researched. We discuss possible extensions to the channel properties themselves, and interesting areas for studying the interactions between different secure channels. We describe some useful enhancements that could be made to the

Casper channel models. One particularly interesting area for future research would be to classify some of the existing secure transport protocols (described in Chapter 2), and to prove which properties they satisfy.

Note Gavin Lowe and I presented joint author papers based on the work in Chapter 3 at CSF'08 [DL08] and WITS'07 [DL07]. As a result of this, some of the wording in this chapter was written by Gavin, but the work was done by me with guidance from Gavin. The work in Chapter 5 was presented at FCS-ARSPA-WITS'08 [Dil08].

Chapter 2

Background

In this chapter we present some background material that the research described in this thesis builds upon. In the first section we introduce the notion of a security protocol, and we present a simple example; we also outline the abstract notation used to describe security protocols in the literature. In Section 2.2 we describe the symbolic encryption model and the active intruder against whom protocols must be resistant; this model of an active attacker is often attributed to Dolev and Yao [DY83].

In Section 2.3 we give a brief introduction to the language and trace semantics of Communicating Sequential Processes (CSP). In Section 2.4 we describe the CSP model of security protocols that was first presented by Lowe [Low96], and which has been extended by several other researchers. We use this model as the basis for the technical work presented in later chapters. In 2003 Broadfoot and Lowe proposed an extension to the CSP model to analyse layered protocols which assume the existence of a more secure transport layer than the traditional Dolev-Yao style network [BL03]; we describe this model in Section 2.5.

One of the main goals of this thesis is to extend the work of [BL03] by providing a framework for specifying and analysing the properties of these sorts of layered architectures. In Section 2.6 we describe several secure transport layer protocols; these protocols are designed to establish secure channels between remote devices over an insecure network.

In Section 2.7 we discuss alternative security protocol analysis techniques. We give a description of other approaches to analysing and specifying secure transport layer protocols in Chapter 4.

2.1 Security protocols

In this section we define what we mean by a security protocol. We give a simple example, and show that although a protocol appears to be valid, there may be subtle attacks against it.

A security protocol is a predefined series of messages to be sent between two or more agents in order to establish a secure communication link, or to achieve a security-related goal over an insecure network. The messages in the protocol typically use cryptographic primitives such as encryption, cryptographic hash functions and digital signatures to achieve the goal of the protocol, and the protocol should be resistant to disruption, even in the presence of a dishonest intruder who is assumed to have control over every point of the network.

A good example of a security protocol is the Needham-Schroeder Public-Key Authentication protocol [NS78]. In this protocol two agents, A (the initiator) and B (the responder), wish to authenticate one another; the goal of this protocol is therefore to make sure that whenever A has been running the protocol, supposedly with B, B was running the protocol with A, and vice versa. In order to achieve their goal, both agents trust a third party AS: the authentication server who informs them of each other's public keys in a reliable and verifiable manner.

The notation $A \rightarrow B : m$ signifies that agent A sends message m to agent B. The message m_1, m_2 is the concatenation of the messages m_1 and m_2, and we write $\{m\}_k$ to indicate that m is encrypted with the key k. PK is a function that returns the public (encryption) key for any agent, and SK returns the secret (signing) key. For any agent B, $SK(B)$ and $PK(B)$ are inverse keys.

$$
\begin{array}{ll}
\mathit{Message}\,1 & A \rightarrow AS : A, B \\
\mathit{Message}\,2 & AS \rightarrow A : \{PK(B), B\}_{SK(AS)} \\
\mathit{Message}\,3 & A \rightarrow B : \{N_A, A\}_{PK(B)} \\
\mathit{Message}\,4 & B \rightarrow AS : B, A \\
\mathit{Message}\,5 & AS \rightarrow B : \{PK(A), A\}_{SK(AS)} \\
\mathit{Message}\,6 & B \rightarrow A : \{N_A, N_B\}_{PK(A)} \\
\mathit{Message}\,7 & A \rightarrow B : \{N_B\}_{PK(B)} .
\end{array}
$$

A first requests B's public key from the server (Message 1), who signs B's public key and B's identity then sends the signed message to A (Message 2). This public key certificate is signed with the server's secret key, so A can verify that she has been given the correct key because she already knows AS's public key. Now that A knows B's public key, she chooses a nonce (a large random number) and encrypts that and her identity with $PK(B)$ and sends them to B (Message 3). B is the only agent who knows $SK(B)$, so he is the only agent who can decrypt this message to learn N_A. B requests A's public key certificate from AS (messages 4 and 5) and then chooses a different nonce N_B to encrypt and send back to A (along with A's original nonce). The last step of the protocol is for A to prove that she knows $SK(A)$ by sending B's nonce back to him, encrypted with his public key.

We simplify the protocol by assuming that A and B have previously run

the protocol and so know each other's public keys:[1]

$$\begin{array}{ll} \textit{Message}\,1 & A \rightarrow B : \{N_A, A\}_{PK(B)} \\ \textit{Message}\,2 & B \rightarrow A : \{N_A, N_B\}_{PK(A)} \\ \textit{Message}\,3 & A \rightarrow B : \{N_B\}_{PK(B)}\,. \end{array}$$

Because the nonces N_A and N_B are always encrypted with either A's public key or B's public key, apparently only A and B can learn them. When A sends her nonce in Message 1, she knows that only B can decipher it; when she receives Message 2 she believes that she can deduce that B is running the protocol because he has included N_A in a new message. In order to authenticate herself to B (i.e. prove that she can decode something encrypted with $PK(A)$) she sends B's nonce back to him in Message 3.

In the presence of an active intruder, this protocol does not achieve its goal (stated above). The intruder can persuade B that he was running the protocol with A, when A was actually running the protocol with the intruder, I.

$$\begin{array}{lll} \alpha.1 & A \rightarrow I : \{N_A, A\}_{PK(I)} & \\ & & \beta.1 \quad I_A \rightarrow B : \{N_A, A\}_{PK(B)} \\ & & \beta.2 \quad B \rightarrow I_A : \{N_A, N_B\}_{PK(A)} \\ \alpha.2 & I \rightarrow A : \{N_A, N_B\}_{PK(A)} & \\ \alpha.3 & A \rightarrow I : \{N_B\}_{PK(I)} & \\ & & \beta.3 \quad I_A \rightarrow B : \{N_B\}_{PK(B)}\,. \end{array}$$

There are two runs of the protocol in this attack. In the first run (α) the intruder, I, and A run the protocol normally. In the second run (β) the intruder pretends to be A and runs the protocol with B. The first thing he does is to replay A's nonce from run α to B in run β. Then, when he receives Message 2 (which he cannot decrypt) he relays it to A in run α. Because he used the same nonce as A in the first message, A accepts this message as part of run α, and returns the nonce N_B to the intruder. The intruder can now complete run β by encrypting N_B with B's public key, and returning it to him. This attack was discovered by Lowe [Low96].

The attack is possible because Message 2 does not contain B's identity, so while the intruder cannot learn N_A and N_B from $\{N_A, N_B\}_{PK(A)}$, he can hijack the entire message, and pass it on to A as a message that he (the intruder) created. If the protocol directed the responder to include their identity in Message 2 the intruder would not be able to hijack it, and would not be able to perform this attack:

$$\begin{array}{ll} \textit{Message}\,1 & A \rightarrow B : \{N_A, A\}_{PK(B)} \\ \textit{Message}\,2 & B \rightarrow A : \{N_A, N_B, B\}_{PK(A)} \\ \textit{Message}\,3 & A \rightarrow B : \{N_B\}_{PK(B)}\,. \end{array}$$

[1] We also assume that the public keys have not been changed, or expired.

2.2 The Dolev-Yao model

In this section we describe the formal model we use to model and analyse security protocols; this model was first proposed by Dolev and Yao [DY83].

We treat encryption (and cryptography in general) as a perfect system: we are not trying to detect flaws in security protocols that are due to attacks against the cryptosystem in use. For this reason we model encryption symbolically (in the style of Needham and Schroeder [NS78], and Dolev and Yao [DY83]). We write $\{m\}_k$ to mean that the message m is encrypted with the key k.

We consider two types of cryptographic key: symmetric keys are self-inverse, while asymmetric keys are not. We assume that most agents possess mutually inverse public and secret key pairs; the functions PK and SK return the public key and secret key for any agent. We assume that every agent knows the PK function (i.e. every agent knows every public key), but that an agent's secret key is only known to that agent.

We treat encryption and decryption as symbolic operations; encrypting a message twice with the same symmetric key leaves the message unchanged; encryption with an asymmetric key followed by encryption with the inverse key also leaves the message unchanged. We note that while encryption and decryption are inverse operations, they are not commutative. We assume that ciphertexts contain enough redundancy that we can distinguish a valid ciphertext from a random stream of bits. We also assume that encrypted messages can only be decrypted with the inverse key to the key they were originally encrypted with; trying to decrypt an encrypted message with any other key results in garbage.

We model deductions over the message space with the $\vdash$ relation: if the message m can be deduced from the set of messages M we write $M \vdash m$. Whether or not m can be deduced from M is decided by the following five rules:

Membership $m \in M \Rightarrow M \vdash m$;

Splitting $M \vdash m_1, m_2 \Rightarrow M \vdash m_1 \wedge M \vdash m_2$;

Concatenation $M \vdash m_1 \wedge M \vdash m_2 \Rightarrow M \vdash m_1, m_2$;

Encryption $M \vdash m \wedge M \vdash k \Rightarrow M \vdash \{m\}_k$;

Decryption $M \vdash \{m\}_k \wedge M \vdash k^{-1} \Rightarrow M \vdash m$.

The deduction relation also satisfies the following two properties:

Monotonicity $X \subseteq X' \Rightarrow \{m \mid X \vdash m\} \subseteq \{m \mid X' \vdash m\}$;

Transitivity $X \vdash m \wedge X \cup \{m\} \vdash m' \Rightarrow X \vdash m'$.

For example $\{k^{-1}, \{a, b, m\}_k\} \vdash m$, by one application of **decryption** and one application of **splitting**. These rules capture the property that an agent (including the intruder) can only encrypt or decrypt a message if he possesses the requisite key.

This abstract model of cryptography is not the only approach taken by security researchers; several researchers model encryption as an operation over strings of bits (e.g. [BR95, Kra01, Can01, War05, BPW06]). In this 'computational' model encryption keys and messages are modelled as finite bit strings, and the encryption and decryption functions transform plaintext bit strings into ciphertext bit strings. Security results are given in terms of the probability that the intruder can disrupt the goal of the protocol.

In [AR02] Abadi and Rogaway prove a soundness result for a symbolic treatment of symmetric (shared-key) encryption. They establish that secrecy properties that can be proved in the symbolic world are true in the computational world, and thus justify the use of a symbolic treatment of encryption.

The Dolev-Yao model is based on a collection of honest agents who communicate over a network. These honest agents only run the protocol in question: they can perform multiple concurrent runs of the protocol, but they do not deviate from the message flows described by the protocol.

We assume that the network over which the honest agents communicate is under the control of a dishonest intruder who can overhear everything that is sent on the network, block messages from being received, alter messages in transit, and inject fake messages. The intruder can also use these abilities to re-order and delay messages, and to replay old messages. The intruder's ability to create new messages is limited by the deduction rules described above: he can only send or fake messages that he can deduce from what he knew initially and what he has overheard.

We assume that the intruder's initial knowledge includes all 'public' material such as honest identities, public keys, hash functions, etc. We frequently call on the honest agents to create fresh, random values (nonces); we assume that the intruder cannot guess the values of these nonces, and he can create his own. We formalise our interpretation of the honest agents, the intruder and the network in Chapter 3.

2.3 CSP

CSP is a language for describing concurrent systems of communicating processes; CSP was first introduced by Hoare in 1978 [Hoa78]. Since then, Hoare, Roscoe and others have extended the language [Hoa85, Ros98], and developed a collection of mathematical models which support the analysis of systems described in CSP. In this section we give a brief introduction to the CSP language and to the traces model ($\mathcal{T}$). We use this model, and the

notation described in this section in Chapter 3 as our specification language for secure channel properties. We also use some of the CSP notation and the properties of the traces model for the proofs in Chapters 4 and 5, and for describing the updated models in Chapter 6. For a full description of CSP and of the traces model, see Roscoe's book [Ros98].

A CSP process is described by the events it can perform. An event is an atomic communication between processes; events may be simple (e.g. *send*) or they may carry data (e.g. *send.m*). In the case of complex events we consider the first part of the event to be the channel name; if c is a channel then $\{|\, c \,|\}$ is the set of events over c.

The process $STOP$ is the process that cannot perform any events. The process $a \rightarrow P$ performs the event a and then behaves like the process P. The input prefix $c?x \rightarrow P(x)$ accepts any value x (of the correct type) on the channel c (i.e. performs the event $c.x$) and then behaves like the process $P(x)$; the output prefix $c!x \rightarrow P$ performs the event $c.x$ (an output event) and then behaves like the process P. The process $P \,\Box\, Q$ (external choice) can either behave as process P or as process Q; the indexed form of external choice $\Box_{i \in I}\, P_i$ can behave as any of its arguments.

Processes can be combined with the parallel operator. The process $P \underset{X}{||} Q$ behaves like the processes P and Q acting concurrently with the condition that they must synchronise on all events in X. In the special case that $X = \{\}$ we say that the processes are interleaved: they perform events independently of one another; we write $P \,|||\, Q$.

A trace is a sequence of events that a CSP process might perform; for example, $\langle a_1, a_2, \ldots, a_n \rangle$ is the trace containing the events a_1 to a_n in that order; $\langle\rangle$ is the empty trace. If tr and tr' are two finite traces then $tr^\frown tr'$ is their concatenation. We write $tr' \leqslant tr$ if tr' is a prefix of tr. We write a **in** tr if the event a occurs in the trace tr, and $tr \downarrow c$ is the sequence of data communicated in tr over the channel (or set of channels) c. $tr \upharpoonright c$ is the sequence of events communicated in tr restricted to the set of events c. For a channel c (or a set of events X) we write $tr \setminus c$ ($tr \setminus X$) for the trace tr with the events on channel c (or the events in the set X) removed.

In the traces model $\mathcal{T}$ the semantics of a process P is defined as the set of traces it can perform: $traces(P)$. For any process P, $traces(P)$ is non-empty (it always contains the empty trace $\langle\rangle$) and prefix closed (i.e. if tr is a trace of P then tr' is also a trace of P for every $tr' \leqslant tr$). In Chapter 3 we give channel specifications as predicates over traces; a process P satisfies a specification if all of its traces satisfy the specification. One CSP process Q refines another process P (written $P \sqsubseteq Q$) in the traces model if every behaviour of Q is also a behaviour of P; i.e. $traces(Q) \subseteq traces(P)$.

2.4 CSP models of security protocols

The example in Section 2.1 shows how easy it is to design a protocol that appears to be secure, but is in fact not secure because an active intruder can attack it and disrupt the protocol goals. In this section we describe the formalisation of the Dolev Yao model in CSP, and we describe Casper and FDR, the tools that we use for finding attacks against security protocols. Casper was initially developed by Lowe [Low98], but has since been extended by several authors [RB99, HL01, BL03]. Casper automatically builds a CSP model of a protocol from an abstract description, and constructs trace refinement properties that can be checked with FDR [FSE05]. If FDR finds that one of these trace refinement properties does not hold it produces a counterexample trace; Casper takes this counterexample and produces an attack trace (similar to the one shown above).

The system that Casper builds comprises:

- A process for each honest agent; these processes follow the steps in the protocol and are parameterised by the values (nonces, keys, etc.) they introduce in the protocol run. Several copies of each honest agent can be run sequentially or concurrently in order to model that agent running the protocol more than once, or with more than one agent at a time;
- A process for the intruder; this process is built of two parts: one part that can say anything the intruder knows, and the other that builds upon his knowledge by deducing new messages from what he knew initially (this is specified in the Casper input script) and from what he hears the honest agents saying. Because the intruder can say anything he knows he can take part in protocol runs just as the honest agents do.

The system is put together in two steps: first a process is built that interleaves all the honest agent processes; this process is then run in parallel (synchronising on all events) with the intruder process. The intruder hears every message that the honest agents send, and decides whether or not to let an agent receive a message that is sent to him. The intruder process can modify messages before they are received, and can fake messages so that honest agents receive messages that were not sent to them by other honest agents. We also allow the intruder to assume legitimate identities (which the honest agents cannot distinguish from other honest agents) and to take part in protocol runs using these identities. For a more complete description of the CSP model see [RSG+01].

The final stage in building the CSP system is to write trace refinement properties that the model checker FDR [FSE05] can check to look for attacks against the specified goals of the protocol. The Casper user specifies one or

more secrecy or authentication specifications and an actual system (i.e. the configuration of honest agents and the intruder's initial knowledge) in the Casper script. If FDR fails to find traces that invalidate the property it does not mean that the protocol is secure: it just implies that there are no attacks against the particular (limited) system being tested.

The authentication properties that Casper can check are described in [Low97] and [Low98] and include:

Aliveness whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol;

Weak agreement whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A;

Non-injective agreement on *ds* whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agree on the set of data values *ds*;

Agreement on *ds* non-injective agreement on *ds* holds, and every run of A (acting as initiator), apparently with responder B, corresponds to a unique run of B (acting as responder), apparently with A.

The confidentiality properties supported by Casper are of the form: A thinks that *na* is a secret that can only be known to himself and B; a counterexample to these specifications is a trace that results in the intruder learning the secret value *na*.

Lowe and other researchers have extended the Casper model in order to analyse larger systems, and to draw stronger conclusions from a Casper analysis:

- Roscoe and Broadfoot used data independence techniques to simulate a system where agents can use an infinite supply of nonces, keys, etc. with a finite CSP model [RB99]. These systems allow a finite check in FDR to verify protocols in which the agents can perform an unbounded number of sequential runs, and a fixed number of concurrent runs. Roscoe, Broadfoot and Lowe extended these techniques, and build automated support for them into Casper [BLR00].

- Hui and Lowe built automated support for their simplifying transformations into Casper [HL01]; these are simplifying transformations that have the property of preserving insecurities. A verification of a simplified protocol is also a verification of the original protocol; any attacks

that are possible against the original protocol are also possible against the simplified protocol.

- Broadfoot and Lowe adapted the Casper model to analyse layered architectures in an abstract way [BL03]; we describe this work in more detail in the next section.

Lowe also proved that, subject to some conditions on the protocol being studied, the verification of a protocol running in a particular small system can be extended to a proof of the security of the protocol in a larger system [Low99].

2.5 A layered approach

In [BL03] Broadfoot and Lowe describe an adaptation to the CSP model of security protocols to model, abstractly, some of the properties we might desire a secure transport layer to provide. Any secure transport layer will, necessarily, limit the activity the intruder can perform. Introducing channels with properties which limit the intruder's capabilities means changing the model of our systems. Rather than modelling the transport layer concretely, Broadfoot and Lowe abstract away from the details, and model the services it provides. The limitations they place on the intruder are such that exactly the traces corresponding to application-layer entities communicating on a transport layer with the specified properties are allowed by the model.

Broadfoot and Lowe specify two types of secure channel; the specifications below are drawn directly from [BL03]. These properties can be expressed in terms of the channel primitives developed in Chapter 3, and so can be compared directly with the channels in our hierarchy; we discuss how this can be done in Chapter 4.

Authentication and integrity requirements Within each session between A and B, the messages accepted by the transport layer of B (and so passed to the application layer) as coming from A are a prefix of those sent by A intended for B:[2]

$$\forall A, B \in \mathit{Honest}; s \in \mathit{Session} \cdot tr \downarrow \mathit{receive}.B.A.s \leqslant tr \downarrow \mathit{send}.A.B.s\,.$$

This definition is a very strong form of authentication and integrity. If an honest agent B receives message m in session s, purportedly from A, he can be sure that it was originally sent by A, that A intended it for him, and, moreover, that the session A sent the message in is the same as the session in which B received the message. If all of the channels in the model satisfy this property, the intruder can only delay messages (re-ordering is prevented

[2] *Honest* is the set of honest agents; *Session* is the set of (shared) session identifiers.

by the prefix condition), and send messages from his own identities. These channels are modelled in CSP by placing a buffer on the output channel of each honest agent, and connecting the buffer's output directly to the intended recipient's input channel. The intruder overhears everything that is sent into the buffer, and controls whether the buffer outputs or not. The intruder cannot remove messages from the buffer, nor can he change the message order inside it, or add messages to it.

Secrecy requirements An agent B can receive message m from the intruder only if that message can be produced from the intruder's initial knowledge (IIK) and those messages that previously have been deliberately sent to him:[3]

$$\forall B \in Honest; I \in Dishonest; s \in Session; m \in Message; tr' \in Trace \cdot \\ tr'^\frown\langle receive.B.I.s.m\rangle \leqslant tr \Rightarrow IIK \cup sentToIntruder(tr') \vdash m\,.$$

These channels prevent the intruder from overhearing messages sent between the honest agents; he can only learn from what the honest agents send to him. These channels are modelled by a direct communication channel between the honest agents; the intruder can add messages to this channel, remove them from it, and re-order or replay messages sent on this channel, but he cannot learn the content of these messages.

2.6 Secure transport protocols

In the previous section we described Broadfoot and Lowe's strong authentication and confidentiality properties of secure transport layers. One of the primary goals of this thesis is to provide a framework for specifying these sorts of properties, and to come up with more general specifications of these properties. In this section we describe five commonly used secure transport layer protocols. We give an abstract overview of the protocols, and we discuss some of the properties that they provide.

2.6.1 SSL and TLS

The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols were designed to provide secure communication channels between networked devices in order to protect application-layer protocols. SSL was developed by Netscape, and version 3.0, which fixed several flaws in version 2.0, was published in 1996 [FKK96]. In 1999 the IETF published the TLS protocol [DA99], which is based on version 3.0 of SSL. TLS provides

[3] *Dishonest* is the set of dishonest identities: the identities the intruder can assume; *Message* is the set of messages; the function *sentToIntruder* gives the set of messages sent by honest agents to the intruder in a trace.

support for unilateral or bilateral (mutual) authentication, and for a variety of message authentication and encryption algorithms. In the rest of this section we discuss the TLS protocol, and some of the security properties it provides to application-layer protocols that use it.

The TLS protocol has two phases: the first phase, the handshake protocol, is used to authenticate one, or both, of the agents taking part in the protocol, and to establish fresh secret keying material; the second stage, the record layer protocol, uses the key material established in the first stage to provide a confidential and authenticated communication link between the two parties. The following abstract description of the TLS handshake protocol is based on the description given in [Pau99].

The TLS handshake protocol is run by a client and a server; the client initiates the protocol by sending a message to the server which includes the client's identity, a session identifier, a fresh nonce and his set of preferences for encryption and message authentication. In response, the server sends a message containing the client's session identifier, another fresh nonce, his set of encryption and message authentication preferences (a subset of the client's preferences) and his public key certificate. The client then generates the *premaster secret* (another nonce), encrypts this with the server's public key and sends it to the server. The client then calculates the *master secret* from the premaster secret and both nonces. In order to prove his identity, the server must decrypt the encrypted premaster secret to calculate the master secret, and then prove that he knows the master secret.

The master secret is used to derive four new secrets: two of these are used for keying MAC functions, and two are used for encryption. Different MAC keys and encryption keys are used by the client and server when they send messages in the record layer protocol. The handshake protocol concludes with each agent calculating a hash of all the messages they have exchanged so far, encrypting this hash with the newly established keys, and sending the hash to the other agent. When the client receives the encrypted hash from the server he knows that the server is who he says he is, because only the server could have decrypted the premaster secret to learn the master secret.

The client may, optionally, authenticate himself by sending his public key certificate, and signing a hash of all the previous handshake messages with his secret key. The server verifies the client's identity by checking this signature against the public key certificate the client sent, and by creating the hash himself, and comparing it to the value that the client sent. The TLS handshake protocol can therefore be run in unilateral mode (where only the server's identity is authenticated), or in bilateral mode (where both agents' identities are authenticated). TLS also provides support for running the handshake protocol in unauthenticated mode, and creating the premaster secret with a Diffie-Hellman exchange [DH76]; however, anonymous Diffie-Hellman exchanges are vulnerable to man-in-the-middle attacks [Res01], so

unauthenticated TLS is not recommended.

The TLS handshake protocol authenticates one or both agents to one another, and establishes fresh secrets that both parties have contributed entropy to; these secrets are the client-write MAC secret, the client encryption key, the server-write MAC secret and the server encryption key. The application-layer message m is sent in the record layer protocol as $\{m, h_{wk}(seqNo, m)\}_{ek}$, where wk is the relevant write MAC secret (client or server), h_{wk} is the HMAC function [KBC97] using key data wk, and ek is the relevant encryption key.

The encryption with the client or server encryption key protects the confidentiality of the message, and also binds the message to the current session (so that messages from one session cannot be replayed in another); because the client and the server use different encryption keys, the encryption key used also identifies the sender of the message. The encrypted component of the transport-layer message includes a keyed hash of the application-layer message and a sequence number. The hash protects the integrity of the message, and the sequence number prevents an intruder from removing or re-ordering messages in the stream of messages sent by either party without the message recipient detecting the change.

In Chapter 3 we describe which of our channel properties we believe that SSL and TLS provide. In Chapter 7 we study some application-layer protocols that rely on TLS and SSL connections to achieve their goals correctly and securely.

2.6.2 SSH Transport Layer protocol

The Secure Shell (SSH) protocol [YL06a] was designed to provide a secure replacement for insecure remote login shells such as telnet, rlogin and rsh which all send information, including authentication credentials such as the user's password, in the clear. The first version of SSH (SSH-1) was developed by Ylönen in 1995; at the end of 1995 Ylönen founded SSH Communications Security to market and develop SSH, and in 1996 SSH-2 was released. In 2006 the SSH-2 protocol was published as a series of RFCs [YL06a].

The SSH protocol has three major components: the transport layer handles initial key exchange, and authentication of the server to the client; the user authentication layer runs on top of the transport layer, and handles authentication of the client to the server by a number of possible methods (e.g. password or authentication of the user's public key certificate); the connection layer establishes channels on top of the SSH transport layer. A single SSH transport layer connection can support multiple channels. In the rest of this section we describe the SSH transport layer and the services it provides to the SSH user authentication and connection layers.

The SSH transport layer [YL06b] has similar functionality to unilateral TLS: it provides strong encryption, server authentication, message integrity

protection, and it establishes a secure session that prevents messages from being replayed or dropped. The SSH transport layer is based on two protocols: a key exchange protocol and the Binary Packet Protocol; the roles of these two protocols are comparable to the TLS handshake protocol and record layer.

The key exchange protocol is initiated by the client, and establishes a shared secret K, and an exchange hash H. The client starts the protocol by sending a fresh nonce and a set of preferences (for encryption, MAC and compression algorithms) to the server; the server then sends a fresh nonce and his set of preferences to the client. Both agents follow the same procedure to determine which algorithms to use, and then carry out a Diffie-Hellman exchange [DH76] to establish the shared secret.

The client and the server choose two secret values x and y; the client calculates $e = g^x \bmod p$ and the server calculates $f = g^y \bmod p$, where p is a large safe prime (i.e. a prime number p such that $p = 2q + 1$ and q is also prime), and g is a generator for an order q subgroup of the field $\mathbb{Z}/p\mathbb{Z}$ [YL06b]. The client then sends e to the server.

The server calculates the shared secret and the exchange hash. The shared secret K is the value established by the Diffie-Hellman exchange (i.e. $K = g^{xy} \bmod p$); the exchange hash H is the hash value of the client's identity, the server's identity, the first two protocol messages (including both nonces), the server's public key certificate, the values e and f from the Diffie-Hellman exchange, and the secret K. The server signs the exchange hash with his secret key, and sends the signed hash, his value f and his public key certificate to the client.

The client calculates the shared secret and the exchange hash independently, and then verifies the server's signature on the exchange hash. The client and server have now established a shared secret (K) and the exchange hash (H), and the server has been authenticated to the client. They both use these values to calculate six secret values: a different initialization vector (IV), MAC key and encryption key for communication in both directions (client to server and server to client). The exchange hash is then used as the session identifier.

SSH application-layer messages are carried by the Binary Packet Protocol. This protocol uses symmetric-key encryption in Cipher Block Chaining (CBC) mode and message authentication using HMAC to provide a confidential and authenticated stream of packets in both directions. An application-layer message m is sent as the transport-layer message $\{m, n\}_{ek}, h_{wk}(m, n, seqNo)$, where n is a nonce (random padding), ek is the appropriate symmetric encryption key, h_{wk} is the HMAC function using key data wk and $seqNo$ is an implicit sequence number. Sequence numbers start at 0, and are incremented with every packet; the sequence numbers in different directions are different. Further, the initialization vector for packet P_i is the final block from the encrypted block P_{i-1}.

The SSH transport layer specification document [YL06b] claims that the Binary Packet Protocol establishes an authenticated and confidential stream of messages in each direction. However, in 2004, Bellare et al. showed that the encryption method used by SSH is not secure [BKN04]. They describe an attack against the transport layer protocol whereby an active intruder can have his guess of a previously encrypted message confirmed by interacting with the protocol. They suggest three alternative ways of fixing the encryption scheme, but these fixes do not change the general message construction.

2.6.3 WEP and WPA

IEEE 802.11b, 802.11g and 802.11n are a set of standards for wireless (radio) communication between computers. As part of the original specification for wireless networks, the 802.11 working group specified a secure-transport layer protocol (Wired Equivalent Privacy – WEP) which was designed to protect the confidentiality of application-layer communication over a wireless network [LAN07].

An 802.11 wireless network does not, by default, contain any security measures to protect the communications over the transport layer. An intruder can overhear all communication between all agents (by listening to all radio traffic), he can inject and send his own messages, and he can, to some extent, block the local radio receivers of devices communicating on the network to prevent messages from being received. A secure-transport layer protocol is therefore absolutely vital if the wireless network is to be used for secure communication.

The following description of the WEP protocol is based on the description given in [BGW01]. WEP uses a secret key k, which is shared by all legitimate agents who communicate over the wireless network, and a random initialization vector v to generate a keystream (using the RC4 algorithm) to encrypt each application-layer message (i.e. each packet of data) before it is sent on the network. In order to send the message m to agent B, the agent A sends the following transport-layer message:

$$v, ((m, c(m)) \oplus RC4(v, k))$$

where c is an unkeyed integrity checksum function (this is implemented as a 32-bit cyclic redundancy check [PB61]).

This encryption mechanism is designed to keep the application-layer message (m) confidential, and to protect its integrity (by means of the integrity checksum). However, several attacks against the protocol have been discovered (see e.g. [BGW01, Wal00, SIR02]); these attacks are due to the short initialization vector (24-bits), which inevitably leads to keystream reuse.

WEP also specifies an authentication protocol which is designed to authenticate agents (requesters) to a wireless access point (the responder) by

proving that they know the secret key. The authentication protocol proceeds as follows: when an agent wishes to authenticate himself to the access point, the access point issues a plaintext challenge (a nonce) to the agent. The agent encrypts the challenge using the WEP encryption algorithm, and sends the encrypted message back to the access point.

However, as described in [BGW01], an attacker in possession of a plaintext/ciphertext pair of length l can fake a properly encrypted WEP message of length l without knowing the secret key. Once the intruder has observed a legitimate authentication sequence, he learns the plaintext challenge (the nonce), and the ciphertext sent by the agent in response to the challenge. He can use these to derive the keystream used to encrypt the response, and hence to respond to authentication requests without knowing the secret key k.

In light of the attacks against WEP that prevent it from reliably providing confidentiality, authentication and data integrity, the 802.11 working group published an amendment (802.11i) to the standard specifying new security mechanisms. During the development of the new security mechanisms the Wi-Fi Alliance created the Wi-Fi Protected Access protocol (WPA) [Wi-03], which was designed to replace WEP, and which implements most of the 802.11i standard.

802.11i addresses all the known vulnerabilities of WEP by adding mutual authentication between clients and the access point using 802.1X access control [LAN01] with Extensible Authentication Protocol (EAP) authentication [ABV$^+$04]; and by using the CCMP (Counter Mode with Cipher Block Chaining Message Authentication Code Protocol) AES-based encryption algorithm to provide message integrity checks, and unique keys for every encrypted message.

802.11i supports two modes of operation: in *enterprise mode* a separate authentication server is used to authenticate the user, and to establish a secure Pairwise Master Key (PMK); in *home mode* a pre-shared key is used to authenticate the user to the access point, and the PMK is derived from the pre-shared key. The enterprise mode supports several methods for EAP authentication: each method is designed to authenticate the user who wishes to join the wireless network (the supplicant) to the access point (via an authentication server), and vice versa. For example, EAP-TLS establishes a bilateral TLS connection between the supplicant and the authentication server; PEAP (Protected EAP) and EAP-TTLS (Tunneled TLS) establish a unilateral TLS (or SSL) connection from the authentication server to the supplicant, and then use MS-CHAPv2 to authenticate the supplicant over the secure channel (using a password).

Once the supplicant and the authentication server have authenticated one another and established a new secret PMK, the supplicant and the access point (the authenticator) run the 4-Way Handshake Protocol; this protocol is used to generate a fresh Pairwise Transient Key (PTK) for the new session,

and a Group Temporal Key (this is used to decrypt and encrypt multicast and broadcast network traffic). The new PTK is now used with CCMP to encrypt all messages exchanged in the session. CCMP combines Cipher Block Chaining Mode with Counter Mode to provide message integrity and privacy; in 2002 Jonsson proved the security of CCMP [Jon02].

The improved security mechanisms of 802.11i establish a strong secure transport layer for securing wireless networks. Mutual authentication of the supplicant and the authenticator is achieved using an EAP method, and then a new session is established between the two devices. The confidentiality and integrity of messages within the session is guaranteed by the AES-based CCMP encryption; the intruder cannot inject fake messages without knowing the PMK or the PTK.

2.6.4 IP Security

SSL and TLS were designed to provide secure communication channels by establishing connections on top of TCP connections. The Internet Protocol Security (IPsec) protocols [KS05] were designed to operate at the level of IP connections, and thus provide a lower-level secure transport layer. While applications must be designed specifically to use SSL or TLS connections, IPsec can be used invisibly as the services it provides can protect all protocols that can be carried over IP (including IP itself and TCP).

IPsec is a flexible, and configurable architecture that may be used to provide some of the following services: access control, connectionless message integrity, data origin authentication, detection and rejection of message replays and confidentiality [KS05]. Most of these security services are provided by two different protocols:

- The Authentication Header (AH) protocol [Ken05a] offers message integrity, authentication of the message sender and, optionally, replay protection;
- The Encapsulating Security Protocol (ESP) [Ken05b] offers the same services as AH, but it also offers confidentiality.

IPsec also specifies an automated key management protocol (the Internet Key Exchange (IKE) protocol [Kau05]) for establishing keys for use in the AH and ESP protocols. Since the services offered by AH are also offered by ESP, we focus only on IKE and ESP in the rest of this section. IPsec can operate in two modes:

- In transport mode only the IP payload is protected; this mode is used for host-to-host connections, as the usual IP routing methods are used. Transport mode provides protection for transport and application layer messages, but not for the IP layer headers.

- In tunnel mode the complete IP packet is protected; in this mode the IP packet must then be transported in a new IP packet for routing to work. Tunnel mode can be used for network-to-network, host-to-network and host-to-host connections.

An IPsec implementation uses a local Security Policy Database (SPD) to perform one of three actions to every packet sent through the host machine's IP interface: packets may be protected by the IPsec mechanisms, discarded, or they may bypass the IPsec mechanisms. The SPD provides a way to specify which security protocol should be used (AH or ESP), which options of that protocol to use, which mode to operate in (transport or tunnel), and which cryptographic algorithms to use for every packet that is protected by IPsec.

In the rest of this section we give a brief overview of the IKE and ESP protocols, and we summarise the security properties they provide. The Internet Key Exchange protocol is used in IPsec to establish Security Associations (SAs), and for mutual authentication of the parties running the protocol. A Security Association is a set of encryption and MAC algorithms together with shared secret information for use in those algorithms; a different SA is established for each direction of an IPsec channel.

The IKE protocol initiator sends a message to the responder containing his security association preferences (i.e. the list of encryption and MAC algorithms that he supports), a fresh nonce, and a Diffie-Hellman value. The responder sends back a similar message containing his choice of the suggested preferences, another fresh nonce, and his Diffie-Hellman value. Both parties can now compute *SKEYSEED*: a value derived from the nonces and the Diffie-Hellman secret by a pseudo random function. *SKEYSEED* is used later in the protocol for the derivation of secret keying material.

The next stage in IKE is for the initiator and the responder to authenticate one another. The initiator signs the entire contents of the responder's first message and the responder's nonce, and sends this in an encrypted message with his identity and his public key certificate. This message is encrypted using the negotiated encryption algorithm, and a key derived from *SKEYSEED*; the message is also protected with a cryptographic hash function which is keyed using material derived from *SKEYSEED*. The responder authenticates his identity by sending a similarly constructed message to the initiator.

The authentication messages described above are also used to establish the first child SA; the key material for this SA is generated by iterated application of a pseudo random function to *SKEYSEED*, both nonces, and the Diffie-Hellman value. At any stage after establishing this initial child SA the agents can establish new child SAs, or they can rekey an existing SA by sending new nonces and new Diffie-Hellman values encrypted under the existing SA keys. The IKE protocol may use EAP methods (as described in

Section 2.6.3) to authenticate one or both parties (rather than using public key certificates and message signatures).

Once the IKE has been run to establish an initial SA and child SA, the agents can send and receive IP packets that are protected by the Encapsulating Security Protocol. A message m is encoded as the following ESP packet: $sa, seqNo, EK(m, n), h(sa, seqNo, EK(m), n)$ where sa identifies the Security Association in use, *seqNo* is the sequence number of the packet (sequence numbers start at zero and are incremented with every packet), $EK(m)$ is the encrypted message (the encryption algorithm and encryption keys are defined by the Security Association), n is random padding, and h is the keyed MAC function defined by the Security Association.

In tunnel mode the ESP packet payload (the message m) is an entire IP packet (including all headers such as sender and destination); in transport mode the payload is just the IP packet payload (i.e. the transport-layer or application-layer message). The ESP protocol provides message confidentiality and integrity (via the encryption and MAC); data origin authentication (i.e. sender authentication) is provided indirectly by the keys used for the MAC and encryption, and the anti-replay and in-order delivery properties are provided by the authenticated sequence numbers.

2.7 Alternative methods

The CSP model described in Section 2.4 is not the only method that researchers are using to find attacks against security protocols, or to prove security protocols correct. In this section we give a brief overview of some of the alternative methods that have been developed.

2.7.1 Rank functions

Steve Schneider has developed an alternative CSP-based approach to proving that protocols satisfy security properties when any number of agents can run them concurrently any number of times [Sch98, SD05]. Schneider's approach uses a similar Dolev-Yao style CSP model to those compiled by Casper, but rather than checking the state space with FDR, Schneider proves authentication and secrecy properties for the processes. In order to prove these properties Schneider assigns a rank to each message (and signal) that can be generated by the honest agents and the intruder.

In Schneider's formalism the honest agents perform signal events at key points in the protocol. The signal $initdone.a.b.n_a$ event is performed after a protocol run by the initiator a, apparently with responder b, using nonce n_a; the signal $respgo.b.a.n_a$ is performed by the responder b during a protocol run apparently initiated by a using nonce n_a. Schneider specifies non-injective agreement on the nonce n_a by requiring that any occurrence

of $initdone.a.b.n_a$ in any trace of the network must be preceded by some occurrence of $respgo.b.a.n_a$ (written $initdone.a.b.n_a$ **precedes** $respgo.b.a.n_a$).

Schneider shows that proving $respgo.b.a.n_a$ **precedes** $initdone.a.b.n_a$ is equivalent to proving that when the honest agent b is prevented from performing the *respgo* event, the honest agent a cannot perform the *initdone* event. In order to prove that this is the case Schneider finds a rank function ρ that assigns a rank to all the messages and signals that can occur in the system, such that ρ assigns a non-positive rank to the *initdone* event. By proving that only messages and signals with a strictly positive rank can occur in the restricted system Schneider proves that the *initdone* event cannot occur when the *respgo* event is blocked.

Theorem 2.7.1 (Rank function theorem)**.** *If ρ (a rank function on the messages and signals in the system) is such that:*

1. $\forall m \in IIK \cdot \rho(m) > 0$;
2. $\forall S \subseteq Message \cdot (\rho(S) > 0 \wedge S \vdash m) \Rightarrow \rho(m) > 0$;
3. $\rho(b) \leqslant 0$;
4. $\forall i \cdot (USER_i \underset{\{a\}}{||} STOP)$ **sat** $\rho(tr \upharpoonright rec) > 0 \Rightarrow \rho(tr) > 0$;

then $(|||_i\ USER_i) \underset{\{|trans,rec|\}}{||} ENEMY(IIK)$ **sat** a precedes b.

The rank function theorem is the fundamental underpinning of Schneider's approach. Condition 1 in the theorem states that all the messages the intruder initially knows have positive rank; Condition 2 states that the intruder cannot introduce messages of non-positive rank if he only hears messages of positive rank. These two conditions can be proved independently, and do not depend on the protocol under consideration. Condition 3 states that ρ assigns a non-positive rank to the message (or signal) we wish to show cannot happen. Condition 4 is entirely dependent upon the protocol in question, and so it is this condition that must be proved when a new protocol is analysed; the condition says that the honest agents cannot introduce messages of non-positive rank if they only receive messages of positive rank.

Proving that Condition 4 holds involves a case analysis of each stage of the protocol for each agent. When trying to prove this one obtains constraints for the rank function ρ. If the proof of Condition 4 fails (i.e. if contradictory constraints on ρ are found), then one can conclude that there can be no possible rank function for the protocol; in this case, one can often find an attack against the protocol.

Finding the constraints by trying to prove that Condition 4 holds is the most illustrative way to discover a suitable rank function. However, Heather

and Schneider have developed a decision procedure for finding a suitable rank function if one exists, or for showing that a rank function does not exist (which, almost invariably, implies the existence of an attack against the protocol) [HS05]. The decision procedure works by partitioning the infinite message space into a finite number of equivalence classes, and repeatedly applying the protocol steps and message generation rules (the $\vdash$ relation) to the equivalence classes, starting from the intruder's initial knowledge. The RankAnalyser tool [Hea05] generates rank functions automatically by following this decision procedure.

Schneider and others have developed extensions to the rank function approach in order to model more general security protocols and to prove more general properties. Alternative cryptographic algorithms (for example Vernam encryption) can be incorporated into the model by specifying their algebraic properties [Sch02]. Temporal rank functions are a generalisation of rank functions, and have a more general rank function theorem; they allow one to analyse secrecy properties which involve messages only needing to have positive rank during a certain period of time [DS04].

Dutertre and Schneider have used the PVS theorem prover [COR+95] to provide automated support to the rank function method [DS97]. Using a semantics embedding of CSP they specify the network model in PVS, and then use specialised PVS rewrite rules and proof commands to conduct the proofs of authentication theorems using user-specified rank functions.

2.7.2 BAN logic

Burrows, Abadi and Needham developed the BAN logic to prove that protocols achieve specified authentication goals, to discover the assumptions they rely upon, and to discover whether they do anything unnecessary [BAN90]. The logic was not designed to reason about secrecy, or authentication of untrusted principals. The logic defines several constructs to reason about the facts that agents believe, the facts that they say and see (i.e. that other agents send to them), and the facts that they control (i.e. those that other agents trust them to produce). The logic specifies several rules for evolution of belief; for example, if P believes that k is a key that P and Q can use for communication, and P sees X encrypted with k (from some agent other than P), then P should believe that Q said X.

The first step in analysing a protocol with the BAN logic is to convert the protocol messages into formulae in the logic; this is referred to as idealisation. For example, the message $A \rightarrow B : \{A, K_{AB}\}_{K_{BS}}$ may tell B (who knows K_{BS}) that K_{AB} is a good key to use to communicate with A. In this case, the idealised message would be the encryption with K_{BS} of the statement in the logic that K is a good key for A and B to communicate with: $A \rightarrow B : \{A \overset{K_{AB}}{\longleftrightarrow} B\}_{K_{BS}}$. Once the protocol has been idealised, the next step is to annotate it with formulae about the agents' belief before the

first message, and after every message. The beliefs before the first message are usually fairly standard (e.g. the shared keys that agents possess, fresh nonces that agents will introduce, and assumptions about trust between agents). The final step in a protocol analysis is to prove that certain conclusions hold at the end of the protocol run. The authors do not define what they mean by authentication, but they give some examples of conclusions that ought to hold after a protocol run; e.g.:

$$A \textbf{ believes } A \overset{k}{\longleftrightarrow} B \quad \wedge \quad B \textbf{ believes } A \overset{k}{\longleftrightarrow} B\,.$$

The BAN logic can be very useful for finding weaknesses in protocols, or for discovering redundancies in the protocol messages. However, the assumption that all parties are honest is one of the major weaknesses of the approach. In [BAN90] the authors prove that after a run of the original Needham-Schroeder public key authentication protocol each agent believes that the nonces are shared secrets (that no-one else could know). However, we know that this is not the case when an active intruder can disrupt the messages the agents send.

2.7.3 NRL Protocol Analyzer

The NRL Protocol Analyzer [Mea94, Mea96b] is a protocol verification tool, written in Prolog, that models protocols as interactions between a set of state machines. The Analyzer uses proof techniques for state machines and for formal languages, coupled with an exhaustive backwards search to prove that user specified insecure states are unreachable. The NRL Protocol Analyzer is effective at finding flaws in security protocols and at proving that protocols satisfy security guarantees (specified as unreachability results for insecure states).

The NRL Protocol Analyzer is based on the Dolev-Yao term rewriting system [DY83]. The intruder's goal is to manipulate the algebraic system (the protocol) to learn words that he does not know (e.g. encryption keys) or to convince a principal that a certain word has properties that it does not. To prove that a protocol is secure one first specifies the intruder's goals, and then proves that they are impossible. It is often necessary to prove reachability results: for example, a key establishment algorithm should be able to reach a state in which the two users running the protocol share a key. To specify states such as this the state machines store local state variables to represent the knowledge of the principals.

In order to use the Analyzer one must specify the protocol under consideration in terms of the words in use (user names, nonces, cryptographic keys, etc.), the intruder's initial knowledge, and the behaviour of the state machines that represent the users running the protocol. Once a protocol has been specified in this way the user interacts with the Analyzer by inputting a state; the Analyzer then returns a description of the states that

may immediately precede it. The Analyzer uses rules to discard from the output states that are unreachable; there are some standard rules (such as the rule to ignore a state where the intruder learns a word before it has been generated), but the user can also specify rules to discard states that he knows to be unreachable.

The Analyzer also uses formal languages to limit the search space. One of the most common causes of infinite loops in the Analyzer is searching for unbounded sets of words to determine whether the intruder can learn a particular word. The user can specify context-free formal languages, and the Analyzer can then prove whether or not the language is reachable. If during a protocol analysis the Analyzer must discover whether the intruder can learn a word belonging to an unreachable language then the search can be terminated before entering an infinite loop.

The Analyzer's backwards search can proceed in two ways. In automatic mode the Analyzer uses unreachability results entered by the user and certain heuristics to narrow the search, but fully explores the pre-image of the returned states. In manual mode the user chooses subsets of the states for the Analyzer to explore. A full protocol analysis is usually conducted using both modes; if an automatic search produces a search tree that is too bushy, the user can switch to manual mode, undo part of the tree, and query it manually before switching back to automatic mode.

The NRL Protocol Analyzer has successfully been used to find new attacks against protocols [Mea92] such as the Simmons Selective Broadcast Protocol [Sim85], and to reproduce well-known attacks [Mea96a] such as Lowe's attack against the Needham-Schroeder Public-Key protocol. The Analyzer has also been used to find weaknesses or ambiguities in protocol specifications (e.g. the original Internet Key Exchange protocol) [Mea99].

2.7.4 Inductive analysis

Paulson's inductive approach [Pau98] combines features from model checkers such as Casper and the NRL Analyzer (it has a concrete notion of events) and from belief logics such as BAN logic (guarantees are derived from each message in the protocol).

In the inductive approach protocols are specified as a set of rules by which honest agents may extend the current trace of events; these rules define the set of all possible traces inductively. Events are either of the form Says $A\ B\ X$ when agent A sends the message X to B, or Notes $A\ X$ when agent A stores the message X, though the model could be extended to allow other kinds of events (such as an agent gaining a new long term key). Any agent may respond to any prior event any number of times (but the honest agents only respond to events addressed to them), so in this model agents may participate in many runs concurrently.

Theorems are proved by induction over the set of possible traces; in

order to prove a theorem P one must prove that P holds for the empty trace ($P([])$), and that whenever P holds for a trace evs it also holds for any extension of that trace $ev\#evs$ ($P(evs) \Rightarrow P(ev\#evs)$). In contrast with the short proofs one can obtain in a belief logic, the inductive proofs are often too long and detailed to be generated by hand; Paulson uses the theorem prover Isabelle [Pau94] to generate proofs.

The intruder in Paulson's model is specified independently of the protocol, and so most of the results about the intruder can be reused between analyses. Paulson has a common body of rules for all protocols (e.g. there is a rule governing what the intruder can say). As in the CSP model, the intruder hears all network traffic and can send messages that he can deduce from what he overhears. The intruder has control over an unspecified set of compromised agents.

In order to verify a protocol one must first prove possibility theorems (e.g. that the message formats agree, and the protocol can successfully be completed). The proof of these theorems does not guarantee liveness properties of the protocol; just as in the CSP model, only safety properties can be considered. The proof of theorems will often involve simplification by the use of forwarding lemmas, rewrite rules and symbolic evaluation of the sets of messages that the intruder can learn and send. After proving possibility theorems one can prove secrecy theorems: these pertain to the set of messages the intruder can obtain by analysis of the messages sent in traces (e.g. if the intruder holds some session keys he cannot use them to reveal others). Finally one must prove authenticity theorems for all parties involved in the protocol; for example, it can be shown that the initiator in the original Needham-Schroeder public key authentication protocol receives a guarantee that the responder was running the protocol with him, but an equivalent theorem cannot be proved for the responder (due to the attack discovered by Lowe).

The inductive approach is good at proving protocols correct (i.e. proving that they satisfy authenticity or secrecy theorems) and also at finding flaws in protocols (these are usually discovered when the proof of a theorem fails). Paulson has used his approach to prove authentication and secrecy results for the TLS handshake protocol [Pau99]; Paulson and Bella extended the method to include timestamps, and to prove authentication and secrecy properties for Kerberos [BP97]; and Paulson et al. have used the inductive method to verify the Purchase protocol of the Secure Electronic Transaction suite of protocols [BMP06]. In Chapter 4 we describe how Bella et al. adapted the inductive approach to prove properties about *second-level security protocols*: security protocols that rely on an underlying security protocol in order to achieve their goals.

2.7.5 Strand spaces

The strand space approach to proving security protocols correct was developed by Guttman et al. at MITRE in 1998 [FHG99b]. A strand space is a set of strands, each of which is a sequence of events that a single principal may engage in. A strand is therefore a sequence of message transmissions and receptions, with values of keys, nonces, and other data items given explicitly. Each legitimate strand represents one run in a protocol by one party.

Guttman et al. define several penetrator strands: sequences of message transmissions and receptions that model basic capabilities of the intruder (e.g. receiving a key and a message encrypted with that key, and sending the decrypted message; or receiving two messages and sending their concatenation). This explicit model of the possible behaviours of the intruder allows the authors to develop general theories that bound his abilities; these theories can be applied to any protocol under consideration.

In the strand spaces approach the standard Dolev-Yao assumption of free encryption is made. Strand spaces are not full (i.e. do not contain all possible strands); fresh values (such as nonces, encryption keys) are established by including only one originating strand for each item. Unguessable values are modelled by ensuring that no penetrator strands contain the value unless the intruder previously received it.

A partial order, consisting of two kinds of edges, is introduced on the space:

- Successive nodes of a strand are connected; this is a causal ordering of the actions under the control of the principal in question. For honest agents this corresponds to the order in which they send and receive messages; for the intruder it represents causal constraints on his behaviour;
- Message transmissions and subsequent receptions are linked.

A bundle is a portion of a strand space: it is a causally connected set of nodes. Bundles must be well founded: there must be a unique transmission node for each reception node (however there might be transmission nodes without a corresponding reception node, or with more than one reception node). For any strand corresponding to a given run of a principal one constructs all possible bundles containing nodes of the strand; this set of bundles encodes all possible interactions of the environment with that principal in the given run of the protocol. Typically for a protocol to be correct each bundle must contain one strand for each legitimate principal apparently participating, all of which must agree on all of the data items in use in that run.

To prove non-injective agreement (as in [Low97]) one must show that whenever a bundle contains a strand representing a responder run, then it

also contains a strand representing an initiator run with the same data values. To prove (injective) agreement, one must show that the initiator strand is unique. A value x is secret in a bundle if regular strands (i.e. strands representing honest agents' runs) never emit it, and the intruder can never emit it.

A proof of a security protocol in the strand space approach can give a detailed insight into the reasons why the protocol is correct, and the assumptions that are necessary for it to be correct. The proofs are simple and informative, and are easily developed by hand; they identify exact conditions under which a protocol can be relied upon.

Guttman et al. have used strand spaces to prove several general properties about security protocols. In [FHG99a] they define multi-protocol *mixed strand spaces*, and show how one can use these to prove that a particular security protocol can remain correct even when used in combination with a range of other protocols; in [GF00b] they use mixed strand spaces to prove a more general result: two protocols are independent (i.e. they both remain correct if they are used together) if they use encryption in non-overlapping ways (disjoint encryption). In [GF00a] Guttman and Fábrega define the notion of *authentication tests* to simplify the proofs of authentication properties. An authentication test is an inference that a regular protocol participant may make when they transmit a message containing a new value v, and later receive v in a cryptographically altered form. The participant may conclude that some principal possessing the relevant key has transformed the message containing v; in certain cases, this must be a regular participant, not the penetrator.

In [GF05] Guttman and Fábrega define the notion of *skeletons*: a skeleton describes the shape of the regular nodes (the events performed by honest agents) in a bundle (a causally well-founded subgraph of a collection of nodes; it expresses a potential execution of a protocol). They then use the notion of skeletons, and operations on them to show that a protocol execution involving many runs of the protocol roles, but only a small number of nonces, can be collapsed to a smaller execution, and hence to show that if there is a counterexample to a formula expressed in a first order language, there is a counterexample using a limited number of runs of the protocol roles.

Several other authors have used strand spaces to prove general properties of security protocols. For example, Heather et al. show that tagging messages with an indication of their intended type prevents type-flaw attacks [HLS03]; Malladi et al. show that tagging all encrypted components in protocol messages with a session identifier and a component number can prevent replay attacks against protocols [MAFH02].

2.7.6 Spi calculus

The spi calculus was developed by Abadi and Gordon [AG99] to model cryptographic security protocols in the pi calculus [Mil99]. The pi calculus is a programming language for describing systems of independent parallel processes. Processes in the pi calculus communicate by passing messages on named channels. In contrast to CSP, where a process can only communicate on a set of channels that is determined before computation, processes in the pi calculus may only communicate on channels they are in the scope of, and the scope of channels may evolve during computation. When a process sends a restricted channel as a message to a process outside that channel's scope, the scope is *extruded* to embrace the receiving process. In the spi calculus scope restriction and extrusion are used as the formal model of possession and communication of secrets.

Security guarantees are expressed as equivalences between processes in the eyes of an arbitrary environment. A protocol keeps X secret if the protocol when run with X is indistinguishable from the protocol when run with X' for any X'. Because the environment is arbitrary, it is not necessary to model the intruder explicitly. The scoping rules of the pi calculus guarantee that the intruder (i.e. the environment) cannot access channels that are not explicitly given.

Authentication guarantees are proven by checking that the process running the protocol is equivalent to a 'magical' process that guarantees that receivers receive the correct message (so the intruder cannot force them to receive a different message from that sent by the sender). Both of these equivalence checks are based on an equivalence relation defined as follows:

For any closed process P (i.e. a process with no free variables), P exhibits β (written $P \Downarrow \beta$) if, eventually, P will input or output on β. A *test* on P is a pair (R, β) where R is another process (an arbitrary environment) and β is a channel name; P passes the test if and only if the parallel composition $P|R$ exhibits β. Then:

$$P \sqsubseteq Q \mathrel{\widehat{=}} \text{for any test } (R, \beta) \cdot (P|R) \Downarrow \beta \Rightarrow (Q|R) \Downarrow \beta$$
$$P \simeq Q \mathrel{\widehat{=}} P \sqsubseteq Q \text{ and } Q \sqsubseteq P$$

This form of testing equivalence corresponds to partial, or safety, correctness.

All of the process communication and secrets could be modelled in the pi calculus, but it is the extensions given in the spi calculus that make this approach useful. The authors have given formal semantics for public and shared key encryption, private key message signing and hashing. The spi calculus model of cryptography makes the standard Dolev-Yao assumptions: an encrypted message can only be decrypted with the corresponding key, encrypted messages do not reveal the key used for the encryption and there is sufficient redundancy in encrypted messages so that decryption can be verified.

In Chapter 4 we describe Bugliesi and Focardi's extension to the spi and pi calculi to model secure channels by the properties that they provide to protocols that use them [BF08].

2.7.7 SAT-based model checking

Armando and Compagna present a model checking technique that, given a protocol description in a multi-set rewriting formalism and an integer $k > 0$, builds a propositional formula whose models correspond to attacks against the protocol that can be carried out in fewer than k steps [AC08]. Finding these models, if they exist, and hence finding attacks against protocols is solved by checking the propositional formula for satisfiability, and this can be done efficiently with a SAT solver.

The multi-set rewriting formalist of [AC08] uses facts to describe the states of the honest agents and the intruder, and rewrite rules to model the behaviour of the honest agents and the intruder. The facts include $\texttt{ik}(m)$, the intruder knows message m; $\texttt{msg}(j, s, r, m)$, agent s sent message m to agent r at protocol step j; and $\texttt{state}(j, s, r, ms, c)$, agent r is ready to execute stage j of the protocol in session c, and he knows the messages in ms. The protocol is described by the rewrite rules for the honest agents.

The intruder in [AC08] is a Dolev-Yao style active attacker whose capabilities (to overhear messages, deduce new messages and fake messages) are modelled by rewrite rules. For example, the following rewrite rule models the intruder faking a send of message m (which is message j of the protocol) from agent a to agent b:

$$\texttt{ik}(m).\texttt{ik}(a).\texttt{ik}(b) \xrightarrow{\texttt{fake}_j(a,b,m)} \texttt{ik}(m).\texttt{ik}(a).\texttt{ik}(b).\texttt{msg}(j, a, b, m)\,.$$

This rule says that a set containing the facts $\texttt{ik}(m)$, $\texttt{ik}(a)$ and $\texttt{ik}(b)$ can be rewritten to a set containing these three facts, and the fact $\texttt{msg}(j, a, b, m)$.

Security properties are specified in [AC08] by specifying bad states: states in which the intruder can learn a message that the honest agents believe is a secret, or states in which the intruder can disrupt the authentication goals of the protocol. The authentication properties from [Low97] are specified in [AC08] in terms of facts that represent the initiator beginning a protocol run with value v of data item d ($\texttt{witness}(a, b, d, v)$), and the responder accepting v at the end of a protocol run ($\texttt{wrequest}(b, a, d, v)$). A non-injective agreement specification is tested by searching for a state in which $\texttt{wrequest}(b, a, d, v)$ holds, but the corresponding `witness` fact does not hold.

Armando and Compagna's SAT-based Model Checker (SATMC) starts with a protocol insecurity problem and a protocol description (as described above) and applies optimising techniques to simplify the problem. It then generates a propositional formula that represents all possible attacks on the

protocol of length at most i for all values of i, starting at 0, and stopping when it reaches k (the bound specified by the user). The model checker passes each of these propositional formulae to a SAT solver to check for satisfiability. If the SAT solver finds that the formula corresponding to length i is satisfiable, then an attack of length i exists against the protocol. The model checker may discover that none of the formulae of length less than k are satisfiable; in this case one may conclude that no attacks against the protocol of length less than k are possible.

In Chapter 4 we describe a model for analysing security protocols that rely on other security protocols (i.e. secure transport layers) in the formalism of [AC08].

2.7.8 ProVerif

Blanchet's ProVerif [Bla01, Bla02] is an automatic protocol verifier that works by representing protocols as Horn clauses (Prolog rules). ProVerif constructs an over-approximation of the protocol, and then checks whether or not a fact can be determined from the protocol rules. The over-approximation means that although the verifier always terminates, it may sometimes give false attacks. Blanchet has built an automatic translator from a restricted version of the pi calculus to the Prolog rules that define the protocol and the intruder.

The Prolog rules that represent a protocol are built from terms (which represent messages exchanged during a run of the protocol) and facts (which represent facts about these messages). The protocol messages are built from names and function applications; names are used to represent atomic values (such as keys and nonces), and functions are used to build new terms (e.g. by pairing messages, by encrypting messages, or by applying a hash function to a message). The most important fact is the predicate $\texttt{attacker}(m)$: the attacker knows the message m. A rule of the form $F_1 \wedge \ldots \wedge F_n \rightarrow F$ means that if all the facts $F_1, \ldots, F_n$ are true then F is also true.

Cryptographic primitives are represented in ProVerif by two kinds of functions: constructors appear in messages, and are used to build new terms (e.g. the constructor $\mathsf{sencrypt}(m, k)$ represents the encryption of message m with the symmetric key k); destructors are used to manipulate terms, and are defined by equations of the form $g(m_1, \ldots, m_n) = m$ (e.g. the destructor $\mathsf{sdecrypt}(\mathsf{sencrypt}(m, k), k) = m$ represents decryption with the symmetric key k).

Blanchet models a Dolev-Yao style attacker by encoding his abilities in Prolog rules. For example, if f is a constructor of arity n then this leads to the rule $\mathsf{attacker}(x_1) \wedge \ldots \wedge \mathsf{attacker}(x_n) \rightarrow \mathsf{attacker}(f(x_1, \ldots, x_n))$; if g is a destructor defined by $g(m_1, \ldots, m_n) = m$ then this leads to the rule $\mathsf{attacker}(m_1) \wedge \ldots \wedge \mathsf{attacker}(m_n) \rightarrow \mathsf{attacker}(m)$.

The protocol itself is represented by a series of sets of rules, one for each

step of the protocol. If agent a sends message i in the protocol then the ith set of rules contains rules that have as hypotheses the patterns of messages received by a earlier in the protocol, and the pattern of the ith message as the conclusion. This creates an over-approximation of the protocol; in particular, freshness (e.g. of nonces) is modelled by letting new names be functions of messages previously received in the protocol, so different values are used per pair of agents rather than per session, and each step of the protocol can be completed several times as long as the previous steps have been completed at least once. This over-approximation means that ProVerif can be more efficient, and can verify the protocol without limiting the number of runs of the protocol. However, it means that sometimes the verifier returns false attacks: sequences of rule applications that do not correspond to a real protocol run. The verifier can sometimes fail to terminate, but Blanchet claims that it always terminates for a large class of 'well-designed protocols' [Bla02].

ProVerif verifies secrecy properties by checking whether the attacker can learn a particular value from observing runs of the protocol. In other words, ProVerif checks whether $\mathsf{attacker}(m)$ can be derived from the rules that define the protocol and the intruder. This is exactly the problem that is usually solved by Prolog systems; however, Blanchet defines a new guided search algorithm to solve this problem because other Prolog systems do not terminate on this problem (there are rules for the attacker that lead to considering more complex terms with an unbounded number of constructor applications). ProVerif's search algorithm is based on unification: when the consequence of one rule unifies with one hypothesis of another (or the same) rule a new rule can be created that corresponds to applying the first rule and then the second one.

ProVerif can prove authentication protocols such as those defined by Lowe [Low97]; these properties are specified by saying that one event cannot happen unless another event happened first. In the case of non-injective agreement the property is that the recipient's end event cannot happen unless the initiator's begin event happened earlier. These properties are verified by determining whether a particular end event (corresponding to the exact authentication property that is being checked) can be derived from the set of rules that define the protocol and the intruder. If the end event is derivable from the rules only when the corresponding begin event is contained in the rules, then the protocol satisfies the authentication property.

Allamigeon and Blanchet have implemented an attack reconstruction tool for ProVerif [AB05]. When ProVerif cannot prove a property the tool tries to reconstruct an attack trace of the protocol by exploring a finite set of traces; this exploration is guided by the set of rules that were used to derive $\mathsf{attacker}(m)$.

2.7.9 Other model checkers

In this section we have described several approaches to model checking security protocols (Casper/FDR, the NRL Protocol Analyzer, SATMC, ProVerif); however, there are many other systems for model checking security protocols.

Murφ is a description language and a verifier for finite state concurrent systems [DDHY92]; it has been used to verify security protocols by Mitchell et al. [MMS97]. To verify a protocol with Murφ one must model it in the Murφ description language, and extend the model with descriptions of the desired properties of the protocol; Murφ then checks if all the reachable states of the model satisfy the given specification. If this is not the case it will terminate with a trace leading to a state that does not satisfy the specification.

Brutus [CJM00] is a model checker for verifying security protocols. A security protocol is modelled in Brutus by specifying the messages that it consists of (these are messages in a message space with a deduction relation identical to $\vdash$), and then building processes for each of the honest principals in the protocol. These processes describe the message flow specified by the protocol (i.e. the send and receive events), as well as special signal events; these are used to mark when the agent begins and ends a run of the protocol. Finally one specifies the properties that the security protocol should satisfy in a first-order logic with a past-time operator. Brutus can then be used to check the protocol satisfies the required properties, and if this is not the case, Brutus will return a counterexample (i.e. an error trace).

There are also alternatives to the abstract layered approach proposed by Broadfoot and Lowe. Hansen et al. use static analysis to verify Version 1.1 of the SAML single sign-on protocol [HSN05]. In order to conduct their analysis they create explicit models of unilateral and bilateral TLS in a variant of the pi calculus, and then build the model of the SAML protocol on top. Hansen et al. actually model a slightly stronger protocol than TLS because the analysis of a 'true' model of TLS threw up a false error, due to an inability to tie the premaster secret to the verification of the client's certificate in a message sent from the client to the server. In Chapter 4 we discuss several alternative approaches to specifying secure channels, and we compare the properties that are specified in these other approaches to the properties described in this thesis.

Chapter 3

Specifying secure channels

In this chapter we present our hierarchy of authenticated and confidential channels. We first formalise an abstract model of a layered network, and relate it to a concrete network. We describe the sets of valid traces that our abstract network accepts, and we provide a framework for specifying secure channels. A secure channel specification is a trace-based specification that restricts the set of valid traces.

In Section 3.2 we describe how we mark confidential channels in a system, and define the properties of a confidential channel in terms of the relation between the intruder's knowledge in our abstract model (which is based on removing his ability to listen on confidential channels) and the intruder's knowledge in the concrete model (which is based on deducing everything he can from everything sent on the network).

In Section 3.3 we define the building blocks we use to create our hierarchy. These building blocks progressively disallow different aspects of the intruder's behaviour, and can be combined to create different channels. Not all combinations are different: in many cases, several different compositions of the building blocks allow essentially the same behaviour (they simulate one another); we collapse such cases, and reach a hierarchy of eleven secure channels. In Section 3.4 we consider several of the secure channels from the hierarchy in more detail, and relate them to real-world secure transport protocols.

In Section 3.5 we consider channel specifications that tie different messages into a single connection. We specify a *session* property that binds messages into a single session, and a stronger *stream* property that not only ensures that messages are not moved from one session to another, but also guarantees that the order messages are received in is the same as that in which they were sent.

Finally, in Section 3.6 we conclude and summarise our findings.

3.1 Channels, formally

In this section we formalise our model of an abstract network and its relation to a concrete network. The abstract network is defined in terms of honest agents, who send and receive messages, and an intruder, who has several events he can use to manipulate the messages being passed on the network, and who can also send and receive messages.

The channel specifications described in this thesis are based on traces of CSP-style processes. This approach allows us to capture the channel properties in a fairly intuitive way (by talking about the events that the intruder can perform, and the events that must precede another event), however, we do not believe that these ideas are restricted to the world of CSP. The channel properties and the means for comparing them could be described in several other formalisms (e.g. spi-calculus, propositional logic), and the results obtained could be directly compared with the results presented in this thesis. Indeed, Kamil and Lowe have translated many of our channel properties into the strand spaces formalism [KL08].

Our model reflects the traditional internet protocol stack, but we add a new layer between the transport layer and the application layer: the secure transport layer. The secure transport layer relies on the services of the transport layer (to deliver messages from point to point), and provides authentication and confidentiality services to the application layer. We abstract all of the layers beneath the secure transport layer into a network layer; see Figure 3.1. We describe how our secure channel specifications should be interpreted in the context of our abstract network, and also in the context of a concrete network.

The most commonly used transport layer protocol is probably the Transmission Control Protocol [Ins81b]. TCP aims to provide a reliable, in-order delivery of streams of bytes over an Internet Protocol (IP) [Ins81a] network. However, because we assume that the intruder is in full control of the network process, the reliability properties that TCP aims to provide cannot be guaranteed. For security-critical internet applications (such as online banking) a TLS (or SSL) connection is usually established on top of the TCP connection. This secure transport layer protocol makes certain guarantees to the application protocol using it, and relies on the services of TCP in order to function. In our terminology a TLS or SSL connection between two agents running an application protocol is a channel.

Our model uses entities at two interfaces: between the application layer and the secure transport layer, and between the secure transport layer and the underlying network. The application layer is the layer in which agents establish channels, and send and receive messages. The secure transport layer contains *protocol agents*, which translate the higher level events into lower level events (e.g. by encrypting or signing messages), and vice versa (e.g. by decrypting messages or verifying signatures).

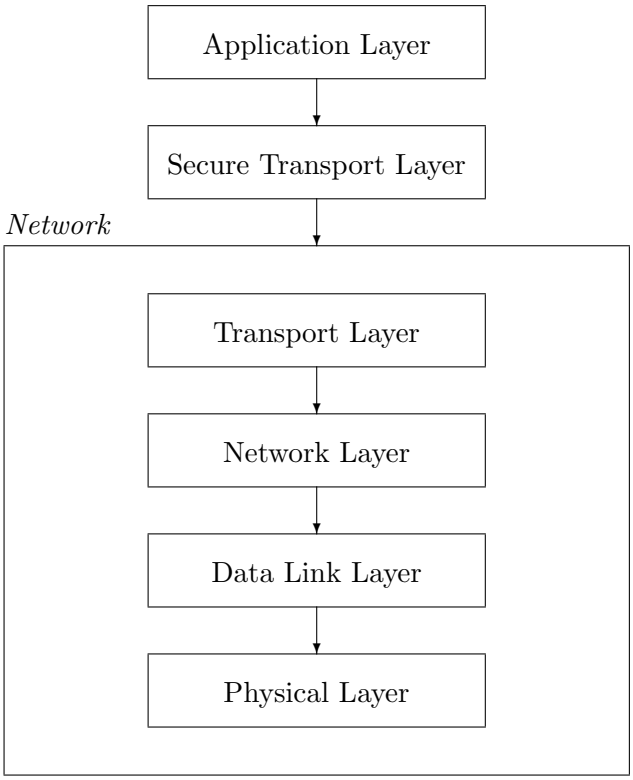


Figure 3.1: The protocol stack in our abstract network.

Most of the events are at the interface between the application layer and the secure transport layer, and describe the application layer data: these events are enough to capture authentication guarantees. The model also uses events at the interface between the secure transport layer and the underlying network, which describe the network messages: these events are necessary to capture confidentiality properties formally.

3.1.1 Describing a channel

We assume a non-empty set *Identity* of agent identities. Each identity is either considered *Honest* (i.e. the agent follows the application-layer protocols) or *Dishonest* (i.e. the agent is under the intruder's control).

We also assume a non-empty set *Role* of roles in the application-layer protocols (e.g. *Needham-Schroeder Initiator*, *Yahalom Responder*), ranged over by R_i, R_j, etc. Each role in an application protocol exchanges a series of messages with some of the other roles in the protocol. We assume that the roles used by different protocols are distinct.

Definition 3.1.1. An *Agent* is an identity taking a role:

$$Agent \mathrel{\widehat{=}} Identity \times Role\,.$$

We use A, B, etc., to range over either *Identity* or *Agent*, as convenient. We abuse notation by sometimes writing *Honest* for $Honest \times Role$, and similarly for *Dishonest*. We write $\hat{R}_i$ for $Identity \times R_i$.

A secure channel connects two agents, each playing a particular role.

Definition 3.1.2. A channel is an ordered pair of roles (R_i, R_j):

$$(R_i, R_j) \in Role \times Role\,.$$

We write $R_i \rightarrow R_j$ for the channel (R_i, R_j) as this emphasises the difference between the sending and the receiving roles; in particular it highlights the distinction between the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$.[1] Our specifications are given in terms of channels; the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ may satisfy different properties, but we will specify some symmetric properties.

We treat encryption formally, as described in Chapter 2. All messages are drawn from the message space, *Message*. This is a non-empty set of symbols which is built from basic types (such as identities, nonces and timestamps) by operations such as concatenation and encryption. We assume a relation $\vdash$ defined over this set: for $X \subseteq Message$, and $m : Message$, $X \vdash m$ means that m can be deduced from the set X. We assume that the relation $\vdash$ satisfies the following two properties:

Monotonicity $X \subseteq X' \Rightarrow \{m \mid X \vdash m\} \subseteq \{m \mid X' \vdash m\}$;

Transitivity $X \vdash m \wedge X \cup \{m\} \vdash m' \Rightarrow X \vdash m'$.

Often in our examples we use the deduction rules from [RSG+01] (shown in Chapter 2) which model Dolev-Yao style symbolic encryption: the intruder can only read messages he has the decryption keys for, and can only create encrypted (or signed) messages when he knows the requisite keys. However these rules can be modified to model commutable encryption, or guessable values [Low04], or other encryption models as required; the results in this thesis hold for any deduction relation that satisfies the two properties above.

We assume that the intruder has some initial knowledge $IIK \subseteq Message$. He may use this knowledge and messages he overhears on the network to generate new messages and facts. We restrict the intruder's behaviour so that he can only send messages that can be deduced from his initial knowledge and what he has overheard.

The message space is partitioned into two sets: application-layer messages ($Message_{App}$) and transport-layer messages ($Message_{TL}$). We assume that there are no interactions between the messages of the two layers; in particular, when we give example transport-layer protocols we assume that the

[1] In the case that $i = j$ the channels (R_i, R_j) and (R_j, R_i) are necessarily the same, and satisfy the same properties.

messages we describe could not be confused for application-layer messages. We conjecture that a property similar to the disjoint encryption property of [GF00b] is sufficient to ensure this.

The agents, including those under the intruder's control, communicate in sessions, distinguished locally by connection identifiers. A connection identifier can be thought of as a *handle* to the communication channel: when the protocol agent creates a new channel, a connection identifier is returned, which the agent uses for all communication over that channel.

We use the following events, where m ranges over the set $Message_{App}$ of application-layer messages.

send$.(A, R_i).c_A.(B, R_j).m$**:** the agent (A, R_i) sends message m, intended for agent (B, R_j), in a connection identified by A as c_A.

receive$.(B, R_j).c_B.(A, R_i).m$**:** the agent (B, R_j) receives message m, apparently from agent (A, R_i), in a connection identified by B as c_B.

fake$.(A, R_i).(B, R_j).c_B.m$**:** the intruder fakes a *send* of message m to agent (B, R_j) in connection c_B; the intruder fakes the message with the identity of honest agent (A, R_i); he may be injecting the message into a pre-existing connnection, or causing B to start a new one. In order to fake a message, the intruder must be able to choose the message from those he knows.

hijack$.(A, R_i).(A', R_i).(B, R_j).(B', R_j).c_{B'}.m$**:** the intruder modifies a previously sent message m and changes the sender from (A, R_i) to (A', R_i), and the receiver from (B, R_j) to (B', R_j) so that B' accepts it in connection $c_{B'}$; we write this event as $hijack.(A, R_i) \rightarrow (A', R_i).(B, R_j) \rightarrow (B', R_j).c_{B'}.m$ to highlight its intent.

The *hijack* event can be used by the intruder in four different ways:

- To *replay* a previously-sent message: the intruder either chooses an existing connection or initialises a new one, and causes the recipient to receive the message in that connection; we abbreviate the event and write $hijack.(A, R_i).(B, R_j).c_B.m$;
- To *re-ascribe*[2] a message: the intruder changes the sender's identity and chooses a connection for the recipient to receive it in; we abbreviate the event and write $hijack.(A, R_i) \rightarrow (A', R_i).(B, R_j).c_B.m$;[3]

[2]To ascribe means to attribute a text to a particular person; hence we use "re-ascribe" to describe the intruder's activity when he changes the identity of the sender of a message.

[3]The arrow notation resolves any possible ambiguity that might arise from abbreviating the event in this manner.

- To *redirect* a message: the intruder changes the identity of the recipient and chooses a connection for the new recipient to receive it in; we abbreviate the event and write $\mathit{hijack}.(A, R_i).(B, R_j) \rightarrow (B', R_j).c_{B'}.m$;
- To *re-ascribe and redirect* a message: the intruder changes both identities and chooses a connection for the new recipient to receive the message in.

For example, if application layer message m from A to B is encoded as the transport layer message:

$$A \rightarrow B : A, \{m\}_{PK(B)} ,$$

where $PK(B)$ is B's public key, then a dishonest agent may re-ascribe this message, replacing the identity A with an arbitrary other identity. On the other hand, if m is encoded as:

$$A \rightarrow B : \{\{m\}_{PK(B)}\}_{SK(A)} ,$$

where $SK(A)$ is A's secret key, then the intruder can only re-ascribe it by replacing the signature with his own: he can only do so with a dishonest identity. Recall that the intruder can only fake messages that he knows, so in both the above cases, the intruder could not have used a *fake* event, except if he happened to know m.

Likewise, if m is encoded as:

$$A \rightarrow B : B, \{m\}_{SK(A)} ,$$

then a dishonest agent may redirect this message, replacing the identity B with an arbitrary other identity. On the other hand, if m is encoded as:

$$A \rightarrow B : \{\{m\}_{SK(A)}\}_{PK(B)} ,$$

then the intruder can redirect it only if he possesses $SK(B)$: he can only redirect messages sent to him. Note that the intruder could not have used a *fake* event, because he cannot choose the value of m.

In Section 3.1.3 we show how these abstract events relate to the concrete events in the transport layer; there is a picture of the events in Figure 3.2. Under some circumstances the dishonest events may collapse (i.e. the behaviour of the intruder with one event can be simulated with another); we explore these circumstances later in this chapter.

3.1.2 An abstract network

We now specify four rules that define the application-layer behaviour accepted by our networks. We are not yet trying to capture channel properties; rather, we are defining some sanity conditions in order to remove artificial and irrelevant behaviour from our networks.

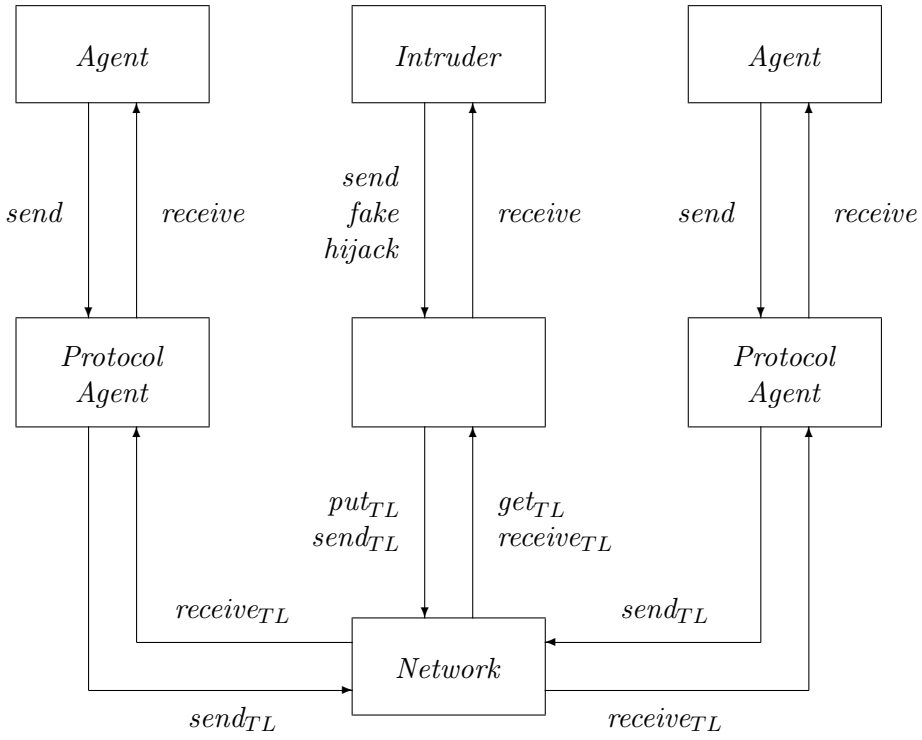


Figure 3.2: The concrete and abstract levels of the network.

1. The intruder never sends or fakes messages to himself, and never fakes messages with a dishonest identity (as he can perform a send).

$$\begin{aligned}&\mathcal{N}_1(tr) \mathrel{\widehat=} \\ &\quad tr \downarrow \{\mid send.Dishonest.Connection.Dishonest, \\ &\qquad\qquad fake.Agent.Dishonest, fake.Dishonest \mid\} = \langle\rangle\,.\end{aligned}$$

2. The intruder can only hijack messages that were previously sent (not faked).

$$\begin{aligned}&\mathcal{N}_2(tr) \mathrel{\widehat=} \\ &\quad \forall A, A', B, B' : Agent; c_{B'} : Connection; m : Message_{App} \cdot \\ &\qquad hijack.A \rightarrow A'.B \rightarrow B'.c_{B'}.m \textbf{ in } tr \Rightarrow \\ &\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr\,.\end{aligned}$$

3. In order to define the intruder's capabilities, we require a means to describe exactly what the intruder knows. In the next section we define a function:

$$IntruderKnows_{IIK} : Trace \rightarrow \mathbb{P}(Message_{App})$$

such that $IntruderKnows_{IIK}(tr)$ gives the set of messages that the intruder knows (and so can send) after tr, assuming that his initial knowledge is IIK. We limit the intruder's actions in the application layer: he can only send or fake messages that he knows.

$$\begin{aligned}&\mathcal{N}_{3,IIK}(tr) \mathrel{\widehat{=}} \\ &\quad \forall I : Dishonest; c_I : Connection; B : Honest; tr' : Trace; m : Message_{App} \cdot \\ &\qquad tr' \frown \langle send.I.c_I.B.m \rangle \leqslant tr \Rightarrow m \in IntruderKnows_{IIK}(tr') \wedge \\ &\quad \forall A, B : Honest; c_B : Connection; tr' : Trace; m : Message_{App} \cdot \\ &\qquad tr' \frown \langle fake.A.B.c_B.m \rangle \leqslant tr \Rightarrow m \in IntruderKnows_{IIK}(tr') \,.\end{aligned}$$

4. No agent may receive a message that was not previously sent, faked or hijacked to them.

$$\begin{aligned}&\mathcal{N}_4(tr) \mathrel{\widehat{=}} \\ &\quad \forall B : Honest; c_B : Connection; A : Agent; m : Message_{App} \cdot \\ &\qquad receive.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \exists A', B' : Agent; c_A : Connection \cdot \\ &\qquad\quad send.A.c_A.B.m \ \mathbf{in}\ tr \vee \\ &\qquad\quad fake.A.B.c_B.m \ \mathbf{in}\ tr \vee \\ &\qquad\quad hijack.A' \rightarrow A.B' \rightarrow B.c_B.m \ \mathbf{in}\ tr \,.\end{aligned}$$

These four rules give the basic properties that any network must satisfy. The first three place basic restrictions on the intruder's activity, and are intended to represent a realistic view of the capabilities of a Dolev-Yao style active attacker; the final rule captures the notion that a message can only be received when there is some preceding activity that might have caused it.

3.1.3 Relating the abstraction to a concrete network

We specify our channels as restrictions on the activity allowed in the application layer. In order to relate our results to a concrete model, we show how the application-layer events correspond to transport-layer events.

When an agent sends a message in the application layer (i.e. performs a *send* event), their protocol agent creates a corresponding $send_{TL}$ event in the transport layer. The network then generates a $receive_{TL}$ event for the recipient (unless the intruder hijacks the message first), which causes the recipient's protocol agent to perform a *receive* event in the application layer.

We assume the existence of a partial, symbolic decoding function $\mathcal{D}$ that transforms traces of transport-layer send and receive events on a single connection into traces of application-layer send and receive events on that connection:

$$\mathcal{D} : (Agent \times Agent) \times Connection \times Trace \nrightarrow Trace \,,$$

such that:

$$\forall A, B : \mathit{Agent}; c_A : \mathit{Connection}; tr : \mathit{Trace} \cdot \\ \mathcal{D}(A \rightarrow B)(c_A)(tr) \in \{\mid \mathit{send}.A.c_A.B, \mathit{receive}.A.c_A.B \mid\}^* .$$

The decoding function gives the trace of application-layer events that would result from undoing encryption, validating signatures and performing other functions necessary for the implementation of the secure channel in use.

There is not necessarily a 1-1 relationship between application-layer and transport-layer messages: some channels may have an initial key-establishment phase, or may send several transport-layer messages for each application-layer message, or aggregate several application-layer messages into a single transport-layer message. However, the decoding function has the following (prefix) property:

$$\forall A : \hat{R}_i, B : \hat{R}_j : \mathit{Agent}; c_A : \mathit{Connection}; tr, tr' : \mathit{Trace} \cdot \\ tr' \leqslant tr \Rightarrow \mathcal{D}(A \rightarrow B)(c_A)(tr') \leqslant \mathcal{D}(A \rightarrow B)(c_A)(tr) .$$

Consider the example channel given earlier where the application-layer message m from A to B is encoded as $A, \{m\}_{PK(B)}$. On this channel, $\mathcal{D}$ simply encrypts each message that is sent and pairs it with the sender's identity, or removes the sender's identity and decrypts each message that is received:

$$\mathcal{D}(A \rightarrow B)(c_A)(tr) \mathrel{\widehat{=}} \\ \langle f(e) \mid e \textbf{ in } tr \upharpoonright \{\mid \mathit{send}_{TL}.A.c_A.B, \mathit{receive}_{TL}.A.c_A.B \mid\} \rangle ,$$

where:

$$f(\mathit{send}_{TL}.A.c_A.B.\langle A, \{m\}_{PK(B)} \rangle) = \mathit{send}.A.c_A.B.m , \\ f(\mathit{receive}_{TL}.A.c_A.B.\langle B, \{m\}_{PK(A)} \rangle) = \mathit{receive}.B.c_B.A.m .$$

This is a very simple example, but $\mathcal{D}$ can easily be defined for more complicated secure channels such as TLS.

We specify two rules that define the formal relation between the abstract and concrete events:

1. In every honest agent's connection the decrypted stream of messages that the protocol agent sends is a prefix of the messages sent by the agent:

   $$\mathcal{A}_1(tr) \mathrel{\widehat{=}} \\ \forall A : (\mathit{Honest}, R_i); c_A : \mathit{Connection}; B : \hat{R}_j \cdot \\ (\mathcal{D}(A \rightarrow B)(c_A)(tr)) \downarrow \mathit{send}.A.c_A.B \leqslant tr \downarrow \mathit{send}.A.c_A.B .$$

   In other words, the protocol agents faithfully translate the *send* events performed by the honest agents into send_{TL} events. Furthermore, the protocol agents must pass the messages on to the network in the same order that they receive them from the agent;

2. In every honest agent's connection the stream of messages that the agent receives is a prefix of the decrypted stream of messages received by the protocol agent:

$$\begin{array}{l}\mathcal{A}_2(tr) \mathrel{\widehat{=}} \\ \quad \forall B : (\mathit{Honest}, R_j); c_B : \mathit{Connection}; A : \hat{R}_i \cdot \\ \qquad tr \downarrow \mathit{receive}.B.c_B.A \leqslant (\mathcal{D}(A \rightarrow B)(c_B)(tr)) \downarrow \mathit{receive}.B.c_B.A\,.\end{array}$$

   In other words, the protocol agents faithfully translate the $\mathit{receive}_{TL}$ events into *receive* events. Furthermore, the protocol agents must pass messages on to the agent in the same order that they receive them from the network. This rule, and the previous one, ensure that the protocol agents act honestly: they do not take messages from one connection, and transmit them in another;

The examples above show that the send_{TL} events have the same type as the *send* events (and likewise the $\mathit{receive}_{TL}$ and *receive* events). The examples also show that a get_{TL} event takes the same form as the send_{TL} event of the message being got, and the put_{TL} event takes the form of the $\mathit{receive}_{TL}$ event it will cause to be generated.

The intruder has additional capabilities: as well as performing send_{TL} or $\mathit{receive}_{TL}$ events he can add transport-layer messages to the network (put_{TL}) or remove them from it (get_{TL}). The events the intruder performs in the application layer (*send*, *receive*, *fake*, and *hijack*) define his high-level strategy; the transport-layer events define the implementation of that strategy. For example, in order to hijack a message, the intruder will *get* the transport layer message, modify it, and then *put* it back. The full trace of a re-ascribing hijack on the example channel above is shown below:

$$\begin{array}{rl}tr \mathrel{\widehat{=}} \langle & \mathit{send}.A.c_A.B.m,\ \mathit{send}_{TL}.A.c_A.B.\langle A, \{m\}_{PK(B)}\rangle, \\ & \mathit{get}.A.c_A.B.\langle A, \{m\}_{PK(B)}\rangle, \\ & \mathit{hijack}.A \rightarrow A'.B.c_B.m, \\ & \mathit{put}.B.c_B.A'.\langle A', \{m\}_{PK(B)}\rangle, \\ & \mathit{receive}_{TL}.B.c_B.A'.\langle A', \{m\}_{PK(B)}\rangle,\ \mathit{receive}.B.c_B.A'.m\,\rangle\,.\end{array}$$

We do not directly specify a formal relationship between the intruder's application-layer and transport-layer events, since the intruder is not forced to follow the protocol. In general, however, we expect a *hijack* event to be preceded by a get_{TL} event and followed by a put_{TL} event; similarly, we expect a *fake* event to be followed by a put_{TL} event.

3.1.4 Specifying channels

We specify channels by giving trace specifications. In order to prove that a particular transport layer protocol really does satisfy a channel specification

one would have to define a protocol agent, translating between application-layer and transport-layer messages, and prove that all traces of the resulting system satisfy the trace specification.

In the previous sections we formalised the relation between events in our abstract and concrete networks. The rules $\mathcal{N}_1$–$\mathcal{N}_4$, $\mathcal{A}_1$ and $\mathcal{A}_2$ define a set of traces that we refer to as *valid system traces*.

Definition 3.1.3 (Valid system traces)**.** The set of valid system traces is the (prefix-closed) set of traces composed of application-layer *send*, *receive*, *fake* and *hijack* events, and transport-layer $send_{TL}$, $receive_{TL}$, put_{TL} and get_{TL} events that satisfy properties $\mathcal{N}_1$–$\mathcal{N}_4$ and $\mathcal{A}_1$–$\mathcal{A}_2$.

$$
\begin{aligned}
&\mathit{ValidSystemTraces}_{IIK} \mathrel{\widehat{=}} \\
&\quad \{tr \in \{|\mathit{send}_{TL}, \mathit{receive}_{TL}, \mathit{put}_{TL}, \mathit{get}_{TL}, \mathit{send}, \mathit{receive}, \mathit{fake}, \mathit{hijack}|\}^* \mid \\
&\qquad \forall tr' \leqslant tr \cdot \\
&\qquad\quad \mathcal{N}_1(tr') \land \mathcal{N}_2(tr') \land \mathcal{N}_{3,IIK}(tr') \land \mathcal{N}_4(tr') \land \mathcal{A}_1(tr') \land \mathcal{A}_2(tr')\} \,.
\end{aligned}
$$

Definition 3.1.4 (Channel specification)**.** A channel specification is a predicate over traces:

$$\forall IIK \subseteq \mathit{Message} \cdot \mathit{ChannelSpec} : \mathit{ValidSystemTraces}_{IIK} \to \mathbb{B} \,.$$

Every channel specification we consider is built as the conjunction of simpler properties of the form $P(R_i \to R_j)$, which talk about messages on the channel $R_i \to R_j$.

Definition 3.1.5. A channel specification has a natural interpretation: the set of valid system traces that it accepts, assuming some value of the intruder's initial knowledge:

$$
\begin{aligned}
&\mathit{traces}_{IIK}(\mathit{ChannelSpec}) = \\
&\quad \{tr \in \mathit{ValidSystemTraces}_{IIK} \mid \forall tr' \leqslant tr \cdot \mathit{ChannelSpec}(tr')\} \,.
\end{aligned}
$$

We omit the intruder's initial knowledge when it is assumed to be constant, or if a property or a definition should be interpreted independently of it.

We note that a channel specification P, built as a conjunction of simpler properties $P_k(R_i \to R_j)$, is not necessarily a unique description of $\mathit{traces}(P)$. Any channel specification $P_k(R_i \to R_j)$ necessarily holds for every trace in $\mathit{traces}(P)$, but if it turns out that $Q(R_i \to R_j)(tr)$ holds for every trace tr in $\mathit{traces}(P)$, it does not necessarily follow that Q is one of the P_k.

We also note that if we have two channel specifications P and Q such that $P \Rightarrow Q$, then a channel that satisfies P can be used anywhere a channel that satisfies Q can be used and, in this case, $\mathit{traces}(P) \subseteq \mathit{traces}(Q)$. In Section 3.3 we see some pairs of channels that are not equivalent as predicates, but which simulate one another; we collapse such pairs.

3.2 Confidential channels

In this section we specify confidential channels in our framework. Confidential channels are specified in terms of the intruder's knowledge after observing (and interacting with) a valid system trace. The confidential channel specification uses the transport-layer events as well as the application-layer events.

A confidential channel should protect the confidentiality of any message sent on it from all but the intended recipient. For example, a confidential channel to B can be implemented by encoding the application layer message m as the transport layer message $\{m\}_{PK(B)}$. We identify confidential channels by tagging them with the label C (e.g. writing $C(R_i \rightarrow R_j)$). The notion of confidentiality we consider is that of Dolev and Yao [DY83]: the intruder can only decrypt messages when he possesses the decryption key; we do not attempt to capture any other definition of confidentiality (such as indistinguishability).

The *IntruderKnows* function is then defined so that the intruder only learns messages that are sent on non-confidential channels, or that are sent to him:

$$\begin{array}{l} IntruderKnows_{IIK} : Trace \rightarrow \mathbb{P}(Message_{App}) \\ IntruderKnows_{IIK}(tr) \mathrel{\widehat=} \\ \quad \{m \mid (IIK \cup SentToIntruder(tr) \cup SentOnNonConfidential(tr)) \vdash m\} \,. \end{array}$$

$IIK \subseteq Message$ is the intruder's initial knowledge. *SentToIntruder* gives the set of messages sent by honest agents to dishonest agents:

$$\begin{array}{l} SentToIntruder : Trace \rightarrow \mathbb{P}(Message_{App}) \\ SentToIntruder(tr) \mathrel{\widehat=} \\ \quad \{m \mid \exists A : Agent; c_A : Connection; I : Dishonest \cdot send.A.c_A.I.m \textbf{ in } tr\} \,. \end{array}$$

SentOnNonConfidential gives the set of messages sent between agents on non-confidential channels:

$$\begin{array}{l} SentOnNonConfidential : Trace \rightarrow \mathbb{P}(Message_{App}) \\ SentOnNonConfidential(tr) \mathrel{\widehat=} \\ \quad \{m \mid \exists R_i, R_j : Role \cdot \neg C(R_i \rightarrow R_j) \wedge \\ \qquad \exists A : \hat{R}_i; c_A : Connection; B : \hat{R}_j \cdot send.A.c_A.B.m \textbf{ in } tr\} \,. \end{array}$$

We give channel specifications that tag some channels as being confidential in this way. When we consider these specifications we examine their traces (as defined in Section 3.1); the traces of the specification clearly depend on which channels are confidential, but they also depend on the intruder's initial knowledge. As we increase the set of messages the intruder knows initially, we increase the number of deductions he can make, and so we increase the number of messages he can send. When we claim that a

secure transport layer is confidential, we usually make that claim subject to a restriction on the intruder's initial knowledge (usually that the intruder's initial knowledge does not contain the honest agents' secret keys).

We specify confidential channels by requiring that the $IntruderKnows(tr)$ function does indeed capture what the intruder would know after the trace tr. The messages the intruder knows after observing a trace are those that can be deduced from his initial knowledge and the messages sent on the network:

$$
\begin{array}{l}
IntruderKnows_{TL,IIK} : Trace \to \mathbb{P}(Message) \\
IntruderKnows_{TL,IIK}(tr) \mathrel{\widehat=} \\
\quad \{m \mid \{m' \mid \exists A, B : Agent; c_A : Connection \cdot send_{TL}.A.c_A.B.m' \textbf{ in } tr\} \cup \\
\qquad IIK \vdash m\} \,.
\end{array}
$$

The confidential channels must protect the confidentiality of the messages sent on them. In other words, although the intruder can see the transport layer messages that are sent on the network, he ought not to be able to deduce the application layer messages within them. For any implementation of a confidential channel, *ChannelImp*, the intruder should gain exactly the same knowledge by listening to the transport-layer messages as he does by listening to the application-layer messages.

$$
\begin{array}{l}
\forall IIK \subseteq Message; tr \in traces_{IIK}(ChannelImp) \cdot \\
\quad IntruderKnows_{TL,IIK}(tr) \cap Message_{App} = IntruderKnows_{IIK}(tr) \,.
\end{array}
$$

3.3 Authenticated channels

In this section we describe our authenticated channel specifications. Unlike the confidential channel property, the specifications for these channels only refer to the application-layer events. These properties are specified by defining several building blocks; these building blocks progressively block the intruder's behaviour. The building blocks in this section, and the confidential property from the previous section are combined to construct more complex properties: these are the channel specifications.

We specify authenticated channels by describing the relationship between the *receive* and *send* events performed by the agents at either end of the channel. In particular, we specify under what circumstances an agent may perform a particular *receive* event. Our specifications are trace-based (and hence are safety properties[4] [Ros98]).

We are building a hierarchy of channels, so we need a bottom element: the standard Dolev-Yao network model used by many protocol analyses (this is captured by statements $\mathcal{N}_1$–$\mathcal{N}_4$ in Section 3.1).

[4]They identify the traces that the network cannot perform, but cannot force it to perform any events at all; this is one of the reasons that none of our channels guarantee that a message will be received.

There are two dishonest events the intruder can perform: faking and hijacking. As the examples in Section 3.1 show, with some transport protocols the latter can only be performed using dishonest identities. We specify our channels by placing restrictions on when he can perform these events. The restrictions below are the building blocks that we use to construct more interesting properties.

Definition 3.3.1 (No-faking)**.** If $NF(R_i \rightarrow R_j)$ then the intruder cannot fake messages on the channel:

$$NF(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} tr \downarrow \{\!| \mathit{fake}.\hat{R}_i.\hat{R}_j |\!\} = \langle\rangle \,.$$

Definition 3.3.2 (No-re-ascribing)**.** If $NRA(R_i \rightarrow R_j)$ then the intruder cannot change the sender's identity when he hijacks messages:

$$\begin{aligned} &NRA(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{\!| \mathit{hijack}.A \rightarrow A'.B \rightarrow B' \mid A, A' : \hat{R}_i; B, B' : \hat{R}_j \cdot A \neq A' |\!\} = \langle\rangle \,. \end{aligned}$$

Definition 3.3.3 (No-honest-re-ascribing)**.** If $NRA^-(R_i \rightarrow R_j)$ then the intruder can only change the sender's identity to a dishonest identity when he hijacks messages:

$$\begin{aligned} &NRA^-(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{\!| \mathit{hijack}.A \rightarrow A'.B \rightarrow B' \mid A, A' : \hat{R}_i; B, B' : \hat{R}_j \cdot \\ &\qquad\qquad A \neq A' \wedge \mathit{Honest}(A') |\!\} = \langle\rangle \,. \end{aligned}$$

Definition 3.3.4 (No-redirecting)**.** If $NR(R_i \rightarrow R_j)$ then the intruder cannot redirect messages:

$$\begin{aligned} &NR(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{\!| \mathit{hijack}.A \rightarrow A'.B \rightarrow B' \mid A, A' : \hat{R}_i; B, B' : \hat{R}_j \cdot B \neq B' |\!\} = \langle\rangle \,. \end{aligned}$$

Definition 3.3.5 (No-honest-redirecting)**.** If $NR^-(R_i \rightarrow R_j)$ then the intruder cannot redirect messages that were sent to honest agents:

$$\begin{aligned} &NR^-(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{\!| \mathit{hijack}.A \rightarrow A'.B \rightarrow B' \mid A, A' : \hat{R}_i; B, B' : \hat{R}_j \cdot \\ &\qquad\qquad B \neq B' \wedge \mathit{Honest}(B) |\!\} = \langle\rangle \,. \end{aligned}$$

All of the above specifications work by blocking events; when we specify this we do not mean that the intruder cannot generate the application-layer fake and hijack events on the channels. What we intend is that when the intruder generates such events, he will either be unable to modify the transport-layer messages in order to generate the necessary put_{TL} events, or the honest protocol agents will reject the messages. Any behaviour of the system where the events are generated but then rejected can be simulated by

a behaviour where the events are not created. The simplest way to specify these properties is to ban the events.

An alternative to Definitions 3.3.1–3.3.5 would be to give eighteen variants of $\mathcal{N}_4$,[5] and to apply them to channels, rather than the entire network. By specifying one of the properties above, we limit the possibilities of the single $\mathcal{N}_4$ rule. We discuss our choice of approach to the specifications, and discuss some refinements and extensions to these building blocks in Section 4.6. In the next section we will combine these building blocks to form more interesting properties.

The intruder can use a hijack event to cause an honest agent to receive a message in a particular connection without changing either of the identities associated with the message. This activity is not blocked by any of the properties above. In particular, the intruder can cause an agent to receive a message more times than it is sent, i.e. to replay a message. We do not specify a no-replaying property in the building blocks because we do not wish to consider it independently; this is for two reasons:

- Replaying is fundamentally different to re-ascribing and redirecting. Without some preventative mechanism built into the channel, messages can be replayed simply by replaying the transport-layer (or lower-layer) messages. In order to re-ascribe or to redirect a message the intruder will typically have to modify the transport-layer message;

- It is, typically, expensive to implement a channel that prevents replaying. For example, adding authenticated, unique serial numbers to individual messages requires the message recipient to remember all the serial numbers they have seen previously in order to detect replays. However, there are other properties one can implement that provide no-replaying at little, or no, extra cost (for example, stream channels prevent replaying – see Section 3.5).

In Section 3.5 we describe channel properties that bind messages into sessions: these channels specify that a message cannot be replayed outside a particular connection. However, these channels do not prevent messages from being replayed within the session. We also give session properties that indirectly prevent replaying, and properties that provide a stronger guarantee (e.g. that messages are received in the same order that they were sent). In later sections we do not always explicitly state that a message may have been replayed, but we do not block this possibility either.

3.3.1 Combining the building blocks

We now consider how the building blocks can be combined. They are not independent, since no-re-ascribing implies no-honest-re-ascribing, and like-

[5]There are two possibilities for no-faking, three possibilities for no-re-ascribing and three possibilities for no-redirecting.

wise for no-redirecting. Further, not all combinations are fundamentally different; certain pairs of combinations allow essentially the same intruder behaviours: each simulates the other. We therefore collapse such combinations.

Collapse$_1$: Non-confidential channels that allow faking but which satisfy one of the forms of no-re-ascribing or no-redirecting simulate the bottom channel; the intruder can learn messages and fake them to effect a message re-ascribe or redirect. For example, the trace $\langle send.A.c_A.B.m, fake.A.B'.c_{B'}.m\rangle$ simulates a redirection of m from B to B'.

Collapse$_2$: Any re-ascribable channel that prevents faking simulates a re-ascribable channel that allows faking: the intruder can send messages with his own identity and then re-ascribe them; this activity simulates a fake; e.g. $\langle send.I.c_I.B.m, hijack.I \rightarrow A.B.c_B.m\rangle$.

Collapse$_3$: Non-confidential channels that satisfy $NF \wedge NRA$ simulate non-confidential channels that satisfy $NF \wedge NRA^-$; the intruder can always learn messages and then send them with his own identity to simulate a dishonest re-ascribe; e.g. $\langle send.A.B.c_B.m, send.I.c_I.B.m\rangle$.

Collapse$_4$: Confidential channels that do not satisfy NR^- or NR simulate non-confidential channels because the intruder can redirect messages sent on them to himself, and so learn the messages.

Collapse$_5$: Confidential, fakeable channels that satisfy NR simulate confidential, fakeable channels that satisfy NR^-; the intruder learns messages that are sent to him, and so can fake them; e.g. $\langle send.A.c_A.I.m, fake.A.B.c_B.m\rangle$.

After taking these collapsing cases into consideration we arrive at a hierarchy of four non-confidential and seven confidential channels, shown in Figure 3.3 (where several cases collapse to one, the figure gives the weakest specification in each case). In Figure 3.4 we give simple example transport protocols that we believe satisfy each of the properties; we explain the names in the right-hand column when we discuss these combinations in Section 3.4.

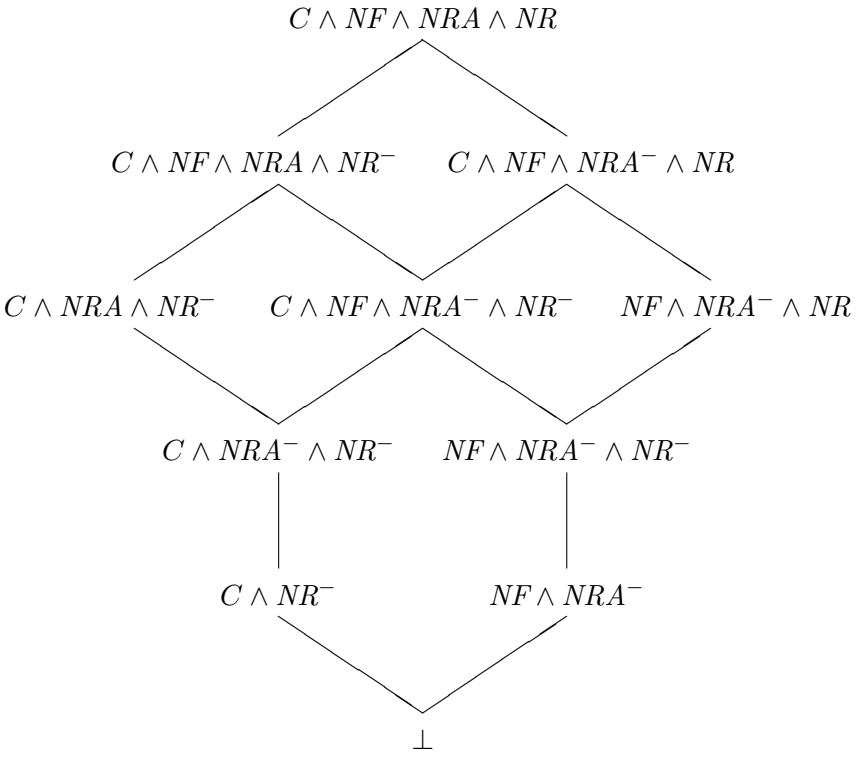


Figure 3.3: The hierarchy of secure channels.

Specification				Example	Name
$\bot$				m, A, B	Dolev-Yao
	NF	NRA^-		$\{m\}_{SK(A)}, B$ or unilateral TLS (Server → Client)	Sender authentication
	NF	NRA^-	NR^-	$\{h(m, n_A)\}_{SK(A)}, \{n_A\}_{PK(B)}, m$	Responsibility
	NF	NRA^-	NR	$\{m, B\}_{SK(A)}$	Strong authentication
C			NR^-	$\{m\}_{PK(B)}, A$ or unilateral TLS (Client → Server)	Confidentiality and intent
C		NRA^-	NR^-	$\{m, k\}_{PK(B)}, \{m \oplus k\}_{SK(A)}$	
C		NRA	NR^-	$\{m, A\}_{PK(B)}, A$	Credit
C	NF	NRA^-	NR^-	$\{h(m, n_A)\}_{SK(A)}, \{m, n_A\}_{PK(B)}$	Responsibility
C	NF	NRA^-	NR	$\{\{m\}_{PK(B)}\}_{SK(A)}$	
C	NF	NRA	NR^-	$\{\{m\}_{SK(A)}\}_{PK(B)}, A$	
C	NF	NRA	NR	$\{\{m, A\}_{PK(B)}\}_{SK(A)}$, $\{\{m, B\}_{SK(A)}\}_{PK(B)}$, or mutual TLS	Strong authentication

Figure 3.4: Example implementations of the secure channels.

3.4 Some interesting authenticated channels

In the previous sections we introduced a set of building blocks for creating authenticated channel specifications. These building blocks can be combined with the confidential channel specification in eleven different ways. The eleven channels created by these combinations are arranged into a hierarchy (shown in Figure 3.3) where the specifications of lower channels are implied by the specifications of higher channels. In the next chapter we introduce a simulation relation that compares the honest traces of channel specifications; we show that the simulation relation establishes the same hierarchy of these channel properties as implication of their specifications. In this section we examine some of the channels in more detail, and describe which of these properties we believe are satisfied by the secure transport layers described in Chapter 2.

3.4.1 Sender authentication

When an agent B receives a message, purportedly from A, he might ask whether he can be sure that A really sent the message. In other words: at some point in the past, did A send that message to someone, not necessarily B? This condition is certainly not met by a *fake.A* or a *hijack.*$A' \rightarrow A$ event: we want to guarantee the existence of a *send.A* event for the message, sometime in the past. However, we should not discount the possibility that A sent a message that the intruder redirected.

Definition 3.4.1 (Sender authentication)**.** The channel $R_i \rightarrow R_j$ provides sender authentication if $NF(R_i \rightarrow R_j) \wedge NRA(R_i \rightarrow R_j)$.

An obvious way to implement this property is for agents to sign messages they send with their secret key: $\{m\}_{SK(A)}$. The signature does not contain the intended recipient's identity, so a channel implemented in this way is redirectable. The intruder cannot fake messages on this channel, nor re-ascribe messages sent by other agents so that they appear to have been sent by A, because he does not know A's secret key. He can, however, learn the message, sign it himself and then send it using his own identity (note that this is a send rather that a re-ascribe); as noted above (collapsing case $Collapse_3$), any non-confidential channel that satisfies $NF \wedge NRA$ simulates a non-confidential channel that satisfies $NF \wedge NRA^{-}$.

With unilateral TLS (i.e. the standard web model), the client is not authenticated to the server. The channel from the server to the client provides authentication of the server's identity, but as the client's identity is not verified, this channel is redirectable (the messages may be received by someone other than the agent the server intended them for) and hence does not satisfy confidentiality. We believe this channel satis-

fies $\mathit{StrongStream} \wedge \mathit{NF} \wedge \mathit{NRA}$.[6]

3.4.2 Intent

When agents sign messages with their secret key, their intent might not be preserved: the intruder can redirect their messages to whomever he likes. We now specify a channel that provides a guarantee of (the original sender's) intent: whenever B receives a message, he knows that the agent who originally sent it intended him to receive it. On these channels we forbid redirection (as this would allow the intruder to change the recipient's identity), but we allow faking and re-ascribing.

Definition 3.4.2 (Intent)**.** The channel $R_i \rightarrow R_j$ provides a guarantee of intent if $\mathit{NR}(R_i \rightarrow R_j)$.

The easiest way to design a channel that provides a guarantee of intent is to encrypt messages with the intended recipient's public key. We have already used this method as the most obvious implementation of a confidential channel.

Recall that non-confidential, non-redirectable, fakeable channels simulate message redirection by learning messages and faking them (collapsing case $\mathit{Collapse}_1$). We therefore always combine intent with confidentiality or non-fakeability. Further, fakeable, confidential channels that satisfy NR can simulate fakeable, confidential channels that satisfy NR^-, because the intruder learns messages that are sent to him, and so can fake them to 'redirect' them to another agent (collapsing case $\mathit{Collapse}_5$).

With unilateral TLS, the channel from the client to the server provides a guarantee of the sender's (the client's) intent, as the client must have verified the server's identity; however it does not provide authentication of the client's identity. We believe this channel satisfies $\mathit{StrongStream} \wedge C \wedge \mathit{NR}$.

3.4.3 Strong authentication

Strong authentication is the combination of the previous two properties: whenever B receives a message from A, A previously sent that message to B. By analogy with [Low97], we often refer to sender authentication as *weak authentication*, and (strong) authentication as *authentication*.

Definition 3.4.3 (Strong authentication)**.** The channel $R_i \rightarrow R_j$ provides strong authentication if $\mathit{NF}(R_i \rightarrow R_j) \wedge \mathit{NRA}(R_i \rightarrow R_j) \wedge \mathit{NR}(R_i \rightarrow R_j)$.

We can achieve strong authentication by encoding m as $\{B, m\}_{SK(A)}$. The intruder cannot change the recipient's identity while maintaining A's signature, so this channel is unredirectable; he cannot fake messages on this

[6] *StrongStream* is a session property and is defined in Section 3.5.

channel because he does not know A's secret key; and he cannot re-ascribe messages so that they appear to have been sent by an honest agent. (As with sender authentication, he can learn the message and sign it himself; again this is not a re-ascribe.) This channel guarantees that when B receives a message from A, then previously A sent it to B.

We believe that bilateral TLS establishes an authenticated stream in each direction, and so both channels satisfy $\mathit{StrongStream} \wedge C \wedge \mathit{NF} \wedge \mathit{NRA} \wedge \mathit{NR}$. Such a channel is equivalent to the authenticated channels of Broadfoot and Lowe [BL03] (see Section 4.6 for more details).

We note that neither $\{\{m\}_{SK(A)}\}_{PK(B)}$ nor $\{\{m\}_{PK(B)}\}_{SK(A)}$ provides strong authentication; the former can be redirected when B is dishonest, and the latter re-ascribed with a dishonest identity; they satisfy, respectively, $C \wedge \mathit{NF} \wedge \mathit{NRA} \wedge \mathit{NR}^-$ and $C \wedge \mathit{NF} \wedge \mathit{NRA}^- \wedge \mathit{NR}$. By concatenating the recipient's identity and the sender's identity to the message in the above channels we do create strong authentication channels; e.g.:

$$\{\{B, m\}_{SK(A)}\}_{PK(B)}\,,$$
$$\{\{A, m\}_{PK(B)}\}_{SK(A)}\,.$$

3.4.4 Credit and responsibility

In [Aba98], Abadi highlighted two different facets of authentication. When an agent B receives a message m from an authenticated agent A, he could interpret it in two different ways:

- He might attribute *credit* for the message m to A; for example, if B is running a competition, and m is an entry to the competition, he would give credit for that entry to A;
- He might believe that the message is supported by A's authority, and so assign *responsibility* for it to A; for example, if m is a request to delete a file, then his decision will depend on whether or not A has the authority to delete the file.

Abadi argued that these two interpretations of authentication are not the same, and that protocol designers tend not to state which form of authentication their protocols provide: in many cases protocols will offer one, but not the other.

Definition 3.4.4 (Credit)**.** The channel $R_i \rightarrow R_j$ can be used to give credit if $C(R_i \rightarrow R_j) \wedge \mathit{NRA}(R_i \rightarrow R_j) \wedge \mathit{NR}^-(R_i \rightarrow R_j)$.

The intruder can fake messages on these channels, but in doing so he only gives another agent credit for his messages.

Abadi gives the following example of a protocol suitable for assigning credit:

$$A \rightarrow B : \{A, K\}_{PK(B)}, \{m\}_K \,.$$

When B receives this message he knows that he can give credit for m to the person who encrypted the key k; however he cannot be sure that it was really A who did this. So while the intruder can fake messages on this channel, he will only be giving credit to someone else, rather than claiming it for himself.

Definition 3.4.5 (Responsibility)**.** The channel $R_i \rightarrow R_j$ can be used to assign responsibility if $NF(R_i \rightarrow R_j) \wedge NRA^{-}(R_i \rightarrow R_j) \wedge NR^{-}(R_i \rightarrow R_j)$.

The only attack the intruder could perform on such a channel would be to overhear a message, or to claim it as his own. In the latter case, he will either not have the authority required for the message (as in the example of a fileserver and a delete message), or he will be accepting the blame for something. The example given for an authenticated channel would be a suitable implementation of this channel.

In some circumstances, one might wish to strengthen such a channel so that it also provides intent (i.e. $NF \wedge NRA^{-} \wedge NR$), to ensure that the correct agent assigns the responsibility.

3.4.5 Guaranteed Knowledge

Both of the previous channels (credit and responsibility) provide a further property: they guarantee that the apparent sender of a message knew the content of the message. This is important for these channels as an agent should not be able to claim credit for a message that he does not know, and no agent should claim responsibility for a message that he does not know.

Fakeable channels cannot provide this property: if the intruder can fake messages with another agent's identity, he can send messages that they have no chance of knowing. Further, if the intruder can re-ascribe a message to an honest agent then the channel cannot provide guaranteed knowledge. If the intruder can re-ascribe a message to himself (i.e. to a dishonest agent) then the channel can only provide guaranteed knowledge if it is non-confidential.

Definition 3.4.6 (Guaranteed Knowledge)**.** The channel $R_i \rightarrow R_j$ provides a guarantee that the apparent sender of a message knew the message if $NF(R_i \rightarrow R_j) \wedge NRA^{-}(R_i \rightarrow R_j) \wedge (C(R_i \rightarrow R_j) \Rightarrow NRA(R_i \rightarrow R_j))$.

It is interesting to note that if channel specification P provides guaranteed knowledge and channel Q is stronger than P, then it does not follow that Q provides guaranteed knowledge. For example, $NF \wedge NRA^{-} \wedge NR$ provides guaranteed knowledge, but $C \wedge NF \wedge NRA^{-} \wedge NR$ does not (the intruder can re-ascribe an honest agent's message to himself, but he cannot know the message because the channel is confidential).

3.5 Session and stream channels

The properties described earlier in this chapter allow us to specify channels that provide guarantees for individual messages. In practice it is often necessary to group together different messages that were sent in the same connection into a single *session*; application-layer protocols frequently rely on secure transport protocols to bind messages together in this way. In this section we consider six properties relating different messages in the same connection; they can be combined with the properties of Figure 3.3.

Given a trace tr we might ask whether it is feasible that the event $send.A.c_A.B.m$ is responsible for the event $receive.B'.c_{B'}.A'.m$; i.e. if the first event had not happened, the second might not have. Certainly if $A = A'$ and $B = B'$ then it is quite possible that the first event is the cause of the second. If either of these equalities fail there must be a hijack event between the two events in order for the first to be responsible for the second.

$$
\begin{aligned}
&\mathit{Responsible}_{tr}(\mathit{send}.A.c_A.B.m, \mathit{receive}.B'.c_{B'}.A'.m) \mathrel{\widehat{=}} \\
&\quad \exists tr', tr'' : \mathit{Trace} \cdot tr' \frown \langle \mathit{send}.A.c_A.B.m \rangle \frown tr'' \frown \langle \mathit{receive}.B'.c_{B'}.A'.m \rangle \leqslant tr \wedge \\
&\qquad (A = A' \wedge B = B') \vee \\
&\qquad (\mathit{hijack}.A \rightarrow A'.B \rightarrow B'.c_{B'}.m \ \mathbf{in}\ tr'') \,.
\end{aligned}
$$

The hijack event is only defined when the roles played by the new sender and receiver are the same as those played by the old: a hijacked message cannot be taken from one type of channel and put on another. In order for it to be feasible that a particular send event is responsible for a receive event we assume, implicitly, that the roles of the new and the old sender are the same, and likewise for the receivers; this assumption is hidden in the formulation of the property above in the agents in the events (recall that an agent is a pair: an identity and a role, and that $\hat{R}_i$ stands for $\mathit{Agent} \times R_i$).

We note that in a valid system trace, for any particular receive event there may be more than one send event such that $\mathit{Responsible}_{tr}$ holds. In particular, any system trace tr induces a relation $\mathcal{R}_{tr}$ (*receives-from*) over the set of connection identifiers in that system: $c_B \mathcal{R}_{tr} c_A$ if *all* of the messages received in the connection c_B could feasibly have been sent in the connection c_A.

We need to be careful about the way we deal with connections that receive faked messages. The *fake* event is an abstraction of the activity that the intruder performs when he fakes messages: he creates a new protocol agent with a false identity, and then uses that protocol agent to establish connections to other agents. He then uses these connections to fake messages. We partition *Connection* into honest and intruder connection identifiers. An honest agent's connection that receives faked messages is related to every intruder connection.

$$
\begin{array}{l}
c_B \mathcal{R}_{tr} c_A \mathrel{\widehat{=}} \\
\quad \forall A', B : \mathit{Agent} \cdot \\
\quad\quad \exists A, B' : \mathit{Agent} \cdot \forall m : \mathit{Message}_{App} \cdot \mathit{receive}.B.c_B.A'.m \ \mathbf{in}\ tr \Rightarrow \\
\quad\quad\quad \mathit{Responsible}_{tr}(\mathit{send}.A.c_A.B'.m, \mathit{receive}.B.c_B.A'.m) \vee \\
\quad\quad c_A \in \mathit{IntruderConnection} \wedge \forall m : \mathit{Message}_{App}; tr' : \mathit{Trace} \cdot \\
\quad\quad\quad tr' \frown \langle \mathit{receive}.B.c_B.A'.m \rangle \leqslant tr \Rightarrow \mathit{fake}.A'.B.c_B.m \ \mathbf{in}\ tr' \,.
\end{array}
$$

The connection c_B is related to every connection that could feasibly have sent (or faked) all of the messages that are received in c_B; if no messages are received in c_B a relation is established to every other connection.

It is not hard to formulate valid system traces that induce non-functional relations for connections that receive at least one message.[7] For example, if two agents send sequences of messages that share a common subsequence that is received by another agent:

$$
tr \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.B.m, \mathit{send}.A.c'_A.B.m, \mathit{receive}.B.c_B.A.m \rangle \,.
$$

In this example trace, A sends a message to B in two different connections (c_A and c'_A), and B receives one copy of that message in the connection c_B; hence $c_B \mathcal{R}_{tr} c_A$ and $c_B \mathcal{R}_{tr} c'_A$.

For such traces there are different ways of interpreting the events and of resolving the non-determinism in the relation. Each of these interpretations is represented by a *maximal functional refinement* of $\mathcal{R}_{tr}$.

Definition 3.5.1 (Maximal functional refinement)**.** A maximal functional refinement of the relation $\mathcal{R}_{tr}$ is any relation $\mathcal{R}'$ that:

1. Is a subset of $\mathcal{R}_{tr}$;
2. Has the same left-image as $\mathcal{R}_{tr}$;
3. Is functional (where $\mathcal{R}_{tr}$ might not be).

We write $\mathcal{R}' \ll \mathcal{R}_{tr}$ when $\mathcal{R}'$ is a maximal functional refinement of $\mathcal{R}_{tr}$. Note that $\ll$ is not necessarily reflexive, but if $\mathcal{R}_{tr} \ll \mathcal{R}_{tr}$ then $\mathcal{R}_{tr}$ is functional so has no proper refinements.

We can think of these refinements as possible ways of resolving the non-determinism in $\mathcal{R}_{tr}$.

In order to specify session and stream properties we place conditions on the maximal functional refinements of the receives-from relation. We also restrict attention to a pair of roles (R_i, R_j) (i.e. we restrict traces to events on the channels $R_i \to R_j$ and $R_j \to R_i$) so

[7] Non-functional relations are always induced for connections that do not receive any messages.

that the relation refers only to the connection identifiers of agents playing roles R_i and R_j; we define $\mathcal{R}_{tr} \upharpoonright (R_i \to R_j) = \mathcal{R}_{tr \upharpoonright \Sigma(R_i \to R_j)}$, and $\mathcal{R}_{tr} \upharpoonright \{R_i, R_j\} \mathrel{\widehat{=}} \mathcal{R}_{tr \upharpoonright \Sigma(R_i \to R_j) \cup \Sigma(R_j \to R_i)}$, where:

$$\Sigma(R_i \to R_j) = \{| \, send.\hat{R}_i.Connection.\hat{R}_j,\ receive.\hat{R}_j.Connection.\hat{R}_i,\\ fake.\hat{R}_i.\hat{R}_j,\ hijack.\hat{R}_i \to \hat{R}_i.\hat{R}_j \to \hat{R}_j \, |\} \,.$$

3.5.1 Session channels

Consider the example implementation of a secure channel that satisfies $C \wedge NR^{-}$ given in Figure 3.4:

$$A \to B : \{m\}_{PK(B)}, A \,.$$

There is nothing in the transport layer message to distinguish this message from one sent by A to B in a different connection. It is clear that if A does send two messages to B in different connections, the system will accept traces in which B receives them in a single connection. Further, since the intruder can fake messages on this channel, it is possible that B receives a mix of messages from A and from the intruder in the same session:

$$tr \mathrel{\widehat{=}} \langle send.A.c_A.B.m_1, receive.B.c_B.A.m_1,\\ fake.A.B.c_B.m_2, receive.B.c_B.A.m_2 \rangle \,.$$

If A's protocol agent included a fresh nonce with the first message that A sent to B, and then sent that nonce with each subsequent message then, as long as that nonce remains secret, neither of the attacks above are possible:

$$A \to B : \{n_A, m\}_{PK(B)}, A \,.$$

The messages that A sends are bound together by the nonce, so B's protocol agent knows that each message it receives from A was sent in a single connection, and so will ensure that B receives them in a single connection. It is impossible for the intruder to inject messages into the connection as he does not know the nonce. In order to fake messages the intruder must bind them together with a single nonce; in this case they can be thought of as coming from a single dishonest connection.

This modification allows us to use this transport layer protocol to establish sessions: all of the messages sent in a single connection will be received in a single connection, and the intruder cannot inject messages into the session. The intruder can still remove and re-order messages within the session.

Definition 3.5.2 (Session)**.** A channel $R_i \to R_j$ is a session channel if the relation $\mathcal{R}_{tr} \upharpoonright (R_i \to R_j)$ is left-total:

$$Session(R_i \to R_j)(tr) \mathrel{\widehat{=}}\\ \forall c_B : Connection \cdot \exists c_A : Connection \cdot (c_B, c_A) \in \mathcal{R}_{tr} \upharpoonright (R_i \to R_j) \,.$$

If $\mathcal{R}_{tr} \upharpoonright (R_i \rightarrow R_j)$ is left-total then each of its maximal functional refinements is also left-total because they all have the same domain as $\mathcal{R}_{tr} \upharpoonright (R_i \rightarrow R_j)$.

The transport layer protocol described above has the unfortunate property that the intruder can replay messages from old sessions, and cause B to believe that A wishes to start a new session with him. This is an attack against the injectivity of the transport protocol. Similar attacks are possible against other transport protocols in which, for example, one session is played to two different agents. The next property prevents this sort of attack.

Definition 3.5.3 (Injective Session)**.** A channel $R_i \rightarrow R_j$ is an injective session channel if it is a session channel and there exists an injective maximal functional refinement $\mathcal{R}'$ of the relation $\mathcal{R}_{tr} \upharpoonright (R_i \rightarrow R_j)$:

$$
\begin{array}{l}
\mathit{InjectiveSession}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \mathit{Session}(R_i \rightarrow R_j)(tr) \wedge \\
\quad \exists \mathcal{R}' \ll \mathcal{R}_{tr} \upharpoonright (R_i \rightarrow R_j) \cdot \\
\qquad \forall c_A, c_B, c_{B'} : \mathit{Connection} \cdot c_B \mathcal{R}' c_A \wedge c_{B'} \mathcal{R}' c_A \Rightarrow c_B = c_{B'} \, .
\end{array}
$$

The security parameters of a TLS connection are used to protect the integrity of every record layer message. This integrity check, and the secrecy of the security parameters ensures that TLS is a session channel. The agents calculate the security parameters together in the handshake, so they both know they have contributed to the values of the security parameters, and so they are communicating in a new session. In order to replay a TLS session the intruder would have to be able to choose the security parameters to match those of the old session. TLS therefore establishes injective sessions.

The TLS handshake protocol ensures that the connections held by the client and server are bound together in a single session. However, it is not the case that every injective session channel achieves this; consider the following trace (illustrated in Figure 3.5):

$$
\begin{array}{rl}
tr \mathrel{\widehat{=}} \langle & \mathit{send}.A.c_A.B.m_1, \mathit{receive}.B.c_B.A.m_1, \\
& \mathit{send}.B.c'_B.A.m_2, \mathit{receive}.A.c_A.B.m_2, \\
& \mathit{send}.A.c'_A.B.m_3, \mathit{receive}.B.c'_B.A.m_3, \\
& \mathit{send}.B.c_B.A.m_4, \mathit{receive}.A.c'_A.B.m_4 \rangle \, .
\end{array}
$$

This trace satisfies the injective session property: each connection receives messages from exactly one other connection. However, the connections have been interleaved in such a way that the message A receives in c_A is not in response to the message she sent in c_A.

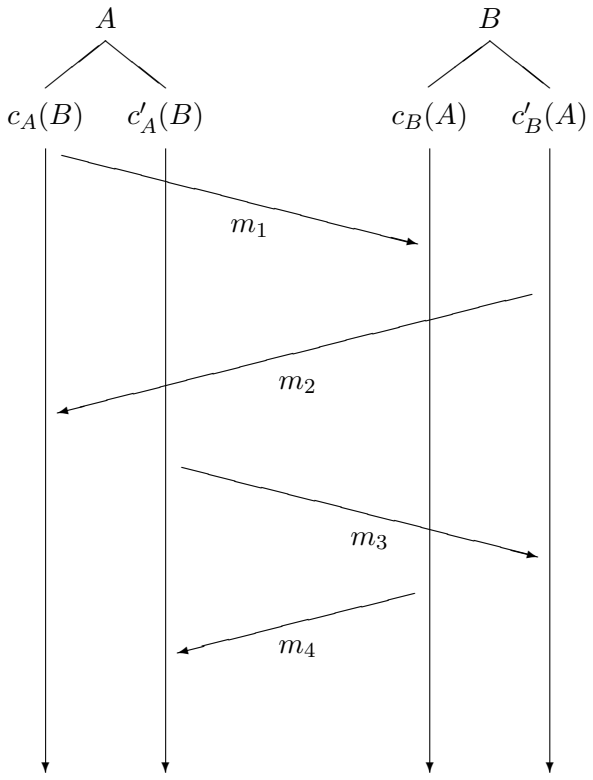


Figure 3.5: The connections between A and B are interleaved: the message received in c_A is not in response to that sent in c_A.

Definition 3.5.4 (Strong Session)**.** A channel $R_i \rightarrow R_j$ is a strong session channel if the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ are session channels and there exists a symmetric maximal functional refinement $\mathcal{R}'$ of the relation $\mathcal{R}_{tr} \upharpoonright \{R_i, R_j\}$:

$$\begin{aligned} &\mathit{StrongSession}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\ &\quad \mathit{Session}(R_i \rightarrow R_j)(tr) \wedge \mathit{Session}(R_j \rightarrow R_i)(tr) \wedge \\ &\quad \exists \mathcal{R}' \ll \mathcal{R}_{tr} \upharpoonright \{R_i, R_j\} \cdot \forall c_A, c_B : \mathit{Connection} \cdot c_B \mathcal{R}' c_A \Rightarrow c_A \mathcal{R}' c_B \,. \end{aligned}$$

The strong session property ensures that the sort of session interleaving and de-coupling, as shown above, cannot happen.

A strong session also ensures injectivity: if $c_B \mathcal{R}' c_A$ and $c_{B'} \mathcal{R}' c_A$ then, because the relation is symmetric, $c_A \mathcal{R}' c_B$ and $c_A \mathcal{R}' c_{B'}$ and so, because the relation is functional, $c_B = c_{B'}$. This symmetry also ensures that $\mathit{StrongSession}(R_i \rightarrow R_j) \Leftrightarrow \mathit{StrongSession}(R_j \rightarrow R_i)$. It does not make much sense to talk about $R_i \rightarrow R_j$ as a strong session channel without $R_j \rightarrow R_i$ also being a strong session channel, so we write $\mathit{StrongSession}(R_i \leftrightarrow R_j)$.

3.5.2 Stream channels

The record layer of TLS also includes sequence numbers. Every time an agent sends a message in a TLS connection their protocol agent increases the sequence number that is sent with the message. The sequence number is authenticated by the usual TLS integrity protection, so the recipient of a TLS stream can be sure that he has not missed any messages, nor received messages in any order other than that intended by the sender.

TLS therefore provides a stronger guarantee than the strong session property: the stream of messages an agent receives is a prefix of the stream of messages sent by the other agent in the session. This property prevents the intruder from permuting the order in which messages are received, or inserting or removing messages from a session. However, the intruder can terminate a stream at any point.

We define stream channels by altering the definition of the *receives-from* relation $\mathcal{R}_{tr}$ to form the *stream-receives-from* relation: $\mathcal{S}_{tr}$. $c_B \mathcal{S}_{tr} c_A$ if the stream of messages received in c_B is a prefix of the stream of messages sent in c_A. In other words, every message that is received in c_B was previously sent in c_A in the same order, and, subject to termination of the stream, every message sent in c_A is received in c_B. *Stream*, *injective stream* and *strong stream* channels are defined in the same way as the session channels using $\mathcal{S}_{tr}$ in place of $\mathcal{R}_{tr}$.

$$
\begin{array}{l}
c_B \mathcal{S}_{tr} c_A \mathrel{\widehat{=}} \\
\quad \forall A', B : \mathit{Agent} \cdot \\
\quad\quad \exists A, B' : \mathit{Agent} \cdot tr \downarrow \mathit{receive}.B.c_B.A' \leqslant tr \downarrow \mathit{send}.A.c_A.B' \wedge \\
\quad\quad\quad \forall m : \mathit{Message}_{App} \cdot \mathit{receive}.B.c_B.A'.m \ \mathbf{in}\ tr \Rightarrow \\
\quad\quad\quad\quad \mathit{Responsible}_{tr}(\mathit{send}.A.c_A.B'.m, \mathit{receive}.B.c_B.A'.m) \vee \\
\quad\quad c_A \in \mathit{IntruderConnection} \wedge tr \downarrow \mathit{receive}.B.c_B.A' \leqslant tr \downarrow \mathit{fake}.A'.B.c_B \,.
\end{array}
$$

Definition 3.5.5 (Stream)**.** A channel $R_i \rightarrow R_j$ is a stream channel if the relation $\mathcal{S}_{tr} \upharpoonright (R_i \rightarrow R_j)$ is left-total:

$$
\begin{array}{l}
\mathit{Stream}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \forall c_B : \mathit{Connection} \cdot \exists c_A : \mathit{Connection} \cdot (c_B, c_A) \in \mathcal{S}_{tr} \upharpoonright (R_i \rightarrow R_j) \,.
\end{array}
$$

Definition 3.5.6 (Injective Stream)**.** A channel $R_i \rightarrow R_j$ is an injective stream channel if it is a stream channel and there exists an injective maximal functional refinement $\mathcal{S}'$ of the relation $\mathcal{S}_{tr} \upharpoonright (R_i \rightarrow R_j)$:

$$
\begin{array}{l}
\mathit{InjectiveStream}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \mathit{Stream}(R_i \rightarrow R_j)(tr) \wedge \\
\quad \exists \mathcal{S}' \ll \mathcal{S}_{tr} \upharpoonright (R_i \rightarrow R_j) \cdot \\
\quad\quad \forall c_A, c_B, c_{B'} : \mathit{Connection} \cdot c_B \mathcal{S}' c_A \wedge c_{B'} \mathcal{S}' c_A \Rightarrow c_B = c_{B'} \,.
\end{array}
$$

Definition 3.5.7 (Strong Stream)**.** A channel $R_i \rightarrow R_j$ is a strong stream channel if the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ are stream channels and there exists a symmetric maximal functional refinement $\mathcal{S}'$ of the relation $\mathcal{S}_{tr} \upharpoonright \{R_i, R_j\}$:

$$
\begin{array}{l}
\mathit{StrongStream}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \mathit{Stream}(R_i \rightarrow R_j)(tr) \wedge \mathit{Stream}(R_j \rightarrow R_i)(tr) \wedge \\
\quad \exists \mathcal{S}' \ll \mathcal{S}_{tr} \upharpoonright \{R_i, R_j\} \cdot \forall c_A, c_B : \mathit{Connection} \cdot c_B \mathcal{S}' c_A \Rightarrow c_A \mathcal{S}' c_B \,.
\end{array}
$$

We believe that TLS (in unilateral and bilateral mode) establishes strong stream channels [KL08].

It is clear that the *stream-receives-from* relation is a subset of the *receives-from* relation (i.e. that $c_B \mathcal{S}_{tr} c_A \Rightarrow c_B \mathcal{R}_{tr} c_A$), hence each of the stream channels is simulated by the equivalent session channel. The simulation relations between the session and stream properties are shown in the session channel hierarchy in Figure 3.7.

We believe that each of the channels in Figure 3.3 except the bottom one can be strengthened to give a session property by including a session identifier in the transport-layer message. However, this must be done with care; for example, the channel that sends messages as $\{\{m\}_{SK(A)}\}_{PK(B)}$ cannot be strengthened to a session channel by including a session identifier outside the sender's signature ($\{\{m\}_{SK(A)}, c_A\}_{PK(B)}$), as this would allow the intruder to take messages from two different sessions between A and himself,

and combine them into a new session between A and some other honest agent. Instead, the session identifier must be bound to the application-layer message: $\{\{m, c_A\}_{SK(A)}\}_{PK(B)}$.

Some of the channels can be further strengthened to give a strong session property. We also believe that any session channel can be strengthened to give a stream property, and any injective or strong session channel can be strengthened to give an injective or strong stream property by binding authenticated sequence numbers to every message, as in TLS.

3.5.3 Synchronised stream channels

A stream channel provides a guarantee to the message recipient that they are receiving messages in the correct order, and that they have not missed any. However, the intruder can delay a stream of messages indefinitely, so the recipient cannot be sure that they have received all of the sender's messages.

The most obvious way to strengthen the stream properties is to restrict the intruder's capability to delay messages. We define the *n-restricted-stream-receives-from* relation: $\mathcal{S}^n_{tr}$. $c_B \mathcal{S}^n_{tr} c_A$ if the stream of messages received in c_B is a prefix of the stream of messages sent in c_A, and there are at most n messages that were sent in c_A that have not yet been received in c_B.

$$
\begin{array}{l}
c_B \mathcal{S}^n_{tr} c_A \mathrel{\widehat{=}} \\
\quad \forall A', B : \mathit{Agent} \cdot \\
\qquad \exists A, B' : \mathit{Agent} \cdot tr \downarrow \mathit{receive}.B.c_B.A' \leqslant tr \downarrow \mathit{send}.A.c_A.B' \wedge \\
\qquad\quad tr \downarrow \{\!|\, \mathit{send}.A.c_A.B' \,|\!\} - tr \downarrow \{\!|\, \mathit{receive}.B.c_B.A' \,|\!\} \leqslant n \wedge \\
\qquad\quad \forall m : \mathit{Message}_{App} \cdot \mathit{receive}.B.c_B.A'.m \ \mathbf{in}\ tr \Rightarrow \\
\qquad\qquad \mathit{Responsible}_{tr}(\mathit{send}.A.c_A.B'.m, \mathit{receive}.B.c_B.A'.m) \vee \\
\qquad c_A \in \mathit{IntruderConnection} \wedge tr \downarrow \mathit{receive}.B.c_B.A' \leqslant tr \downarrow \mathit{fake}.A'.B.c_B \wedge \\
\qquad\quad tr \downarrow \{\!|\, \mathit{fake}.A'.B.c_B \,|\!\} - tr \downarrow \{\!|\, \mathit{receive}.B.c_B.A' \,|\!\} \leqslant n\,.
\end{array}
$$

$\mathcal{S}^1_{tr}$ is the relation established by a channel that only allows the intruder to delay a stream by one message. The message sender cannot send their next message until all of the previous messages have been received; this sort of property could be implemented by the receiver sending an authenticated acknowledgement of every message. We define the synchronised stream properties *SyncStream*, *InjectiveSyncStream* and *StrongSyncStream* analogously to the stream properties using $\mathcal{S}^1_{tr}$ in place of $\mathcal{S}_{tr}$.

Definition 3.5.8 (Synchronised Stream)**.** A channel $R_i \rightarrow R_j$ is a synchronised stream channel if the relation $\mathcal{S}^1_{tr} \upharpoonright (R_i \rightarrow R_j)$ is left-total:

$$
\begin{array}{l}
\mathit{SyncStream}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \forall c_B : \mathit{Connection} \cdot \exists c_A : \mathit{Connection} \cdot (c_B, c_A) \in \mathcal{S}^1_{tr} \upharpoonright (R_i \rightarrow R_j)\,.
\end{array}
$$

Definition 3.5.9 (Injective Synchronised Stream)**.** A channel $R_i \rightarrow R_j$ is an injective synchronised stream channel if it is a synchronised stream channel and there exists an injective maximal functional refinement $\mathcal{S}'$ of the relation $\mathcal{S}^1_{tr} \upharpoonright (R_i \rightarrow R_j)$:

$$\begin{aligned}
&\mathit{InjectiveSyncStream}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
&\quad \mathit{SyncStream}(R_i \rightarrow R_j)(tr) \wedge \\
&\quad \exists \mathcal{S}' \ll \mathcal{S}^1_{tr} \upharpoonright (R_i \rightarrow R_j) \cdot \\
&\qquad \forall c_A, c_B, c_{B'} : \mathit{Connection} \cdot c_B \mathcal{S}' c_A \wedge c_{B'} \mathcal{S}' c_A \Rightarrow c_B = c_{B'} \,.
\end{aligned}$$

Definition 3.5.10 (Strong Synchronised Stream)**.** A channel $R_i \rightarrow R_j$ is a strong synchronised stream channel if the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ are synchronised stream channels and there exists a symmetric maximal functional refinement $\mathcal{S}'$ of the relation $\mathcal{S}^1_{tr} \upharpoonright \{R_i, R_j\}$:

$$\begin{aligned}
&\mathit{StrongSyncStream}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
&\quad \mathit{SyncStream}(R_i \rightarrow R_j)(tr) \wedge \mathit{SyncStream}(R_j \rightarrow R_i)(tr) \wedge \\
&\quad \exists \mathcal{S}' \ll \mathcal{S}^1_{tr} \upharpoonright \{R_i, R_j\} \cdot \forall c_A, c_B : \mathit{Connection} \cdot c_B \mathcal{S}' c_A \Rightarrow c_A \mathcal{S}' c_B \,.
\end{aligned}$$

3.5.4 Mutual stream channels

A strong stream channel or a strong synchronised stream channel between two agents establishes an unpermutable stream of messages in each direction. However, because the intruder can delay each of the streams separately, the agents participating in the session may have different views of the overall stream of messages that they have exchanged. For example, consider the following trace (shown diagrammatically in Figure 3.6 and in trace form below):

$$\begin{aligned}
tr \mathrel{\widehat{=}} \langle\, &\mathit{send}.A.c_A.B.m_1, \mathit{send}.B.c_B.A.m_2, \mathit{receive}.A.c_A.B.m_2, \\
&\mathit{send}.A.c_A.B.m_3, \mathit{receive}.B.c_B.A.m_1, \mathit{send}.B.c_B.A.m_4, \\
&\mathit{receive}.A.c_A.B.m_4, \mathit{receive}.B.c_B.A.m_3 \,\rangle \,.
\end{aligned}$$

In this trace agent A believes that the stream of messages she exchanged with agent B was $\langle m_1, m_2, m_3, m_4 \rangle$ while agent B believes that it was $\langle m_2, m_1, m_4, m_3 \rangle$.

We define stronger mutual stream channels by extending the definitions of the strong stream and strong synchronised stream properties.

Definition 3.5.11 (Mutual Stream)**.** The channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ form a mutual stream channel if the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ are strong stream channels and, between the time that an agent playing role R_i (or R_j) sends a message and the agent playing role R_j (or R_i) receives that

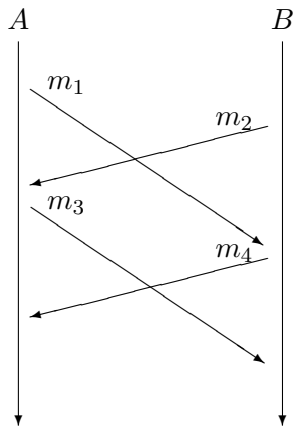


Figure 3.6: The agents A and B have different views of the overall stream of messages they have exchanged.

message, the receiving agent cannot send messages:

$$
\begin{array}{l}
\mathit{MutualStream}(R_i \leftrightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \mathit{Stream}(R_i \rightarrow R_j)(tr) \wedge \mathit{Stream}(R_j \rightarrow R_i)(tr) \wedge \\
\quad \exists \mathcal{S}' \ll \mathcal{S}_{tr} \restriction \{R_i, R_j\} \cdot \forall c_A, c_B : \mathit{Connection} \cdot \\
\qquad c_B \mathcal{S}' c_A \Rightarrow c_A \mathcal{S}' c_B \wedge \\
\qquad \forall A, A' : \hat{R}_i; B, B' : \hat{R}_j; m : \mathit{Message}_{App}; tr', tr'' : \mathit{Trace} \cdot \\
\qquad\quad (c_B \mathcal{S}' c_A \wedge tr' \frown \langle \mathit{send}.A.c_A.B'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \restriction \mathit{receive}.B.c_B.A'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.B.c_B.A' = \langle\rangle \wedge \\
\qquad\quad (c_A \mathcal{S}' c_B \wedge tr' \frown \langle \mathit{send}.B.c_B.A'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \restriction \mathit{receive}.A.c_A.B'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.A.c_A.B' = \langle\rangle \,.
\end{array}
$$

Definition 3.5.12 (Mutual Synchronised Stream)**.** The channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ form a mutual synchronised stream channel if the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ are strong synchronised stream channels and, between the time that an agent playing role R_i (or R_j) sends a message and the agent playing role R_j (or R_i) receives that message, the receiving agent cannot send messages:

$$
\begin{array}{l}
\mathit{SyncMutualStream}(R_i \leftrightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \mathit{SyncStream}(R_i \rightarrow R_j)(tr) \wedge \mathit{SyncStream}(R_j \rightarrow R_i)(tr) \wedge \\
\quad \exists \mathcal{S}' \ll \mathcal{S}^1_{tr} \restriction \{R_i, R_j\} \cdot \forall c_A, c_B : \mathit{Connection} \cdot \\
\qquad c_B \mathcal{S}' c_A \Rightarrow c_A \mathcal{S}' c_B \wedge \\
\qquad \forall A, A' : \hat{R}_i; B, B' : \hat{R}_j; m : \mathit{Message}_{App}; tr', tr'' : \mathit{Trace} \cdot \\
\qquad\quad (c_B \mathcal{S}' c_A \wedge tr' \frown \langle \mathit{send}.A.c_A.B'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \restriction \mathit{receive}.B.c_B.A'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.B.c_B.A' = \langle\rangle \wedge \\
\qquad\quad (c_A \mathcal{S}' c_B \wedge tr' \frown \langle \mathit{send}.B.c_B.A'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \restriction \mathit{receive}.A.c_A.B'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.A.c_A.B' = \langle\rangle \,.
\end{array}
$$

The mutual stream properties prevent both agents from trying to send messages at the same time; in order to send a message an agent must wait until they have received all the messages sent by the other agent. This property ensures that the two agents' views of the stream of messages exchanged in the session are the same.[8] This property could be implemented by the agents sharing sequence numbers (i.e. rather than having different sequence numbers, as in TLS, the channel could have a single shared sequence number). This property could also be implemented by the agents passing a token between one another: only the agent currently holding the token can send messages.

A mutual synchronised stream channel guarantees that both agents have the same view of the stream of messages they have exchanged, and it also guarantees that there is at most one message in the mutual stream that has been sent but not received. However, it allows the sending agent to send one or more messages without waiting for a response. Our final stream channel prevents this: it insists that the agents take turns to send and receive messages.

Definition 3.5.13 (Alternating Stream)**.** The channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ form an alternating stream channel if the channels $R_i \rightarrow R_j$ and $R_j \rightarrow R_i$ are strong synchronised stream channels and the agents playing roles R_i and R_j take turns to send and receive messages:

$$
\begin{array}{l}
\mathit{AltStream}(R_i \leftrightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \mathit{SyncStream}(R_i \rightarrow R_j)(tr) \wedge \mathit{SyncStream}(R_j \rightarrow R_i)(tr) \wedge \\
\quad \exists \mathcal{S}' \ll \mathcal{S}^1_{tr} \upharpoonright \{R_i, R_j\} \cdot \forall c_A, c_B : \mathit{Connection} \cdot \\
\qquad c_B \mathcal{S}' c_A \Rightarrow c_A \mathcal{S}' c_B \wedge \\
\qquad \forall A, A' : \hat{R}_i; B, B' : \hat{R}_j; m : \mathit{Message}_{App}; tr', tr'' : \mathit{Trace} \cdot \\
\qquad\quad (c_B \mathcal{S}' c_A \wedge tr' \frown \langle \mathit{send}.A.c_A.B'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \upharpoonright \mathit{receive}.B.c_B.A'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.B.c_B.A' = \langle\rangle \wedge \\
\qquad\quad (c_B \mathcal{S}' c_A \wedge tr' \frown \langle \mathit{receive}.B.c_B.A'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \upharpoonright \mathit{send}.B.c_B.A'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.A.c_A.B' = \langle\rangle \wedge \\
\qquad\quad (c_A \mathcal{S}' c_B \wedge tr' \frown \langle \mathit{send}.B.c_B.A'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \upharpoonright \mathit{receive}.A.c_A.B'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.A.c_A.B' = \langle\rangle \wedge \\
\qquad\quad (c_A \mathcal{S}' c_B \wedge tr' \frown \langle \mathit{receive}.A.c_A.B'.m \rangle \frown tr'' \leqslant tr \wedge \\
\qquad\qquad tr'' \upharpoonright \mathit{send}.A.c_A.B'.m = \langle\rangle) \Rightarrow tr'' \downarrow \mathit{send}.B.c_B.A' = \langle\rangle \,.
\end{array}
$$

An alternating stream channel could be implemented in a similar way to the mutual stream properties by passing a token between the agents; in this case the token has a strict alternating semantic and can be passed implicitly with each message. This property is often enforced at the application layer by the sorts of protocols that we study where agents only send one message at a time, and do not send their next message until they receive a response.

[8] This is subject to the restriction that the intruder may have delayed or blocked the stream, so one agent's view of the stream may be a prefix of the other agent's view.

These strong stream properties share some similarities with other authors' definitions of authentication properties. A suitably authenticated alternating stream channel only allows message flows that satisfy Bellare and Rogaway's definition of matching conversations [BR93], while the slightly weaker mutual stream property conforms with Bird et al.'s notion of matching histories [BGH+91]. The property of matching runs defined by Diffie, Oorschott and Weiner [DOW92] corresponds with the symmetric stream property, but the synchronised symmetric stream and the mutual and alternating stream properties are stronger than matching runs. For example, the following trace has matching runs for agents A and B, but it does not satisfy any of these stream properties:

$$tr \mathrel{\widehat{=}} \langle send.A.c_A.B.m_1, send.A.c_A.B.m_2, send.B.c_B.A.m_3, \\ receive.A.c_A.B.m_3, receive.B.c_B.A.m_1, receive.B.c_B.A.m_2 \rangle .$$

The hierarchy of session channels is shown in Figure 3.7.

3.6 Conclusions

In this chapter we have presented a hierarchy of secure channel properties. We described a system comprising a set of agents communicating over an insecure network which is controlled by a Dolev-Yao style active intruder. The honest agents can send and receive messages, while the intruder can send, receive, fake and hijack messages. The intruder can use these events to inject new messages into the network and to re-ascribe, redirect and replay existing messages.

The intruder is constrained only by rules that prevent him from performing impossible events: he can only hijack messages that have already been sent, and he can only send and fake messages that he knows. We defined the set of *ValidSystemTraces*: the set of all possible traces that the honest agents and the intruder can perform, and we defined a channel specification as a predicate over *ValidSystemTraces* under some initial value of the intruder's knowledge (*IIK*).

We specified confidential channels that prevent the intruder from overhearing messages, and we described a necessary condition for a secure transport protocol to establish confidential channels: the intruder must only be able to learn messages that were sent to him, or that were sent on non-confidential channels.

We described several dishonest events that the intruder can perform (e.g. honest re-ascribing), and we presented security specifications that prevent the intruder from performing these events. We investigated the combinations of these specifications and we found that several of them collapse: they allow behaviours that simulate events that they block. We presented five collapsing rules, and, having taken these rules into account, we described a hierarchy of eleven confidential and authenticated channel specifications.

We described several of these specifications in more detail, and in every case we presented a simple protocol that satisfies the specification. We have not proved that these protocols satisfy the specifications, however, for these simple examples the proofs appear to be straightforward.

We also investigated channel specifications that group several messages together into a session. We specified injective and symmetric forms of the session property (which respectively prevent sessions from being replayed and prevent sessions from being interleaved). We described even stronger guarantees that secure channels can provide such as the stream property (that messages are received in the same order that they were sent), the synchronised stream property (that the intruder can only delay a small finite number of messages) and the mutual stream property (that the two agents communicating in a symmetric session have the same view of the messages they have exchanged).

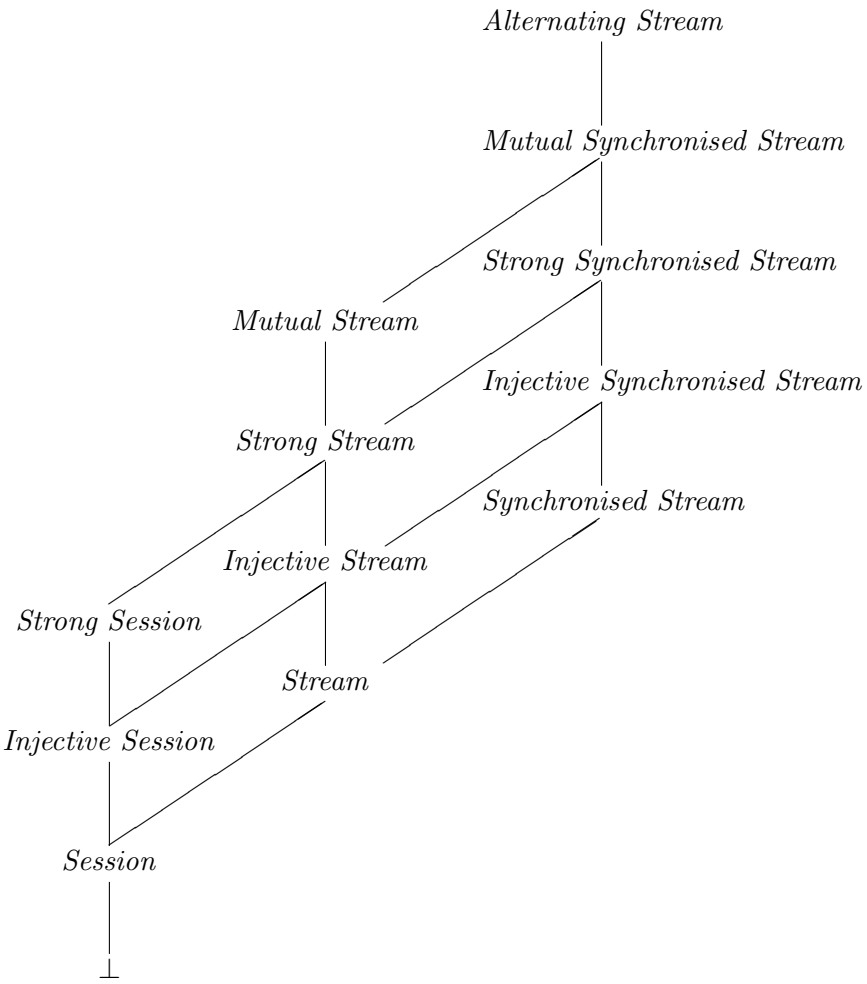


Figure 3.7: The session and stream channel hierarchy.

Chapter 4

Using secure channels

In Chapter 3 we defined a framework for specifying secure channel properties; we set out a hierarchy of 11 confidential and authenticated channel properties, and a hierarchy of 13 independent session and stream properties. In this chapter we describe several useful results about the secure channel properties and the hierarchy.

In Section 4.1 we define a simulation relation on specifications of secure channels. The relation is based on the traces of specifications as they are viewed by the honest agents; it examines the results of the intruder's behaviour, and ignores the differences in the way that these results can be achieved. We use the simulation relation to define an equivalence relation. The simulation relation generalises the implication of specifications property that the hierarchy is based on; if one channel is above another in the hierarchy then the lower channel simulates (allows more attacks than) the higher channel.

In Section 4.2 we use the equivalence relation defined in Section 4.1 to prove the equivalence of an alternative form for each of our channel specifications. Rather than blocking the intruder's behaviour, the alternative form of each specification describes the necessary behaviour that must precede a receive event. The alternative forms of channel specifications are useful for proving properties of our systems, and also provide a different, and informative, perspective on the specifications.

In Section 4.3 we show that every point in the lattice of combinations of the building blocks is either a channel in the hierarchy, or collapses to a unique point in the hierarchy.

In Section 4.4 we prove that some combinations of events that the intruder performs can safely be blocked. The set of honest traces of a channel specification when we block these sets of events is equal to the set of honest traces when we do not block the events. We use the propositions in this section in Chapters 5 and 6 to simplify the proofs. In Section 4.5 we define a sufficient condition on an application-layer security protocol for strong ses-

sion channels to be used instead of strong stream channels. Any application-layer protocol that satisfies this condition also enforces the mutual stream property when strong session channels are used. We use this property again in Chapter 6 to show that protocols can be modelled in Casper using session channels instead of stream channels without introducing false attacks.

In Section 4.6 we discuss alternative approaches to specifying secure channels, and several pieces of related work, and finally, in Section 4.7 we conclude and summarise our findings.

4.1 Simulation

In order to compare the relative strengths of different channels, we need to compare the effect they have on the intruder's capabilities. In particular, we want to check that when the intruder can perform a dishonest activity in two different ways the resulting channels are equivalent. In this section we present a simulation relation that compares channel specifications by comparing the honest agents' views of them. We justify this definition, and use it to establish an equivalence relation (simulation in both directions).

Process simulation is usually defined in terms of a simulation relation between the states of processes. However, we use the term simulation to mean something different: we want to capture the notion that one channel allows the same attacks as another. If specification P simulates specification Q, then P allows every attack that Q allows; in other words, P is no more secure than Q.

The usual preorder defined on processes over the traces model of CSP is trace refinement; process P is refined by process Q if every behaviour of Q is also a behaviour of P:

$$P \sqsubseteq_T Q \mathrel{\widehat{=}} \mathit{traces}(Q) \subseteq \mathit{traces}(P)\,.$$

Trace refinement on our systems would capture too much information for the simulation relation discussed above. For example, a channel on which the intruder cannot perform the fake event, but can hijack and re-ascribe his own messages should certainly simulate a nearly identical channel on which the intruder can fake messages. However, there are traces that the latter specification accepts that the former does not (e.g. any trace that contains a fake event). In order to draw the correct conclusion about these two specifications we need to look at the results of the intruder's behaviour, and not the way in which he performs it. The following two partial traces have the same result: agent B receives a message from agent A that A did not send:

$$\begin{aligned} tr_1 &\mathrel{\widehat{=}} \langle \mathit{fake}.A.B.c_B.m, \mathit{receive}.B.c_B.A.m \rangle\,, \\ tr_2 &\mathrel{\widehat{=}} \langle \mathit{send}.I.c_I.B.m, \mathit{hijack}.I \rightarrow A.B.c_B.m, \mathit{receive}.B.c_B.A.m \rangle\,. \end{aligned}$$

This example shows that we cannot just compare the traces of two systems; rather, we must compare the honest agents' views of the traces. In tr_1, and tr_2 the honest agent B's perspective is the same: he has received a message (m) from agent A. We are not concerned that, according to the honest agent A, B should not have received the message, because in both systems A's perspective is the same: nothing happened.

This way of looking at a system shares much with Roscoe's intensional specifications [Ros96]: the intensional specification of a protocol only looks at the events performed by the honest agents, and ignores those performed by the intruder. By examining only the events performed by the honest agents, we abstract away from the intruder's behaviour: we only see the effect of the events that the intruder performs. We use the honest traces of a system in a different way to the intensional specification: we compare the honest agents' versions of events in two different systems, both of which might allow the intruder to perform some dishonest activity, and test whether the activity of one system is a subset of that in the other. We believe that the intensional specification of a security protocol can be expressed using our simulation relation; we describe how this can be done after we introduce the relation.

Definition 4.1.1. The honest agents' view of a trace is the restriction of that trace to the application-layer send and receive events that the honest agents perform:

$$HonestTrace(tr) \mathrel{\widehat=} tr \upharpoonright \{\!| send.Honest, receive.Honest |\!\}\,.$$

The honest agents' view of a channel specification is their view of the traces of that specification:

$$\begin{aligned} &HonestTraces_{IIK}(ChannelSpec) \mathrel{\widehat=} \\ &\quad \{HonestTrace(tr) \mid tr \in traces_{IIK}(ChannelSpec)\}\,. \end{aligned}$$

Definition 4.1.2 (Simulation)**.** The channel specification $ChannelSpec_1$ simulates $ChannelSpec_2$ if, for all possible values of the intruder's initial knowledge, every trace of the second specification corresponds to a trace of the first specification that appears the same to the honest agents:

$$\begin{aligned} &\forall IIK \subseteq Message \cdot \\ &\quad HonestTraces_{IIK}(ChannelSpec_2) \subseteq HonestTraces_{IIK}(ChannelSpec_1)\,. \end{aligned}$$

When one specification simulates another we write:

$$ChannelSpec_1 \preccurlyeq ChannelSpec_2\,.$$

The definition of simulation can also be thought of in the following way: for all CSP processes $SYSTEM_1$ and $SYSTEM_2$ that satisfy $ChannelSpec_1$ and $ChannelSpec_2$ (respectively):

$$SYSTEM_1 \setminus X \sqsubseteq_T SYSTEM_2 \setminus X\,,$$

where:

$$X = \Sigma \setminus \{|\, send.Honest, receive.Honest \,|\}\,.$$

If $Spec_1 \preccurlyeq Spec_2$ we claim that the intruder can perform any attack on the first specification that he can on the second (i.e. the first specification is no more secure than the second). This is clearly true for those attacks that can be detected by looking at the honest traces. If this were not the case, then there would be a trace of the second specification that did not have a counterpart in the first. The result is not so clear for attacks that cannot immediately be detected by looking at the honest traces; in particular, in order to detect attacks against confidentiality we must examine the intruder's knowledge after traces of the specifications. We expect that if there is a fact f that the intruder can learn under a specification (either by performing a legitimate protocol run with another agent, or by learning a secret), then he should be able to learn that fact in any specification that simulates the first.

Lemma 4.1.3. *If* $tr_1 \upharpoonright \{|\, send.Honest \,|\} = tr_2 \upharpoonright \{|\, send.Honest \,|\}$*, for traces* tr_1 *and* tr_2 *of two channel specifications* $ChannelSpec_1$ *and* $ChannelSpec_2$ *under some sets of initial knowledge* IIK_1 *and* IIK_2 *such that* $IIK_2 \subseteq IIK_1$*, then* $IntruderKnows_{IIK_2}(tr_2) \subseteq IntruderKnows_{IIK_1}(tr_1)$*.*

Proof. First we restate the definition of *IntruderKnows*:

$$IntruderKnows_{IIK}(tr) \mathrel{\widehat{=}} \{m \mid (IIK \cup SentToIntruder(tr) \cup SentOnNonConfidential(tr)) \vdash m\}\,.$$

We note that the *IntruderKnows* function only depends on the intruder's initial knowledge, and on the send events in a trace. It is clear that, given *IIK*, the result of the function can only depend on $tr \upharpoonright \{|\, send \,|\}$, so we must show that it only depends on the send events performed by the *honest* agents.

SentToIntruder cannot depend on the send events performed by the intruder, as the intruder never sends messages to himself ($\mathcal{N}_1$). If, under some initial knowledge *IIK*, $tr = tr' \frown \langle send.(I, R_i).c_I.(A, R_j).m \rangle$ for dishonest I, honest A and a non-confidential channel $R_i \rightarrow R_j$, then it must be that $m \in IntruderKnows_{IIK}(tr')$ because the intruder can only send messages he knows ($\mathcal{N}_3$), so certainly $m \in IntruderKnows_{IIK}(tr)$ (as the function is monotonic). So *SentOnNonConfidential* cannot depend on the send events performed by the intruder. Hence, if:

$$tr_1 \upharpoonright \{|\, send.Honest \,|\} = tr_2 \upharpoonright \{|\, send.Honest \,|\}\,,$$

then:

$$SentToIntruder(tr_2) \cup SentOnNonConfidential(tr_2) = SentToIntruder(tr_1) \cup SentOnNonConfidential(tr_1)\,.$$

Therefore, since $IIK_2 \subseteq IIK_1$:

$$IIK_2 \cup SentToIntruder(tr_2) \cup SentOnNonConfidential(tr_2) \subseteq IIK_1 \cup SentToIntruder(tr_1) \cup SentOnNonConfidential(tr_1)\,.$$

The result follows from the monotonicity of the $\vdash$ relation. □

Corollary 4.1.4. *If* $ChannelSpec_1 \preccurlyeq ChannelSpec_2$ *then:*

$$\forall IIK_1, IIK_2 \subseteq Message \cdot IIK_2 \subseteq IIK_1 \cdot \forall tr_2 \in traces_{IIK_2}(ChannelSpec_2) \cdot \exists tr_1 \in traces_{IIK_1}(ChannelSpec_1) \cdot HonestTrace(tr_2) = HonestTrace(tr_1) \wedge IntruderKnows(tr_2) \subseteq IntruderKnows(tr_1)\,.$$

We define our equivalence relation as simulation in both directions.

Definition 4.1.5 (Equivalence)**.** Channel specifications $ChannelSpec_1$ and $ChannelSpec_2$ are equivalent if they simulate each other:

$$\forall IIK \subseteq Message \cdot HonestTraces_{IIK}(ChannelSpec_1) = HonestTraces_{IIK}(ChannelSpec_2)\,.$$

We write $ChannelSpec_1 \cong ChannelSpec_2$.

Lemma 4.1.6. $\cong$ *is an equivalence relation.*

The intruder has exactly the same capabilities in any two equivalent systems: he can perform the same attacks in both, and there is no fact that he can learn in one but not in the other. We use the simulation and equivalence relations presented in this section later when we demonstrate alternative formulations of our channel specifications, and when we prove results about networks of secure channels.

We claimed earlier that the intensional specification of a security protocol can be captured by our simulation relation. The intensional specification of a security protocol is the requirement that when two or more regular agents[1] run the protocol in the presence of an active attacker, the messages are exchanged in the order specified by the protocol, and the regular agents agree on all of the values in all of the messages in every possible trace. This is the requirement that:

$$SPEC \preccurlyeq SYSTEM\,,$$

where *SYSTEM* is a CSP process in which the regular agents run the protocol in the presence of the intruder, and *SPEC* is a CSP process that only allows runs of the protocol according to the protocol's design. We conjecture that *SPEC* can be modelled by requiring every pair of agents who communicate in the protocol to do so over the top channel in our hierarchy.

[1]The regular agents are the honest agents who participate in the protocol directly; trusted third parties, such as servers, are not regular.

4.2 Alternative channel specifications

We specified our channels by blocking the dishonest events that the intruder can perform. Specifying the channels in this way gives a simple set of definitions; once the intruder's initial powers are understood, it is easy to see the restrictions that are created by blocking, or limiting, his use of one of the dishonest events. However, the specifications are not particularly useful for proving properties about systems. In this section we give alternative formulations for our channel specifications; these alternatives state exactly which events must have occurred before an honest agent can receive a message, and are more conducive to proving properties about our networks.

Network rule ($\mathcal{N}_4$) states the necessary events that must have happened before an honest agent can receive a message; we restate this condition below (just for the channel $R_i \rightarrow R_j$, rather than for the whole network) as it forms the base from which we derive alternative specifications for our channels:

$$
\begin{array}{l}
\mathcal{N}_4(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \forall B : (Honest, R_j); c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
\qquad receive.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
\qquad\quad fake.A.B.c_B.m \ \mathbf{in}\ tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \rightarrow A.B' \rightarrow B.c_B.m \ \mathbf{in}\ tr\,.
\end{array}
$$

When an honest agent B receives message m, apparently from agent A, then either A really sent that message to B, the intruder faked the message or the intruder has hijacked a message and caused B to receive it from A. In the latter case the intruder may have changed the sender's identity, the receiver's identity, or both identities; he may also just have replayed the message (i.e. changed neither of the identities). None of our basic authentication channels prevent messages from being replayed, but the strongest channel ($C \wedge NF \wedge NRA \wedge NR$) prevents all other hijack events, so this channel satisfies a stronger form of $\mathcal{N}_4$ in which the only possibility is that A really did send the message to B:

$$
\begin{array}{l}
StrongAuth(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \forall B : (Honest, R_j); c_B : Connection; A : (Agent, R_i); m : Message_{App} \cdot \\
\qquad receive.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \exists c_A : Connection \cdot send.A.c_A.B.m \ \mathbf{in}\ tr\,.
\end{array}
$$

In this case, it was obvious how to form the alternative specification for the channel: none of the dishonest events is allowed (except a replay), so none of them could have been the cause of the receive event. We note that this alternative form of the specification does not prevent the intruder from performing dishonest events on the channel; however, any dishonest event that he does perform cannot cause an honest agent to receive a message that they would not otherwise have received (as there must also be a send

event), but it does allow the intruder to replay a message so that an honest agent receives it again, or in a different connection. We show that these restrictions are equivalent (with respect to $\cong$) below.

The alternative specification for any combination of the channel primitives is formed as below:

- The alternative form of no-faking (NF) is formed by removing the $fake.A.c_A.B.m$ possibility from $\mathcal{N}_4$; if fake events are not allowed then there must have been some other event that caused the receive event to happen;
- The alternative form of no-re-ascribing (NRA) must not allow message re-ascribing: if the receive event was caused by $hijack.A' \rightarrow A.c_A.B' \rightarrow B.c_B.m$ then $A = A'$;
- The alternative form of no-honest-re-ascribing (NRA^-) must restrict the possibilities for message re-ascribing: if the receive event was caused by $hijack.A' \rightarrow A.B' \rightarrow B.c_B.m$ then $A = A'$ or A must be dishonest;
- The alternative form of no-redirecting (NR) must not allow message redirection: if the receive event was caused by $hijack.A' \rightarrow A.B' \rightarrow B.c_B.m$ then $B' = B$;
- The alternative form of no-honest-redirecting (NR^-) must restrict the possibilities for messages redirection: if the receive event was caused by $hijack.A' \rightarrow A.B' \rightarrow B.c_B.m$ then $B' = B$ or B' must be dishonest.

We give the alternative forms for each of our channels (rather than just specifying the alternative form of a specification as the conjunction of the alternative forms of the primitives that make up the specification), because the alternative forms of the primitives (as described above) do not conjoin in the way that we require them to (we discuss this, and other possible solutions to the problem in Section 4.6). The alternative form of each of the specifications in our hierarchy is given in Appendix A.

One can prove that each of the alternative forms is equivalent to the corresponding original specification. We demonstrate the proof technique by showing the equivalence of the following channel specification:

$$
\begin{array}{l}
(NF \wedge NRA^-)(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad tr \downarrow \{| \, fake.\hat{R}_i.\hat{R}_j, \\
\qquad hijack.A \rightarrow A'.\hat{R}_j \rightarrow \hat{R}_j \mid A, A' : \hat{R}_i \cdot A \neq A' \wedge Honest(A') \, |\} = \langle\rangle \, ,
\end{array}
$$

and its alternative form:

$$
\begin{array}{l}
\mathit{SenderAuth}(R_i \rightarrow R_j)(tr) \mathrel{\widehat{=}} \\
\quad \forall B : (\mathit{Honest}, R_j); c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{\mathit{App}} \cdot \\
\qquad \mathit{receive}.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
\qquad\quad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot \mathit{hijack}.A' \rightarrow A.B' \rightarrow B.c_B.m \ \mathbf{in}\ tr \wedge \\
\qquad\qquad \mathit{Dishonest}(A) \vee A = A' \,.
\end{array}
$$

Theorem 4.2.1. *The following specifications are equivalent:*

$$
\begin{array}{l}
\mathit{ChannelSpec}_1 = \mathit{ChannelSpec} \wedge \mathit{SenderAuth}(R_i \rightarrow R_j) \\
\mathit{ChannelSpec}_2 = \mathit{ChannelSpec} \wedge (\mathit{NF} \wedge \mathit{NRA}^-)(R_i \rightarrow R_j)
\end{array}
$$

where ChannelSpec is any channel specification.

Proof. First, we consider an honest trace of the second specification, and show that it is also an honest trace of the first specification (in order to establish $\mathit{ChannelSpec}_1 \preccurlyeq \mathit{ChannelSpec}_2$). Throughout this proof we assume that the intruder's initial knowledge is fixed (and hence we omit it from the mathematics). Let $tr_2 \in \mathit{HonestTraces}(\mathit{ChannelSpec}_2)$, and consider a trace tr of that system that looks like tr_2 to the honest agents, i.e.:

$$tr \in \mathit{traces}(\mathit{ChannelSpec}_2) \wedge \mathit{HonestTrace}(tr) = tr_2 \,.$$

As both $\mathcal{N}_4$ and $\mathit{NF} \wedge \mathit{NRA}^-$ are satisfied on $R_i \rightarrow R_j$ by the second system, they both hold for this trace:

$$\mathcal{N}_4(R_i \rightarrow R_j)(tr) \wedge (\mathit{NF} \wedge \mathit{NRA}^-)(R_i \rightarrow R_j)(tr) \,.$$

It is clear that $\mathit{SenderAuth}(R_i \rightarrow R_j)(tr)$ since:

$$(\mathcal{N}_4 \wedge \mathit{NF} \wedge \mathit{NRA}^-) \Rightarrow \mathit{SenderAuth} \,,$$

and so because $\mathit{ChannelSpec}_1$ and $\mathit{ChannelSpec}_2$ imply identical constraints on all the other channels, tr must also be accepted by the first specification. Hence $tr_2 = \mathit{HonestTrace}(tr) \in \mathit{HonestTraces}(\mathit{ChannelSpec}_1)$. Therefore:

$$\mathit{HonestTraces}(\mathit{ChannelSpec}_2) \subseteq \mathit{HonestTraces}(\mathit{ChannelSpec}_1) \,,$$

and so $\mathit{ChannelSpec}_1 \preccurlyeq \mathit{ChannelSpec}_2$.

Conversely, we consider an honest trace of the first specification, and show that it is also an honest trace of the second specification (in order to prove that $\mathit{ChannelSpec}_2 \preccurlyeq \mathit{ChannelSpec}_1$). Let $tr_1 \in \mathit{HonestTraces}(\mathit{ChannelSpec}_1)$, and consider a trace tr of that specification that looks like tr_1 to the honest agents:

$$tr \in \mathit{traces}(\mathit{ChannelSpec}_1) \wedge \mathit{HonestTrace}(tr) = tr_1 \,.$$

If $tr \in traces(ChannelSpec_2)$ then the second specification accepts a trace that looks like tr_1 to the honest agents, so the result is immediate. Suppose that this is not the case: it must be because of events on the channel $R_i \rightarrow R_j$ that $ChannelSpec_2$ does not accept tr, because the specifications satisfy identical constraints on all the other channels. Further, since the only channel specification that $ChannelSpec_2$ includes that $ChannelSpec_1$ does not is $(NF \wedge NRA^-)(R_i \rightarrow R_j)$, it must be that $\neg(NF \wedge NRA^-)(R_i \rightarrow R_j)(tr)$.

Consider the trace formed by removing all of the fake events and honest-re-ascribing hijack events on the channel $R_i \rightarrow R_j$ from the trace tr:

$$\bar{tr} \mathrel{\widehat{=}} tr \upharpoonright \Sigma \setminus \{| fake.\hat{R}_i.\hat{R}_j, \\ hijack.A \rightarrow A'.\hat{R}_j \rightarrow \hat{R}_j \mid A, A' : \hat{R}_i \cdot A \neq A' \wedge Honest(A') |\} .$$

We claim that $\bar{tr}$ is a trace of the second specification.

By removing events from tr we might have affected whether $\mathcal{N}_4$ holds on the channel $R_i \rightarrow R_j$ for the trace $\bar{tr}$; if a $receive.B.c_B.A.m$ event occurred in tr because of a previous $fake.A.B.c_B.m$ or $hijack.A' \rightarrow A.B' \rightarrow B.c_B.m$ event, then this receive event may now be unfounded. We show that this cannot be the case. Suppose that there are traces tr' and tr'' such that:

$$tr' \frown \langle fake.A.B.c_B.m \rangle \frown tr'' \frown \langle receive.B.c_B.A.m \rangle \leqslant tr ,$$

or:

$$tr' \frown \langle hijack.A' \rightarrow A.B' \rightarrow B.c_B.m \rangle \frown tr'' \frown \langle receive.B.c_B.A.m \rangle \leqslant tr ,$$

where $A \neq A'$ and A is honest. Since $SenderAuth(R_i \rightarrow R_j)(tr)$, one of the following events must occur in $tr' \frown tr''$:

- $send.A.c_A.B.m$ (for some connection identifier c_A);
- $hijack.A' \rightarrow A.B' \rightarrow B.c_B.m$ (for some agents A' and B' such that $A = A'$ or A' is dishonest).

This argument holds for every receive event on the channel $R_i \rightarrow R_j$ in $\bar{tr}$, and so it follows that $\mathcal{N}_4(R_i \rightarrow R_j)$ is satisfied by $\bar{tr}$. Because $ChannelSpec_1$ and $ChannelSpec_2$ satisfy identical constraints on all the channels except $R_i \rightarrow R_j$, and we have ensured that $NF(R_i \rightarrow R_j) \wedge NRA^-(R_i \rightarrow R_j)(\bar{tr})$, $\bar{tr}$ must be accepted by the second specification.

Since we have only removed dishonest events from tr, it is clear that $HonestTrace(\bar{tr}) = tr_1$. Hence $tr_1 \in HonestTraces(ChannelSpec_2)$, and therefore:

$$HonestTraces(ChannelSpec_1) \subseteq HonestTraces(ChannelSpec_2),$$

so $ChannelSpec_2 \preccurlyeq ChannelSpec_1$, and $ChannelSpec_1 \cong ChannelSpec_2$. $\square$

In the second half of the proof of Theorem 4.2.1 we showed that we can remove the dishonest events from a trace of a specification that satisfies the *SenderAuth* property while maintaining the trace's appearance to the honest agents. For every dishonest event we removed there was already a send event, or a different dishonest event in the trace that had the same effect. In fact, it was this other event that caused the receive event, and not the event that we removed. The alternative forms of the channel specifications allow the intruder to generate events that have no discernible effect.

This could be seen as a weakness of the specifications, but we can think of the concrete realisation of the redundant dishonest events in two ways:

1. The intruder gets a message from the network, and puts it back intact;
2. The intruder cannot generate a correctly formatted *put* event, so the message is rejected by the recipient's protocol agent.

The events do not contribute anything new to the trace in question, so it is safe to remove them.

We assume that the specifications we consider allow all traces that satisfy the conditions we impose on the channels. Of course, the high-level traces of the honest agents' events might be constrained further by the application-layer protocols they are running, but the results still hold in the obvious way.

We do not show any further proofs for the equivalence of channel specifications and their alternative forms as they are similar to the proof shown above. In every case the alternative specification is implied by $\mathcal{N}_4$ and the original specification, so the first simulation is straightforward. Proving the other simulation is always done in the same way: we consider an honest trace of the first specification, and consider a trace that looks like that honest trace. We show that we can remove any dishonest events from that trace that preclude it from satisfying the second specification, without preventing it from satisfying the network rules, and hence show that the honest trace we originally considered is also an honest trace of the second specification.

In the statement of Theorem 4.2.1 we assume that all the channels (other than $R_i \rightarrow R_j$) in the two systems satisfy identical conditions. Once we have shown the equivalence of the conditions applied to $R_i \rightarrow R_j$, we can consider a third specification that differs on a different channel, and prove that that specification is equivalent to the second one. By chaining the proofs in this way we can convert as many channel specifications to their alternative forms as we need to.

4.3 Uniqueness of collapse

The secure channels hierarchy is built from four primitives: confidentiality, no-faking, no-re-ascribing and no-redirecting. There are thirty-six different

ways of combining these primitives; these combinations form a lattice. In this section we prove that every point in the lattice is either a channel specification in the channel hierarchy or it collapses (using the collapsing rules $Collapse_1$–$Collapse_5$) to a unique point in the hierarchy.

We describe a point in the lattice by listing each of its components in the order (C, NF, NRA, NR); e.g. the point $(C, \bot, NRA, NR^-)$ is the channel $C \wedge NRA \wedge NR^-$. In Chapter 3 we saw that several of the points in the lattice collapse: they allow behaviour that simulates one of the properties that they block. The lattice collapses to the eleven point hierarchy described in Chapter 3.

Definition 4.3.1. Points in the lattice are compared component-wise:

$$\begin{aligned}&(c_1, nf_1, nra_1, nr_1) \leqslant (c_2, nf_2, nra_2, nr_2) \Leftrightarrow \\ &\quad c_1 \leqslant c_2 \wedge nf_1 \leqslant nf_2 \wedge nra_1 \leqslant nra_2 \wedge nr_1 \leqslant nr_2\,,\end{aligned}$$

where $\bot < C$, $\bot < NF$, $\bot < NRA^- < NRA$, and $\bot < NR^- < NR$.

The collapsing cases set out in Chapter 3 are described by five collapsing rules. These collapsing cases can be rewritten as pattern matching rules:

$$\begin{aligned}Collapse_1 &\mathrel{\widehat{=}} (\bot,\ \bot\,,\quad x\quad,\ y\ ) \downarrow (\bot,\ \bot\,,\quad \bot\quad,\ \bot\ )\,,\\ Collapse_2 &\mathrel{\widehat{=}} (x\,,\ NF,\quad \bot\quad,\ y\ ) \downarrow (x,\ \bot\,,\quad \bot\quad,\ y\ )\,,\\ Collapse_3 &\mathrel{\widehat{=}} (\bot,\ NF,\ NRA,\ \ x\ ) \downarrow (\bot,\ NF,\ NRA^-,\ \ x\ )\,,\\ Collapse_4 &\mathrel{\widehat{=}} (C,\ \ x\,,\quad y\quad,\ \bot\ ) \downarrow (\bot,\ \ x\,,\quad y\quad,\ \bot\ )\,,\\ Collapse_5 &\mathrel{\widehat{=}} (C,\ \bot\,,\quad x\quad,\ NR) \downarrow (C,\ \bot\,,\quad x\quad,\ NR^-)\,.\end{aligned}$$

For example, rule $Collapse_4$ matches any confidential point that allows redirecting; any points in the lattice that match this pattern are collapsed to a non-confidential point with the other components remaining unchanged.

Definition 4.3.2. For any point (c, nf, nra, nr) in the full lattice, we define $\downarrow (c, nf, nra, nr)$ to be the *collapsed form* of the point. This is the point we reach by continually applying the collapsing rules until we reach a point that cannot be collapsed further.

Proposition 4.3.3. *The collapsed form of every point in the full lattice is unique and well-defined.*

Proof. With the exception of $Collapse_2$ and $Collapse_4$, the patterns for the five rules above are disjoint. The point $(C, NF, \bot, \bot)$ matches the patterns of both $Collapse_2$ and $Collapse_4$, so we examine what happens when we apply $Collapse_2$ first, and when we apply $Collapse_4$ first:[2]

$$\begin{aligned}&(C, NF, \bot, \bot) \downarrow_2 (C, \bot, \bot, \bot) \downarrow_4 (\bot, \bot, \bot, \bot)\,,\\ &(C, NF, \bot, \bot) \downarrow_4 (\bot, NF, \bot, \bot) \downarrow_2 (\bot, \bot, \bot, \bot)\,.\end{aligned}$$

[2]We use $\downarrow_i$ as shorthand for applying collapsing rule $Collapse_i$.

The order in which $Collapse_2$ and $Collapse_4$ are applied makes no difference to the resultant collapsed form; in every other case there is at most one rule that can be applied (since the patterns for the rules are disjoint). Therefore the sequence of collapsing rules that can be applied to any given point is unique and well-defined, hence $\downarrow (c, nf, nra, nr)$ is unique and well-defined for any point in the lattice. □

We note that $\downarrow$ is monotonic with respect to the order on the lattice ($\leqslant$).

Lemma 4.3.4. *For any point (c, nf, nra, nr) in the lattice, $\downarrow (c, nf, nra, nr) = (c', nf', nra', nr')$ is the strongest channel such that:*

$$(c', nf', nra', nr') \leqslant (c, nf, nra, nr)\,.$$

Proof. The proof is a simple case analysis. □

Proposition 4.3.5. *For any two channels in the hierarchy $ChannelSpec_1 = (c_1, nf_1, nra_1, nr_1)$ and $ChannelSpec_2 = (c_2, nf_2, nra_2, nr_2)$:*

$$ChannelSpec_1 \preccurlyeq ChannelSpec_2 \Leftrightarrow (c_1, nf_1, nra_1, nr_1) \leqslant (c_2, nf_2, nra_2, nr_2)\,.$$

4.4 Safely blockable (simulating) activity

In this section we prove that there are some events, and some combinations of events, that the intruder can perform that we can safely block. These combinations of events have the same effect as other events (they simulate them), so we can prevent the intruder from performing these events without losing honest traces. We use the results in this section to simplify the proofs in Chapters 5 and 6.

The intruder can hijack messages to change the identity of the message sender. In Chapter 3 we defined a rule ($\mathcal{N}_2$) that says that before the intruder can perform a $hijack.A \rightarrow A'.B \rightarrow B'.c_{B'}.m$ event there must be a $send.A.c_A.B.m$ event earlier in the trace. In the formulation of $\mathcal{N}_2$ we do not require the original message sender A to be honest; in other words, the intruder can send a message himself, and then hijack it to change the sender's identity. This behaviour simulates a fake event (both combinations of events cause the honest agent B' to receive a message from A' that A' did not send).

Proposition 4.4.1. *We can safely block the intruder from hijacking his own messages; i.e. for any channel specification ChannelSpec comprised of properties from the hierarchy:*

$$ChannelSpec \cong ChannelSpec \wedge tr \upharpoonright \{|\, hijack.Dishonest \,|\} = \langle\rangle\,.$$

Proof. The proof of this proposition relies on the fact that the blocked events are equivalent to (i.e. have the same effect as) fake events. It is clear that the simulation relation holds in one direction: if

$$\mathit{ChannelSpec}(tr) \wedge tr \upharpoonright \{\!|\, \mathit{hijack.Dishonest} \,|\!\} = \langle\rangle \,,$$

holds for a trace tr then $\mathit{ChannelSpec}(tr)$ certainly holds.

In order to show the simulation relation in the other direction suppose that $\mathit{ChannelSpec}(tr)$, but $tr \upharpoonright \{\!|\, \mathit{hijack.Dishonest} \,|\!\} \neq \langle\rangle$. There must be at least one $\mathit{hijack}.A \rightarrow A'.B \rightarrow B'.c_{B'}.m$ event in tr where A is dishonest and A' is honest. Consider the trace $\bar{tr}$ where all such events are replaced by $\mathit{fake}.A'.B'.c_{B'}.m$ events.

We argue that $\bar{tr}$ is still a valid system trace. It is clear that $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_4$, $\mathcal{A}_1$ and $\mathcal{A}_2$ still hold for $\bar{tr}$. It is straightforward to show that $\mathcal{N}_3$ holds for $\bar{tr}$: suppose that there exists some prefix of $\bar{tr}$ such that:

$$tr'^{\frown}\langle \mathit{fake}.A'.B'.c_{B'}.m \rangle \leqslant \bar{tr} \,,$$

where this is one of the new fake events, and A' is honest. Since this fake event replaced a $\mathit{hijack}.A \rightarrow A'.B \rightarrow B'.c_{B'}.m$ event there must, by $\mathcal{N}_2$, be a $\mathit{send}.A.c_A.B.m$ event in tr'. Since A is dishonest this implies that:

$$m \in \mathit{IntruderKnows}_{IIK}(tr') \,,$$

so the new fake event is well-founded, and $\mathcal{N}_3(\bar{tr})$ holds.

We now argue by contradiction that $\mathit{ChannelSpec}(\bar{tr})$ still holds. Suppose that this is not the case: one of the new fake events must contradict one of the channel specifications (from the hierarchy) in *ChannelSpec*. However, every channel property that specifies no faking also specifies, at least, no-honest-re-ascribing, so $\mathit{ChannelSpec}(tr)$ does not hold. This contradicts our initial assumption, and therefore $\mathit{ChannelSpec}(\bar{tr})$ holds.

Since $\mathit{HonestTrace}(tr) = \mathit{HonestTrace}(\bar{tr})$ we conclude that the simulation relation:

$$\mathit{ChannelSpec} \wedge tr \upharpoonright \{\!|\, \mathit{hijack.Dishonest} \,|\!\} = \langle\rangle \preccurlyeq \mathit{ChannelSpec}$$

holds, and hence the two specifications are equivalent. □

The intruder can hijack a message to re-ascribe it to a dishonest agent and to redirect it from a dishonest agent to an honest agent. However, if the intruder redirects a message that was sent to him and re-ascribes it to one of his own identities then he could easily send the message, rather than performing a hijack event.

Proposition 4.4.2. *We can safely block the intruder from re-ascribing messages to dishonest agents at the same time as redirecting messages from dishonest agents; i.e. for any channel specification ChannelSpec:*

$$\begin{aligned} &\mathit{ChannelSpec} \cong \mathit{ChannelSpec} \wedge \\ &\quad tr \upharpoonright \{\!|\, \mathit{hijack.Honest} \rightarrow \mathit{Dishonest.Dishonest} \rightarrow \mathit{Honest} \,|\!\} = \langle\rangle \,. \end{aligned}$$

Proof. As in the previous proof, the first simulation is easy to show because the right hand side of the equivalence implies the left hand side. To show the other simulation, and hence equivalence, suppose that for some trace tr such that $\mathit{ChannelSpec}(tr)$ holds there are honest agents A and B and a dishonest agent I such that a 'bad' hijack event appears in tr:[3]

$$tr' \frown \langle \mathit{hijack}.A \rightarrow I.I \rightarrow B.c_B.m \rangle \leqslant tr\,.$$

Because the intruder can perform this hijack event, there must, by $\mathcal{N}_2$, be a $\mathit{send}.A.c_A.I.m$ event earlier in the trace (i.e. in tr'); it follows that $m \in \mathit{IntruderKnows}_{IIK}(tr')$. We consider the trace $\bar{tr}$ where all such hijack events are replaced by appropriate send events (e.g. $\mathit{send}.I.c_I.B.m$ events). Each of these new send events is well-founded (i.e. $\mathcal{N}_3$ still holds) because the intruder knows each message; the other network rules are unaffected by this change.

ChannelSpec is also unaffected by the change as none of the channel specifications restrict the intruder from sending messages with his own identity. Hence $\mathit{ChannelSpec}(\bar{tr})$ holds, $\mathit{HonestTrace}(tr) = \mathit{HonestTrace}(\bar{tr})$ and the simulation relation:

$$\begin{array}{l} \mathit{ChannelSpec} \wedge \\ \quad tr \upharpoonright \{\!|\, \mathit{hijack}.\mathit{Honest} \rightarrow \mathit{Dishonest}.\mathit{Dishonest} \rightarrow \mathit{Honest} \,|\!\} = \langle\rangle \preccurlyeq \\ \qquad \mathit{ChannelSpec} \end{array}$$

holds. The equivalence relation therefore holds. □

On non-confidential channels the intruder can hijack messages to re-ascribe them to himself. However, because these channels are not confidential, the intruder learns all messages that are sent on them. Rather than re-ascribing messages to himself, the intruder can learn messages and send them with his own identity.

Proposition 4.4.3. *We can safely block the intruder from re-ascribing messages to dishonest agents on non-confidential channels; i.e. for any channel specification ChannelSpec such that the channel $R_i \rightarrow R_j$ is not confidential (i.e. $C(R_i \rightarrow R_j)$ does not hold):*

$$\begin{array}{l} \mathit{ChannelSpec} \cong \mathit{ChannelSpec} \wedge \\ \quad tr \upharpoonright \{\!|\, \mathit{hijack}.(\mathit{Honest}, R_i) \rightarrow (\mathit{Dishonest}, R_i).\hat{R}_j \rightarrow \hat{R}_j \,|\!\} = \langle\rangle\,. \end{array}$$

Proof. As in the previous proofs, the first simulation is easy to show (the right hand side of the equivalence implies the left hand side). To show the other simulation, and hence equivalence, suppose that for some trace tr such that $\mathit{ChannelSpec}(tr)$ holds there is an honest agent $A : \hat{R}_i$, a dishonest

[3] There could be two dishonest agents I_1 and I_2, but the proof is the same with a single dishonest identity.

agent $I : \hat{R}_i$ and agents $B, B' : \hat{R}_j$ such that a 'bad' hijack event appears in tr:

$$tr' \frown \langle hijack.A \rightarrow I.B \rightarrow B'.c_B.m \rangle \leqslant tr\,.$$

The channel $R_i \rightarrow R_j$ is not confidential, so $m \in \mathit{IntruderKnows}(tr')$ (because there must be a send event in tr' for the intruder to be able to perform the hijack event). As before, we consider the trace $\bar{tr}$ where all such hijack events are replaced by appropriate send events (e.g. $send.I.c_I.B'.m$ events). Each of these new send events is well-founded (i.e. $\mathcal{N}_3$ still holds) because the intruder knows each message; the other network rules are unaffected by this change.

ChannelSpec is also unaffected by the change as none of the channel specifications restrict the intruder from sending messages with his own identity. Hence $\mathit{ChannelSpec}(\bar{tr})$ holds, $\mathit{HonestTrace}(tr) = \mathit{HonestTrace}(\bar{tr})$ and the simulation relation:

$$\begin{array}{l} \mathit{ChannelSpec} \wedge \\ \quad tr \upharpoonright \{\!|\, hijack.(Honest, R_i) \rightarrow (Dishonest, R_i).\hat{R}_j \rightarrow \hat{R}_j \,|\!\} = \langle\rangle \preccurlyeq \\ \quad\quad \mathit{ChannelSpec} \end{array}$$

holds. The equivalence relation therefore holds. □

4.5 Using session channels instead of streams

For many of the sorts of protocols that we are interested in studying and developing it is safe to use a session channel instead of a stream channel for communication between pairs of agents. In this section we define a sufficient condition on an application-layer protocol for session channels to be used in place of stream channels without introducing attacks against the protocol.

We consider application-layer security protocols of the following form.

Definition 4.5.1. An application-layer security protocol $\mathcal{P}$ is described by a triple $(\mathcal{R}, \mathcal{M}, \mathcal{T})$:

- $\mathcal{R}$ is the set of roles in the protocol (R_i, R_j, etc.);
- $\mathcal{M}$ is the set of messages in the protocol; each message is a pair of the form (Msg_l, M) where $l \in \mathbb{N}$ is the message number (or label), and $M \subseteq \mathit{Message}_{App}$ is the message text;
- $\mathcal{T}$ is the ordered sequence of message transmissions in the protocol; each message transmission is a triple (R_s, R_r, Msg_l) where R_s is the role of the message sender, R_r is the role of the message receiver and l is the message number.

Definition 4.5.2 (No speaking out of turn)**.** An application-layer security protocol satisfies *no speaking out of turn* if, between every pair of agents who communicate in the protocol, there is never (at any stage in the protocol) any doubt over whose turn it is to send the next message.

Proposition 4.5.3. *Any application-layer security protocol that can be expressed in the notation of Definition 4.5.1 satisfies the no speaking out of turn property.*

This proposition is due to the fact that the message transmissions in $\mathcal{T}$ have a well-defined order, so Msg_i precedes Msg_j if and only if $i \leqslant j$. If we weaken Definition 4.5.1 so that $\mathcal{T}$ only has a partial order, or so that $\mathcal{T}$ is a set of allowable sequences of message, the protocols do not necessarily satisfy *no speaking out of turn*.

We now define a property that ensures that application-protocol messages within a session between two agents cannot be mixed up or re-ordered.

Definition 4.5.4 (Disjoint messages)**.** An application-layer protocol $\mathcal{P} = (\mathcal{R}, \mathcal{M}, \mathcal{T})$ has disjoint messages if the sets of possible values of encrypted components of different messages are disjoint:[4]

$$\begin{array}{l} \forall (Msg_i, M_i), (Msg_j, M_j) \in \mathcal{M} \cdot \\ \quad \forall m_i \in EncryptedComponents(M_i); m_j \in EncryptedComponents(M_j) \cdot \\ \qquad m_i = m_j \Rightarrow i = j \,. \end{array}$$

This property may be gained by using a technique such as disjoint encryption [GF00b] for different messages, or by ensuring that the types of all messages are distinct.

Proposition 4.5.5. *For any application-layer security protocol with disjoint messages that satisfies the no speaking out of turn property, the effect of a mutual stream channel is enforced by the application-layer protocol when new symmetric session channels are used for communication between every pair of agents in every run of the protocol.*

It is clear that an injective session channel is sufficient to enforce the stream property since the messages of the protocol can only be delivered in the order specified by the protocol description, and each message can only be interpreted as the message it is intended to be (i.e. an early message could not be replayed as a later one). Each agent does not send his next message until he has received all the messages he expects to receive, and because the message transmissions have a total order, the intruder cannot re-arrange the

[4]The encrypted components of a message are the outermost encrypted elements of the message; for example, the encrypted components of the message $A, B, \{A, \{n_A\}_{PK(A)}, h(m)\}_{SK(B)}, \{B, m\}_{PK(A)}$ are $\{A, \{n_A\}_{PK(A)}, h(m)\}_{SK(B)}$ and $\{B, m\}_{PK(A)}$.

order in which messages are delivered. Further, since the session channel is injective, the intruder cannot replay old sessions. Because a new session is established for every new run of the protocol, the intruder cannot replay messages from old runs of the protocol.

The argument above proves that if a strong (symmetric) session channel is used then the strong stream property is enforced, so it remains only to show that the mutual stream property is enforced. The mutual stream property just says that each agent's view of the messages exchanged in a symmetric session matches the other agent's view. It is clear that this is the case for these protocols since, if it were not, one agent would have sent and received messages in an order other than that defined by the protocol (i.e. the order on the message transmissions in $\mathcal{T}$).

In Chapter 6 we use the *no speaking out of turn* and *disjoint messages* properties to argue that any application-layer security protocol may be modelled in Casper with session channels instead of stream channels without false attacks being introduced. We also build Casper models that include explicit message numbers so that protocols that do not satisfy these properties can still be analysed.

4.6 Related work

In this section we discuss how our approach to specifying secure channels compares with that taken by other authors.

4.6.1 Broadfoot and Lowe

In [BL03], Broadfoot and Lowe specify a form of secrecy that is slightly different to our confidential channel ($C \wedge NR^{-}$). We recall their definition below (using the notation of this thesis):

$$\begin{array}{l} \mathit{Secrecy}(tr) \mathrel{\widehat{=}} \\ \quad \forall B : \mathit{Honest}; c_B : \mathit{Connection}; I : \mathit{Dishonest}; tr' : \mathit{Trace}; m : \mathit{Message} \cdot \\ \quad\quad tr' \frown \langle \mathit{receive}.B.c_B.I.m \rangle \leqslant tr \Rightarrow \mathit{IIK} \cup \mathit{SentToIntruder}(tr') \vdash m. \end{array}$$

Note that under this definition, either every channel is confidential or no channels are confidential.

Clearly $\mathit{Secrecy} \not\preccurlyeq C \wedge NR^{-}$; consider the trace below where A and B are honest and I is dishonest:

$$tr \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.B.m, \mathit{hijack}.A \rightarrow I.B.c_B.m, \mathit{receive}.B.c_B.I.m \rangle .$$

$\mathit{Secrecy}(tr)$ does not hold for tr since B received a message from I that I does not know, but $C(\mathit{Role} \times \mathit{Role})(tr)$ does because the intruder can re-ascribe a message without learning it (or needing to learn it). The honest projection of this trace is in $\mathit{HonestTraces}_{\mathit{IIK}}(C \wedge NR^{-})$ but not

in $HonestTraces_{IIK}(Secrecy)$, so the simulation relation certainly does not hold.

The difference between our definition of confidentiality and that given in [BL03] is that we allow the intruder to change the identities of the sender and recipient of a message. In the model in which Broadfoot and Lowe's results should be interpreted, the intruder does not possess this capability (he must intercept the message, learn it, and then resend or fake it), so the definition from [BL03] ought to be compared to, and is equivalent to, non-re-ascribable confidential channels (i.e. $C \wedge NRA \wedge NR^{-}$).

It is easy to see that if every channel satisfies $C \wedge NRA \wedge NR^{-}$, then any message an honest agent receives from a dishonest agent must have been sent to him by that dishonest agent (the intruder does not fake with dishonest identities and he cannot re-ascribe messages). In order for the intruder to have sent the message, he must have known it ($\mathcal{N}_3$), and since there are no non-confidential channels, $IntruderKnows(tr) = IIK \cup SentToIntruder(tr)$. Hence *Secrecy* holds.

If we extend the definition of the *Secrecy* property (above) to include the pre-conditions when an honest agent receives a message from another honest agent, then we get the following property:

$$
\begin{array}{l}
Secrecy(tr) \mathrel{\widehat{=}} \\
\quad \forall B : Honest; c_B : Connection; A : Agent; tr' : Trace; m : Message \cdot \\
\quad\quad tr' \frown \langle receive.B.c_B.I.m \rangle \leqslant tr \Rightarrow \\
\quad\quad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr' \vee \\
\quad\quad\quad IIK \cup SentToIntruder(tr') \vdash m\,.
\end{array}
$$

Either agent A (whether honest or not) sent the message to B, or the intruder knew the message (in order to fake it). It is clear that every honest trace of the property above is also an honest trace of $C \wedge NRA \wedge NR^{-}$, since if a $send.A.c_A.B.m$ event does not appear in tr' then the receive event must have been caused by a fake event in tr' (the hijack event is not available to the intruder). Hence we conclude that $Secrecy \cong C \wedge NRA \wedge NR^{-}$.

Broadfoot and Lowe also specify a single form of authenticated channel, which is equivalent to an authenticated stream channel.

$$
\begin{array}{l}
\forall A, B : Agent; c_B : Connection \cdot \exists c_A : Connection \cdot \\
\quad tr \downarrow receive.B.c_B.A \leqslant tr \downarrow send.A.c_A.B.
\end{array}
$$

Initially, we attempted to specify our channels in the form of the secrecy property above: by listing the events that could have caused an agent to receive a message, and not having explicit fake and hijack events. For example, we devised an authentication specification:

$$
\begin{array}{l}
\forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; tr' : Trace; m : Message \cdot \\
\quad tr' \frown \langle receive.B.c_B.A \rangle \leqslant tr \Rightarrow \\
\quad\quad \exists c_A : Connection; B' : \hat{R}_j \cdot send.A.c_A.B' \textbf{ in } tr'\,,
\end{array}
$$

and a channel with guaranteed intent as:

$$\begin{array}{l}\forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; tr' : \mathit{Trace}; m : \mathit{Message} \cdot \\ \quad tr'^\frown\langle \mathit{receive}.B.c_B.A\rangle \leqslant tr \Rightarrow \\ \quad\quad \exists A' : \hat{R}_i; c_{A'} : \mathit{Connection} \cdot \mathit{send}.A'.c_{A'}.B \textbf{ in } tr'.\end{array}$$

Unfortunately, these sorts of specifications are too weak, and do not combine in the way we expect them to. We expect that on an authenticated channel with guaranteed intent (i.e. both of the above properties), a $\mathit{receive}.B.c_B.A.m$ event in a trace should guarantee the existence of a $\mathit{send}.A.c_A.B.m$ event earlier in the trace. However, a channel that satisfies the conjunction of the two properties above allows the following trace:

$$tr \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.B'.m, \mathit{send}.A'.c_{A'}.B.m, \mathit{receive}.B.c_B.A.m\rangle\ .$$

It is clear that this should not be allowed by a channel with strong authentication: the intruder has seen two agents send the same message (or perhaps sent one of them himself) and re-ascribed one, and redirected the other.

The reason that these specifications are too weak is best illustrated by this analogy to logic:

$$(p \vee q) \wedge (p \vee r) \not\equiv p \equiv (p \vee q \vee r) \wedge \neg q \wedge \neg r.$$

The specification we want for an authenticated channel can be thought of as p: the statement that whenever B receives a message from A then previously A sent that message to B. The formulation of this property on the right is what we get by initially allowing all possibilities: that A sent the message to someone else (q), or that someone else sent the message to B (r), and then specifying $\neg q$ and $\neg r$. The formulation of this property on the left corresponds to the conjunction of the properties above.

We note that although these properties have a similar feel to the alternative specifications from Section 4.2, the alternative specifications refer to fake and hijack events, as well as to send events, and so the problem mentioned above does not occur.

We decided to base our specifications on banning explicit events that the intruder performed (the fake and hijack events), however we could equally have altered our model so that each application layer message was given a unique identifier when sent: the message could then be uniquely identified in the specifications, and they would combine in the correct way. However, had we done this, we might not have discovered the rich hierarchy of secure channel properties presented in Chapter 3.

4.6.2 Empirical channels

Creese et al. have developed the notion of *empirical channels*, and adapted the traditional attacker model for analysing protocols in order to study security protocols for pervasive computing [CGH+05]. They have a network

model comprising traditional, high-bandwidth digital communications channels, and empirical, low-bandwidth and human-oriented, channels. The empirical channels are used for non-traditional forms of communication, which often seem necessary for applications in pervasive computing, such as two humans comparing a code printed on each of their laptop screens, or a human entering a code on a printer's keypad.

Over such channels, they specify any combination of the following restrictions on the intruder:

NS No spoofing: the attacker cannot spoof messages on this channel;

NOH No over-hearing: the attacker cannot overhear messages sent on this channel;

NB No blocking: the attacker cannot block messages on this channel.

No spoofing corresponds to our definition of no-faking, although Creese et al. allow two different models of this channel: one that allows redirecting, and one that does not. No over-hearing corresponds to our definition of confidential channels. We do not have an equivalent to the no blocking channel, because on a traditional network, where the intruder is assumed to be in control of all message flows, we do not see how this anti-denial-of-service property could be realised; on the empirical channels suggested in [CGH+05] (such as a human entering a number on a keypad), it is easier to see how this would be possible.

Recently, Roscoe and Nguyen proposed several authentication protocols for use in environments where empirical channels are available alongside a traditional Dolev-Yao communication medium [RN06, RN08]. These protocols are designed for cases where there is no public-key infrastructure, and no secret information to bootstrap security from, so the human-mediated empirical channels are used to secure the communications over the Dolev-Yao network.

4.6.3 Security architectures using formal methods

Boyd [Boy93] defines two different types of channel in a security architecture consisting of users and information about who trusts whom. In Boyd's model a channel is a relationship between two users; a channel between two users is established when they share knowledge of a public key or a shared secret (e.g. a symmetric key). Thus channels are established by utilising existing keys, or propagating new keys between the two users wishing to communicate; often the propagation is over existing channels between trusted users. Boyd considers two types of cryptographic keys:

Confidentiality Only the intended user (or set of users) in possession of the secret key can read the message;

Authentication Only that user (or set of users) in possession of the secret key can write the message.

Boyd's channels can either be symmetric (in which case both users are sure of the other's identity) or not (in which case one user may be unsure of the other's identity). On a non-symmetric confidentiality channel, the message receiver may be unaware of the sender's identity: this is equivalent to a confidential channel in our model (the justification given for this is similar to ours: if A has a public key $PK(A)$ known to all the users in the system, then when she receives a message encrypted with $PK(A)$ she does not know who it is from, but she does know it was intended for her). On a symmetric confidentiality channel the message sender is authenticated to the receiver: this corresponds to an authenticated confidential channel. A non-symmetric authentication channel is redirectable, while a symmetric authentication channel is not. Boyd's authentication is equivalent either to a weakly authenticated channel or a strongly authenticated channel.

New channels can be established by transferring keys between users. For example if there is an authentication channel from A to B (written $A \xrightarrow{a} B$) and A sends a new public key (for which she knows the secret key) on this channel, this will establish a confidentiality channel $B \xrightarrow{c} A$. Boyd proves that this transfer correctly establishes a confidentiality channel, and also proves that some other simple transfers establish new channels correctly.

Boyd's channels can be directly compared with some of our channels, but his reasons for specifying the channels are different to ours. Boyd specifies his channels to describe security architectures in terms of the secure channels available; the model describes when new channels may be established, and formalises some intuitively obvious results (for example that no secure channels can be established between users who possess no secrets); we specify channels in order to enrich our abstract layered model for protocol analysis.

4.6.4 A calculus for secure channel establishment

In [MS94] Maurer and Schmid present a calculus of secure channel properties that allow them to classify and compare security protocols for establishing secure channels. Their channels are characterised by their direction, the time of availability, and, most relevantly to us, their security properties. The authors argue that cryptographic primitives (such as symmetric key distribution and encryption) can be interpreted as transformations for channel security properties, and so cryptographic protocols are just combinations of such transformations.

The main result of [MS94] is to determine necessary and sufficient requirements for establishing a secure channel between every pair of users in a network in terms of the channels available in the initial setup phase, and

the trust relations between users and trusted authorities. However, it is interesting to compare the secure channel properties defined by Maurer and Schmid to our secure channel properties.

A communication channel is defined as a means for transporting messages from a source to a destination; the channel from agent A to agent B is represented by the symbol $A \rightarrow B$. Maurer and Schmid define two basic security properties for a channel:

- A channel is confidential if its output is exclusively available to a specified receiver; such a channel is denoted $A \rightarrow\!\bullet B$.
- A channel is authenticated if its input is exclusively available to a specified sender; such a channel is denoted $A \bullet\!\rightarrow B$.

A channel that provides both confidentiality and authenticity is denoted $A \bullet\!\rightarrow\!\bullet B$.

The attacker in [MS94] cannot hijack messages in transit. A message sent on an authenticated channel may be redirected because the output is not restricted to a single specified recipient, but a confidential channel cannot be redirected: the attacker must learn the message, and then send it on a channel to the new recipient. The confidential channel is therefore simulated by $C \wedge NR^{-}$; the simulation also holds in the other direction because although the intruder cannot redirect messages that were sent to him, he can learn them and fake them.

Similarly, a confidential channel can be re-ascribed because the input is not restricted to a single specified sender, but an authenticated channel cannot be re-ascribed: the intruder must learn the message, and send it on a different channel (either an unauthenticated channel, or an authenticated channel that is restricted to a dishonest agent). The attacker cannot fake on an authenticated channel, so an authenticated channel is therefore simulated by $NF \wedge NRA^{-}$. Again, the simulation holds the other way because although the intruder cannot re-ascribe messages to himself, he can learn them and send them himself.

A channel that provides confidentiality *and* authenticity is equivalent to our top channel: $C \wedge NF \wedge NRA \wedge NR$.

Maurer and Schmid point out that symmetric (shared) key encryption naturally provides confidentiality, and can be adapted to provide authentication; if the plaintext message includes sufficient redundancy (such as a cryptographic hash of the message) to distinguish it from a random message then the fact that a message is encrypted under a certain key proves that the sender knows the key, and so authenticates the message sender. If the redundancy is removed from the plaintext message then symmetric key encryption can provide confidentiality without authentication. On the other hand, public-key encryption, and public-key distribution systems can

be used to establish channels that provide authentication and confidentiality independently, or together.

4.6.5 Language based secure communication

In [BF08] Bugliesi and Focardi develop a calculus of secure communication based on high-level security abstractions. Their calculus, and the abstract security properties are designed to ease the development and analysis of security-sensitive applications, so in many ways, their work shares the same goals as this thesis. The formalism of [BF08] is defined in the pi calculus [Mil99], where security is based on the notion of communication over private channels.

In the variant of the pi calculus described in [BF08], an output of message m originating from a and intended for b is denoted by $\bar{b}\langle a : m\rangle$; the input of that communication by b is denoted by $b(a : m)$.[5] These channel communications correspond to our send and receive events (though these communications do not include connection identifiers). The high-level security properties defined by Bugliesi and Focardi are as follows:

- $\bar{b}\langle - : m\rangle$ denotes a plain output (intended for b): this is a communication that has no security guarantees. An output of this form is equivalent to a send on the bottom channel ($\perp$).
- $\bar{b}\langle a : m\rangle$ denotes an authenticated output which provides the message recipient with a guarantee of the origin of the message. As in [MS94], the adversary cannot hijack messages in transit, and so an output of this form is simulated by a send event on a channel that satisfies $NF \wedge NRA^{-}$.
- $\bar{b}\langle - : m\rangle^{\bullet}$ denotes a secret output: only the intended recipient can learn the high-level message m. Again, the adversary cannot hijack messages in transit, so an output of this form is simulated by a send event on a channel that satisfies $C \wedge NR^{-}$.
- $\bar{b}\langle a : m\rangle^{\bullet}$ denotes a secret and authenticated output: this combines the guarantees of the authentic and secret modes, and is equivalent to a send event on a channel that satisfies $C \wedge NF \wedge NRA \wedge NR$.

The most notable difference between our channel properties and the security properties defined in [BF08] is that their authenticated channel prevents messages from being replayed, whereas none of our channels prevent message replays. The authentication property from [BF08] is designed to capture the notion that every time you receive a message from a then a sent that message to you; there is an injective correspondence between the

[5] We have altered the syntax from [BF08] slightly in order to ease the comparison with our channel properties.

message outputs and message inputs. The authentication property defined in this thesis is designed to capture the notion that if, at any time, you receive a message from a then a previously sent that message.

The formalism in [BF08] is split into two layers; the high-level communications (described above) are run on top of a low-level network, over which the adversary is given full control. At the network level the adversary can intercept communications, and forward or replay messages that he has intercepted. The adversary can forward all messages, but he can only replay unauthenticated communications. The network-level primitives include the network-level view of the message payload for every communication. On confidential channels the intruder can only learn the network-level view of the message, while on non-confidential channels the intruder learns the network-level view of the message as well as the high-level message payload itself.

Bugliesi and Focardi give a compositional formulation of the semantics of their networks which is based on labelled transitions. They then define several bisimilarity and observational equivalence relations; these are used to prove secrecy and authentication properties (e.g. secret outputs guarantee the privacy of the payload). Finally, the authors claim that their high-level security primitives can be implemented by time-variant signatures (for authentication) and randomised public-key encryption (for confidentiality).

4.6.6 LTL model checking for security protocols

In [ACC07] Armando et al. propose a general model for security protocols that uses LTL to specify complex security properties of a more secure system than the standard Dolev-Yao model. Their model is based on the formalism of [CDL+99], and uses set-rewriting to specify the transition system associated with the execution on several concurrent runs of the protocol.

The honest agents and the intruder communicate over channels; the channels are usually named according to the identities of the agents at either end, but in the standard model any agent can send or receive a message on any channel. The fact $\texttt{sent}(rs, a, b, m, c)$ indicates that agent rs, pretending to be agent a, has sent message m on channel c to agent b. The fact $\texttt{rcvd}(b, a, m, c)$ indicates that message m (supposedly sent by agent a) has been received on the channel c by agent b. The set-rewriting rules for the honest agents include rules such as the following one (which models the reception of a message by an honest agent):

$$\texttt{sent}(rs, a, b, m, c) \xrightarrow{\texttt{receive}(b,a,rs,m,c)} \texttt{rcvd}(b, a, m, c).\texttt{ak}(b, m)\,.$$

After receiving message m, agent b knows it ($\texttt{ak}(b, m)$).

The intruder is given three basic capabilities (as well as the capability to send and receive messages): he can fake messages that he knows; intercept (block) messages sent by honest agents on any channel and learn the

messages; and overhear messages sent by honest agents on any channel and learn the messages.

As in many other models, on the standard network the intruder cannot hijack messages in transit, he must overhear (or intercept) them, and then fake a new message. Armando et al. specify three channel properties.

A channel c is confidential to a set of agents AS if its output is exclusive to agents in that set: $\mathbf{G}\forall(\texttt{rcvd}(b, a, m, c) \Rightarrow b \in AS)$. On the confidential channel (c, AS), only agents in the set AS can learn messages sent on the channel, so a confidential channel prevents the intruder from intercepting and overhearing messages, though he can still send (or fake) messages on such a channel. A confidential channel is certainly simulated by $C \wedge NR^{-}$, but it actually provides something much stronger: the intruder can only replay messages on a channel if he can overhear or intercept them, so this channel prevents replaying.

Because the channel is not directional, this property also provides a symmetric session property if the set AS is a pair of agents. For example, the confidential channel $(c, \{a, b\})$ is a symmetric session channel between a and b, and satisfies $C \wedge NR^{-}$ in both directions. Armando et al. also specify a weakly confidential channel: this is the same as the confidential channel property except that the intruder can store previously sent messages and replay them, even though he cannot learn them.

The third channel specified in [ACC07] is a resilient channel; a channel is resilient if it is normally operational but an attacker can delay messages by an arbitrary, but finite, amount of time. We do not have a property like this for the same reason that we do not have a no-blocking property: on the sorts of networks that we are interested in, we do not see how the property could be implemented.

Although they only specify three properties in [ACC07], the formalism is suitably flexible to allow many more properties to be specified. In [ACC^{+}08] Armando et al. specify another confidentiality property, and two authentication properties designed specifically for capturing the services provided by SSL/TLS.

A channel provides weak confidentiality if its output is exclusively available to a single, unknown receiver; if a state contains a $\texttt{rcvd}(b, a, m, c)$ fact then in all successor states the `rcvd` facts on channel c must have a as the recipient. This captures a weaker notion of confidentiality than $C \wedge NR^{-}$: the intruder can learn the messages on the channel, but only if he receives all messages on the channel. He cannot overhear messages that are being received by an honest agent.

A channel provides authentication if its input is exclusively available to a single agent who is sending messages using his own identity, and a channel provides weak authentication if its input is exclusively available to a single agent who many not be sending messages using his own identity. These three channels all provide the session property, but they are not symmetric

(as the channels in [ACC07] are). A unilateral TLS connection is modelled by a pair of channels, one of which is weakly confidential, and the other is weakly authentic, and the agent sending messages in one channel is the agent receiving messages in the other.

4.6.7 Verifying second-level security protocols

Bella et al. have extended the inductive approach to verifying security protocols [Pau98] to verify *second-level security protocols*: security protocols that rely on an underlying security protocol to achieve their goals [BLP03]. Bella et al. extend the inductive approach by modelling the properties of the underlying security protocol abstractly, and then performing a standard inductive analysis on the second-level security protocol. In this respect their approach is similar to ours; however, we specify a great many more possible properties for the underlying secure transport protocol.

The inductive approach models security protocols as a set of rules by which honest agents may extend the trace of events. These rules involve events such as $\mathsf{Says}\ A\ B\ X$ (agent A sends the message X to agent B); $\mathsf{Gets}\ B\ X$ (agent B receives the message X from the network); and $\mathsf{Notes}\ A\ X$ (agent A stores the message X). Under the standard model the Dolev-Yao intruder overhears all Says events, and can cause agents to perform Gets events by faking Says events.

Bella et al. specify three properties that the underlying security protocol can provide to the second-level security protocol:

Authentication In the original approach message reception was modelled with a $\mathsf{Says}\ A'\ B\ X$ event, but the message recipient could not use the value of the sender's identity A' because it could not verify it. In their new model of authenticated channels the sender's identity A of a $\mathsf{Says}\ A\ B\ X$ event cannot be altered once the event appears in a trace, and so the message recipient can use, and rely on, the sender's identity. The intruder can still send messages with his own identity, but he cannot fake messages, nor can he re-ascribe them.

Confidentiality The intruder cannot overhear messages sent on confidential channels. Bella et al. use the $\mathsf{Notes}\ B\ \{|A, B, X|\}$ event to model the confidential transmission (from A to B) of the message X. We note that because the intruder cannot generate Notes events, the confidential channel is also authenticated. Because the honest agents can respond to prior events in the trace as many times as they wish, the model of this channel allows messages to be replayed.

Guaranteed delivery A channel that guarantees message delivery is modelled by the message sender causing the intended recipient to the receive the message. We do not specify a channel property like this, be-

cause we do not think that it is reasonable to expect a secure transport-layer protocol running on a traditional network to achieve it. Bella et al. suggest that this property can be achieved by the message sender repeatedly sending the message until he receives an authenticated acknowledgement; we claim that this property is provided by a symmetric stream channel when the message sender receives a response from the message recipient, and on the synchronised stream channels when the message sender can send his second message.

4.6.8 Key-exchange protocols and secure channels

Canetti and Krawczyk analyse the use of key-exchange protocols as mechanisms for establishing secret keys in order to build secure channels [CK01]. Their work is set in the formalism of the unauthenticated-links model (UM) and the authenticated-links model (AM) of [BCK98].

In the UM, the adversary is assumed to be in full control of the network: he can generate, inject and modify messages, and he can decide whether or not to deliver messages. In the AM the adversary can only deliver messages faithfully (i.e. he cannot fake, re-ascribe or redirect messages); he can, however, re-order messages, or fail to deliver them. The AM corresponds to a network in which all channels satisfy $NF \wedge NRA \wedge NR$.

The authors formalise the notion of session-key security (SK-security); a key-exchange protocol is said to be SK-secure if:

1. Whenever two parties complete matching sessions of the protocol they output the same key;

2. The probability that the adversary can correctly distinguish a random key from the session-key established in any test session is not significantly greater than a half.

The authors prove that an SK-secure protocol in the AM can be transformed into an SK-secure protocol in the UM by applying a suitable perfectly authenticated message-transmission protocol to every message flow in the original protocol.

Canetti and Krawczyk prove that an SK-secure key-exchange protocol can be combined with a MAC function that is secure against chosen-message attacks and a symmetric encryption function that is secure against chosen-plaintext attacks to produce a secure channel protocol that is both secret and authenticated.

They define secrecy as an indistinguishability property: in any session the adversary can choose two messages, one of which an uncompromised party will send over the secure channel; the adversary must not have odds significantly greater than fifty-fifty of guessing which message was sent.

The authentication property is that of the AM: messages must be delivered faithfully. Further, messages cannot be replayed (though they can be

re-ordered), and sessions are established symmetrically. Such a secure channel fits in between a symmetric session and symmetric stream channel that satisfies $C \wedge NF \wedge NRA \wedge NR$ (though of course the notion of confidentiality in [CK01] is different to our (formal) notion).

In [CK02] the authors re-cast their definitions in the Universally Composable framework to show that they can be composed with an arbitrary application protocol, and run concurrently with other protocols.

4.7 Conclusions

In this chapter we formalised our notion of simulation of channel specifications in terms of the honest agents' views of the valid system traces. By hiding the intruder's events, and only examining the events that he can cause the honest agents to perform, we make the intruder's activity abstract; this allows us to compare channel specifications, even when the model of the intruder is different. Our simulation relation can be expressed in terms of CSP process simulation after hiding the events performed by the intruder. We used our simulation relation to define an equivalence relation (mutual simulation).

We used this equivalence relation to prove the equivalence of alternative forms of our channel specifications. Rather than blocking the intruder's events, these alternative forms state the possible events that could precede a receive event, and require that one of them does. The alternative specifications are more conducive to proving properties about the secure channel properties. We illustrated the proof technique for showing the equivalence of a channel property expressed in terms of blocking the intruder's events and its alternative form.

We showed that every possible combination of the channel primitives (the specifications that block the individual intruder events) is either a point in the channel hierarchy, or collapses to a unique point in the hierarchy. We also showed that we could safely block some combinations of the intruder's events (e.g. hijacking messages sent by himself) because they simulate other events; by blocking these combinations of events we reduce the set of valid system traces and so we can simplify proofs about channel specifications.

We specified a sufficient condition for safely substituting session channels for stream channels in an implementation of an application-layer protocol. For any application-layer protocol that satisfies no speaking out of turn and disjoint messages, session channels can be used instead of stream channels (that satisfy the same authentication and confidentiality properties) without introducing attacks to that protocol.

Finally, we compared our channel properties and our framework for specifying channel properties to the approach taken by other researchers, and to the channel properties that they have devised.

Chapter 5

Chaining secure channels

In Chapters 3 and 4 we exclusively described channels that secure point-to-point connections; in this chapter we examine the possibilities for chaining secure channels.

We consider chaining channels in two different ways: first, in Section 5.1, through a set of dedicated intermediaries (simple proxies), and then, in Section 5.2, through a (much smaller) set of trustworthy (multiplexing) proxies. We present a surprising theorem that shows how, under some circumstances, two channels can be chained to produce a stronger channel. We also show that the channel established through a proxy is always simulated by (i.e. is at least as strong as) the greatest lower bound of the channels established to and from the proxy. In Section 5.3 we discuss some similar results about chaining secure channels discovered by other researchers. Finally, in Section 5.4 we conclude and summarise our findings.

The results presented in this section are particularly relevant to real-world use of secure channels. Many organisations arrange their computers in a trusted intranet, and only allow external access through a proxy. For example:

- Many grid architectures (such as Globus [FK97]) only allow communication to the servers that the grid is comprised of through a *gatekeeper*. The role of the gatekeeper is to scan the incoming (and outgoing) network traffic, and to block messages that do not match the set of rules that it holds;

- Firewalls may scan all network traffic that passes through them, and automatically block messages that contain viruses. If computers on the internal network can communicate securely with computers on the external network, the firewall can no longer perform its task. The firewall must establish secure connections with external agents on behalf of the internal agents if it is still to scan messages for viruses.

In these scenarios the gatekeeper and firewall are acting as proxies for the

trusted servers.

It is not obvious that a secure connection is established between two agents by establishing two secure connections through a trusted proxy. It is also not obvious which security properties, if any, the resultant channel provides.

Because we require proxies to establish application-layer connections between agents, we consider proxies to be application-layer agents. While it may be the case that many proxies run on dedicated hardware, and exchange data at lower levels of the network-stack than the application layer, our discussion of proxies, and of channel chaining, is based on application-layer proxies.

5.1 Simple proxies

In this section we consider the case when the proxies are *simple*.

Definition 5.1.1. A *simple proxy* is an agent who is dedicated to forwarding messages from one agent to another. For every pair of roles (R_i, R_j) there is a simple proxy role $Proxy_{(R_i,R_j)}$. For every pair of agents $(A : \hat{R}_i, B : \hat{R}_j)$ such that $A \neq B$ there is a unique simple proxy $P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}$ who forwards messages from A to B. When two roles communicate through a simple proxy, the following trace specification is satisfied:

$$
\begin{aligned}
&SimpleProxies(R_i \to R_j)(tr) \mathrel{\widehat{=}} \\
&\quad tr \downarrow \{\!|\, send.\hat{R}_i.Connection.\hat{R}_j \,|\!\} = \langle\rangle \wedge \\
&\quad tr \downarrow \{\!|\, receive.\hat{R}_j.Connection.\hat{R}_i \,|\!\} = \langle\rangle \wedge \\
&\quad \forall A : \hat{R}_i; B : \hat{R}_j \cdot \exists P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)} \cdot SimpleProxy(P_{(A,B)})(tr) \wedge \\
&\quad \forall A, A' : \hat{R}_i; B : \hat{R}_j; P_{(A',B)} : \widehat{Proxy}_{(R_i,R_j)}; c_A : Connection; m : Message_{App} \cdot \\
&\qquad send.A.c_A.P_{(A',B)}.m \textbf{ in } tr \wedge Honest(A) \Rightarrow A' = A \wedge \\
&\quad \forall A : \hat{R}_i; B, B' : \hat{R}_j; P_{(A,B')} : \widehat{Proxy}_{(R_i,R_j)}; c_B : Connection; m : Message_{App} \cdot \\
&\qquad receive.B.c_B.P_{(A,B')}.m \textbf{ in } tr \wedge Honest(B) \Rightarrow B' = B\,,
\end{aligned}
$$

where each simple proxy satisfies the following specification:

$$
\begin{aligned}
&SimpleProxy(P_{(A,B)})(tr) \mathrel{\widehat{=}} \\
&\quad \forall c_P : Connection; m : Message \cdot \\
&\qquad \forall A' : Agent \cdot receive.P_{(A,B)}.c_P.A'.m \textbf{ in } tr \Rightarrow A' = A \wedge \\
&\qquad \forall B' : Agent \cdot send.P_{(A,B)}.c_P.B'.m \textbf{ in } tr \Rightarrow B' = B \wedge \\
&\qquad \exists c'_P : Connection \cdot send.P_{(A,B)}.c_P.B \leqslant receive.P_{(A,B)}.c'_P.A\,.
\end{aligned}
$$

The simple proxy $P_{(A,B)}$ only establishes connections with A and B, and each connection it establishes is either dedicated to sending messages to B or to receiving messages from A. Further, the simple proxy forwards every message that it receives from A to B.

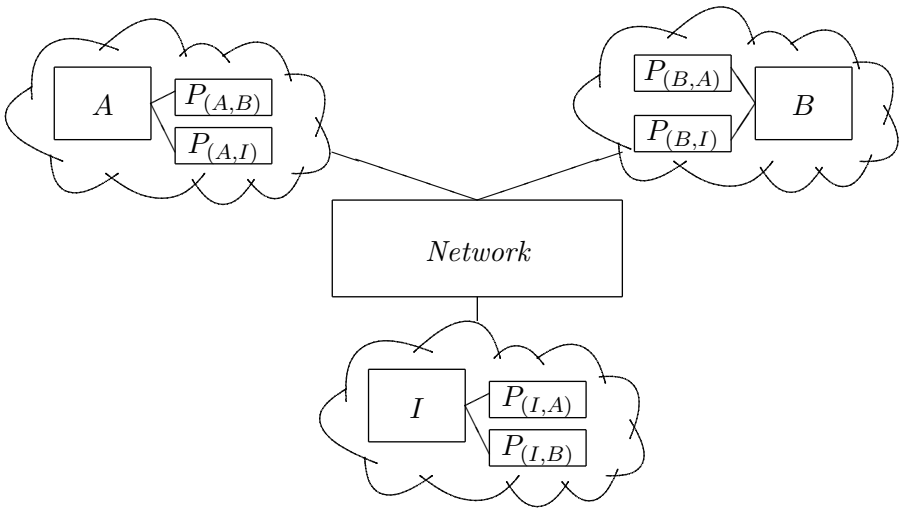


Figure 5.1: Simple proxies.

Because the proxy $P_{(A,B)}$ acts on A's behalf, the proxy is honest if and only if A is honest. We think of the family of proxies $\{P_{(A,B)} \mid B : Agent\}$ as A's proxies (because they all send on her behalf); see Figure 5.1.

Each simple proxy has a particular job: if $P_{(A,B)}$ receives a message that appears to be from A, he forwards it to B; $P_{(A,B)}$ does not receive messages that appear to have been sent by any other agent. $P_{(A,B)}$ only ever sends messages to B, and never to any other agent. We assume that every agent knows all of its proxies, and also knows which proxies send messages to it,[1] and so if an honest agent who is not B is sent a message from $P_{(A,B)}$ he discards it. We assume that honest agents never attempt to send a message to any simple proxies other than their own.

5.1.1 Secure channels through simple proxies

In order to discover which security properties the channel through a simple proxy satisfies we consider each of the components of the hierarchy individually. In the discussion below we refer to the channel to the proxy as $(R_i \rightarrow Proxy_{(R_i,R_j)})$, and the channel from the proxy as $(Proxy_{(R_i,R_j)} \rightarrow R_j)$; we refer to the overall channel through the proxy as $Proxy\overrightarrow{_{(R_i,R_j)}}$.

Confidentiality It is clear that if either of the channels to or from a simple proxy is not confidential, the channel through the proxy is not confidential; i.e. the channel through the proxy is confidential only if the channels to and from the proxy are both confidential.

[1] This could be implemented in the same way as, or even integrated with, a Public Key Infrastructure.

No faking It is also clear that if either of the channels to or from a simple proxy is fakeable, then the channel through the proxy is fakeable. In order to fake a message from A to B, the intruder must either fake sending the message to A's proxy $P_{(A,B)}$, or fake sending the message from A's proxy to B.

No re-ascribing The intruder can either re-ascribe a message on the channel to the proxy or on the channel from the proxy:

1. In order to re-ascribe a message on the channel to the proxy it is not enough for the intruder just to change the sender's identity:

$$tr \mathrel{\widehat{=}} \langle send.A.c_A.P_{(A,B)}.m, hijack.A \rightarrow A'.P_{(A,B)}.c_P.m \rangle \,.$$

   A's proxy will not accept a message that appears to have been sent by another agent (A'). In order to re-ascribe a message on the channel to the proxy, the intruder must also be able to redirect the message to the correct one of the new sender's proxies.

2. On the other hand, re-ascribing a message on the channel from the proxy is straightforward:[2]

$$\begin{aligned} tr \mathrel{\widehat{=}} \langle & send.A.c_A.P_{(A,B)}.m, receive.P_{(A,B)}.c_P.A.m, \\ & send.P_{(A,B)}.c'_P.B.m, hijack.P_{(A,B)} \rightarrow P_{(A',B)}.B.c_B.m, \\ & receive.B.c_B.P_{(A',B)}.m \rangle \,. \end{aligned}$$

   The intruder only needs to change the sender's identity to that of another agent's proxy.

No redirecting The intruder can either redirect a message on the channel to the proxy or on the channel from the proxy:

1. In order to redirect a message on the channel to the proxy the intruder simply redirects the message to a different proxy:

$$\begin{aligned} tr \mathrel{\widehat{=}} \langle & send.A.c_A.P_{(A,B)}.m, hijack.A.P_{(A,B)} \rightarrow P_{(A,B')}.c_P.m, \\ & receive.P_{(A,B')}.c_P.A.m, send.P_{(A,B')}.c'_P.B'.m, \\ & receive.B'.c_{B'}.P_{(A,B')}.m \rangle \,. \end{aligned}$$

2. On the other hand, in order to redirect a message on the channel from the proxy the intruder cannot just change the recipient's identity:

$$\begin{aligned} tr \mathrel{\widehat{=}} \langle & send.A.c_A.P_{(A,B)}.m, receive.P_{(A,B)}.c_P.A.m \\ & send.P_{(A,B)}.c'_P.B.m, hijack.P_{(A,B)}.B \rightarrow B'.c_{B'}.m \rangle \,. \end{aligned}$$

[2]Note that the proxies $P_{(A,B)}$ and $P_{(A',B)}$ are different agents.

B' will not accept a message from the proxy $P_{(A,B)}$ because B' knows which proxies send messages to him; in order to redirect a message on this channel the intruder must also be able to re-ascribe the message to one of the proxies that B' accepts messages from.

The *SimpleProxies* property on the roles R_i and R_j prevents agents playing role R_i from communicating directly with agents playing role R_j: it insists that they only communicate through proxies. This means that the standard definitions of our secure channels (which restrict the intruder's behaviour when hijacking or faking messages) are vacuously satisfied: there are no messages sent by agents playing role R_i to agents playing role R_j to hijack, and no agent playing role R_j will accept a message that appears to be from an agent playing role R_i.

We have seen that in order to fake a message, the intruder can fake it on the channel to the proxy, or on the channel from the proxy. We have also seen that the intruder can hijack messages on either the channel to the proxy, or on the channel from the proxy. In order to block these activities, we must do so on both channels; we state the definitions of our building blocks on the channel through a simple proxy below.

Definition 5.1.2 (No faking)**.**

$$\begin{aligned}&NF(Proxy^{\overrightarrow{\ }}_{(R_i,R_j)})(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{|\, fake.\hat{R}_i.P\widehat{roxy}_{(R_i,R_j)},\ fake.P\widehat{roxy}_{(R_i,R_j)}.\hat{R}_j \,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.1.3 (No-re-ascribing)**.**

$$\begin{aligned}&NRA(Proxy^{\overrightarrow{\ }}_{(R_i,R_j)})(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{|\, hijack.A \rightarrow A'.P_{(A,B)} \rightarrow P_{(A',B')},\ hijack.P_{(A,B)} \rightarrow P_{(A',B')}.B \rightarrow B' \mid \\ &\qquad A, A' \in \hat{R}_i \wedge B, B' \in \hat{R}_j \wedge P_{(A,B)}, P_{(A',B')} \in P\widehat{roxy}_{(R_i,R_j)} \wedge \\ &\qquad A \neq A' \,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.1.4 (No-honest-re-ascribing)**.**

$$\begin{aligned}&NRA^{-}(Proxy^{\overrightarrow{\ }}_{(R_i,R_j)})(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{|\, hijack.A \rightarrow A'.P_{(A,B)} \rightarrow P_{(A',B')},\ hijack.P_{(A,B)} \rightarrow P_{(A',B')}.B \rightarrow B' \mid \\ &\qquad A, A' \in \hat{R}_i \wedge B, B' \in \hat{R}_j \wedge P_{(A,B)}, P_{(A',B')} \in P\widehat{roxy}_{(R_i,R_j)} \wedge \\ &\qquad A \neq A' \wedge Honest(A') \,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.1.5 (No-redirecting)**.**

$$\begin{aligned}&NR(Proxy^{\overrightarrow{\ }}_{(R_i,R_j)})(tr) \mathrel{\widehat{=}} \\ &\quad tr \downarrow \{|\, hijack.A \rightarrow A'.P_{(A,B)} \rightarrow P_{(A',B')},\ hijack.P_{(A,B)} \rightarrow P_{(A',B')}.B \rightarrow B' \mid \\ &\qquad A, A' \in \hat{R}_i \wedge B, B' \in \hat{R}_j \wedge P_{(A,B)}, P_{(A',B')} \in P\widehat{roxy}_{(R_i,R_j)} \wedge \\ &\qquad B \neq B' \,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.1.6 (No-honest-redirecting)**.**

$$
\begin{array}{l}
NR^{-}(\overrightarrow{Proxy_{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad tr \downarrow \{\!| \, hijack.A \rightarrow A'.P_{(A,B)} \rightarrow P_{(A',B')},\ hijack.P_{(A,B)} \rightarrow P_{(A',B')}.B \rightarrow B' \mid \\
\qquad\qquad A, A' \in \hat{R}_i \wedge B, B' \in \hat{R}_j \wedge P_{(A,B)}, P_{(A',B')} \in \widehat{Proxy}_{(R_i,R_j)} \wedge \\
\qquad\qquad B \neq B' \wedge Honest(B) \mid \!\!\} = \langle\rangle \, .
\end{array}
$$

In the proof of the simple chaining theorem, below, we show that the alternative specifications for each of the channels in the hierarchy through a simple proxy are satisfied. These forms of the alternative specifications take account of the fact that messages can be faked or hijacked on the channel to the proxy or on the channel from the proxy; the specifications are shown in Appendix A.2. We present one example below; the alternative form of the channel specification $(C \wedge NRA^{-} \wedge NR^{-})(\overrightarrow{Proxy_{(R_i,R_j)}})$ is:

$$
\begin{array}{l}
Alt(C \wedge NRA^{-} \wedge NR^{-})(\overrightarrow{Proxy_{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\quad fake.P_{(A,B)}.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P_{(A',B')} \rightarrow P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m \textbf{ in } tr \, .
\end{array}
$$

5.1.2 Simple chaining theorem

We make the following observations of the overall channel through a simple proxy.

Observation 5.1.7. If the intruder cannot redirect messages that were sent to honest agents on the channel to the proxy, then he cannot re-ascribe messages on the channel to the proxy. In order to re-ascribe a message the intruder must be able to redirect the message to one of the new sender's proxies. Further, since all honest agents' proxies are honest, no honest agent ever sends a message to a dishonest agent on the channel to the proxy. Subject to the collapsing cases described earlier, if the channel to the proxy satisfies NR^{-} it also satisfies $NRA \wedge NR$.

Observation 5.1.8. If the intruder cannot re-ascribe messages to honest agents on the channel from the proxy, then he cannot redirect messages on the channel from the proxy. In order to redirect a message the intruder must be able to re-ascribe it to one of the proxies who sends messages to the new

recipient. Subject to the collapsing cases described earlier, if the channel from the proxy satisfies $NRA^- $ it also satisfies $NRA^- \wedge NR$.

Theorem 5.1.9 (Simple chaining theorem)**.** *If roles R_i and R_j communicate through simple proxies (i.e. SimpleProxies($R_i \to R_j$)) on secure channels such that:*

$$\begin{aligned} &\mathit{ChannelSpec}_1(R_i \to \mathit{Proxy}_{(R_i,R_j)})\,, \\ &\mathit{ChannelSpec}_2(\mathit{Proxy}_{(R_i,R_j)} \to R_j)\,, \end{aligned}$$

where $\mathit{ChannelSpec}_1$ and $\mathit{ChannelSpec}_2$ are channels in the hierarchy, then the overall channel (through the proxy) satisfies the channel specification:

$$\mathit{ChannelSpec} =\downarrow (\nwarrow_s \mathit{ChannelSpec}_1 \sqcap \nearrow_s \mathit{ChannelSpec}_2)\,;$$

where:

$$\begin{aligned} \nwarrow_s (c, nf, nra, nr) = \\ \quad (c, nf, NRA, NR) \ \textit{if}\ nr \in \{NR^-, NR\}\,, \\ \quad (c, nf, nra, nr) \ \textit{otherwise}\,; \end{aligned}$$

$$\begin{aligned} \nearrow_s (c, nf, nra, nr) = \\ \quad (c, nf, nra, NR) \ \textit{if}\ nra \in \{NRA^-, NRA\}\,, \\ \quad (c, nf, nra, nr) \ \textit{otherwise}\,; \end{aligned}$$

and $\sqcap$ is the greatest lower bound operator in the full lattice.

The proof of the simple chaining theorem is in Section 5.1.3.

Corollary 5.1.10. *If roles R_i and R_j communicate through simple proxies (i.e. SimpleProxies($R_i \to R_j$)) on secure channels such that:*

$$\begin{aligned} &\mathit{ChannelSpec}(R_i \to \mathit{Proxy}_{(R_i,R_j)})\,, \\ &\mathit{ChannelSpec}(\mathit{Proxy}_{(R_i,R_j)} \to R_j)\,, \end{aligned}$$

where ChannelSpec is a channel in the hierarchy, then the overall channel (through the proxy) satisfies a channel specification ChannelSpec$'$ such that:

$$\mathit{ChannelSpec} \preccurlyeq \mathit{ChannelSpec}'\,.$$

In particular, $\mathit{ChannelSpec}(\overrightarrow{\mathit{Proxy}_{(R_i,R_j)}})$ holds.

Example 5.1.11. The channel to the proxy satisfies $C \wedge NRA \wedge NR^-$, and the channel from the proxy satisfies $NF \wedge NRA^- \wedge NR$.

$$\begin{aligned} &\nwarrow_s (C \wedge NRA \wedge NR^-) = C \wedge NRA \wedge NR\,, \\ &\nearrow_s (NF \wedge NRA^- \wedge NR) = NF \wedge NRA^- \wedge NR\,. \end{aligned}$$

The greatest lower bound of these two points is $NRA^- \wedge NR$, which collapses to $\bot$.

The channel to the proxy is fakeable and the channel from the proxy is non-confidential; because of collapsing case $\mathit{Collapse}_1$, the overall channel simulates the bottom channel.

Example 5.1.12. The channel to the proxy satisfies $NF \wedge NRA^- \wedge NR^-$, and the channel from the proxy satisfies $NF \wedge NRA^-$.

$$\begin{aligned} \nwarrow_s (NF \wedge NRA^- \wedge NR^-) &= NF \wedge NRA \wedge NR\,, \\ \nearrow_s (NF \wedge NRA^-) &= NF \wedge NRA^- \wedge NR\,. \end{aligned}$$

The greatest lower bound of these two points is $NF \wedge NRA^- \wedge NR$, which does not collapse. This channel is stronger than both of the individual channels.

The intruder cannot fake messages on this channel, nor can he redirect messages (because he cannot redirect messages using the channel to the proxy, and he cannot re-ascribe messages using the channel from the proxy). The intruder can only re-ascribe messages with his own identity because this is the greatest capability he has on each channel individually.

Example 5.1.13. The channel to the proxy satisfies $C \wedge NR^-$, and the channel from the proxy satisfies $C \wedge NRA^- \wedge NR^-$.

$$\begin{aligned} \nwarrow_s (C \wedge NR^-) &= C \wedge NRA \wedge NR\,, \\ \nearrow_s (C \wedge NRA^- \wedge NR^-) &= C \wedge NRA^- \wedge NR\,. \end{aligned}$$

The greatest lower bound of these two points is $C \wedge NRA^- \wedge NR$, which collapses to $C \wedge NRA^- \wedge NR^-$ by $Collapse_5$. This channel is stronger than the greatest lower bound of the two individual channels.

Although the channel to the proxy is re-ascribable, it only allows the intruder to redirect messages that were sent to him, so the overall channel only allows the intruder to re-ascribe messages to his own identity (on the channel from the proxy).

The resultant channels for all instances of the chaining theorem are shown in Figure B.1 (in Appendix B.1).

5.1.3 An automated proof of the simple chaining theorem

Each instance of Theorem 5.1.9 is relatively simple to prove. One simply starts with a receive event, and calculates which events are allowed (by the channel specifications to and from the proxy, and by the network rules) to precede it in a valid system trace. Each receive event can be traced back to a set of send, fake, or hijack events; it is then straightforward to determine the strongest channel whose alternative specification is satisfied.

Before we describe the automated proof of the theorem, we show an example proof of one instance.

Lemma 5.1.14. *If roles R_i and R_j communicate through simple proxies (i.e. SimpleProxies($R_i \rightarrow R_j$)) on secure channels such that:*

$$\begin{aligned} &(C \wedge NR^-)(R_i \rightarrow Proxy_{(R_i,R_j)})\,, \\ &(C \wedge NRA^- \wedge NR^-)(Proxy_{(R_i,R_j)} \rightarrow R_j)\,, \end{aligned}$$

then $(C \wedge NRA^- \wedge NR^-)(\mathit{Proxy}_{(R_i,R_j)}^{\longrightarrow})$.

Proof. We use the *SimpleProxies* property and the alternative specifications of the channels to and from the proxy to show that the alternative form of $(C \wedge NRA^- \wedge NR^-)(\mathit{Proxy}_{(R_i,R_j)}^{\longrightarrow})$ holds; i.e.

$$
\begin{array}{l}
C(R_i \rightarrow \mathit{Proxy}_{(R_i,R_j)}) \wedge C(\mathit{Proxy}_{(R_i,R_j)} \rightarrow R_j) \wedge \\
\forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; P_{(A,B)} : \widehat{\mathit{Proxy}}_{(R_i,R_j)}; m : \mathit{Message}_{\mathit{App}} \cdot \\
\quad \mathit{receive}.B.c_B.P_{(A,B)}.m \ \mathbf{in}\ tr \Rightarrow \\
\qquad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.P_{(A,B)}.m \ \mathbf{in}\ tr \vee \\
\qquad \exists c_P : \mathit{Connection} \cdot \mathit{fake}.A.P_{(A,B)}.c_P.m \ \mathbf{in}\ tr \vee \\
\qquad \mathit{fake}.P_{(A,B)}.B.c_B.m \ \mathbf{in}\ tr \vee \\
\qquad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{\mathit{Proxy}}_{(R_i,R_j)}; c_P : \mathit{Connection} \cdot \\
\qquad\quad ((A' = A) \vee \mathit{Dishonest}(A)) \wedge ((B' = B) \vee \mathit{Dishonest}(B')) \wedge \\
\qquad\quad \mathit{hijack}.A' \rightarrow A.P_{(A',B')} \rightarrow P_{(A,B)}.c_P.m \ \mathbf{in}\ tr \vee \\
\qquad\quad \mathit{hijack}.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m \ \mathbf{in}\ tr\,.
\end{array}
\tag{$\dagger$}
$$

holds for all valid system traces tr such that for all prefixes $tr' \leqslant tr$:

$$
\begin{array}{l}
\mathit{SimpleProxies}(R_i \rightarrow R_j)(tr')\,, \\
(C \wedge NR^-)(R_i \rightarrow \mathit{Proxy}_{(R_i,R_j)})(tr')\,, \\
(C \wedge NRA^- \wedge NR^-)(\mathit{Proxy}_{(R_i,R_j)} \rightarrow R_j)(tr')\,.
\end{array}
$$

It is clear that $C(R_i \rightarrow \mathit{Proxy}_{(R_i,R_j)})(tr)$ and $C(\mathit{Proxy}_{(R_i,R_j)} \rightarrow R_j)(tr)$ both hold, so we must show that the second half of ($\dagger$) holds.

Let A and B be agents playing roles R_i and R_j, $P_{(A,B)}$ be A's proxy to B, c_B a connection and m an application-layer message. Suppose that the event $\mathit{receive}.B.c_B.P_{(A,B)}.m$ occurs in tr; the network rule $\mathcal{N}_4$ implies the existence of one of several events earlier in the trace. The set of possible events is limited by $C \wedge NRA^- \wedge NR^-$, which holds on the channel from the proxy:

1. $\exists c_P : \mathit{Connection} \cdot \mathit{send}.P_{(A,B)}.c_P.B.m \ \mathbf{in}\ tr$;
2. $\mathit{fake}.P_{(A,B)}.B.c_B.m \ \mathbf{in}\ tr$;
3. $\exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{\mathit{Proxy}}_{(R_i,R_j)} \cdot$
   $\mathit{hijack}.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m \ \mathbf{in}\ tr \wedge$
   $((P_{(A',B')} = P_{(A,B)}) \vee \mathit{Dishonest}(P_{(A,B)})) \wedge$
   $((B' = B) \vee \mathit{Dishonest}(B'))$.

We consider each of these possibilities independently because each one leads to a different trace.

1. Suppose that $\mathit{send}.P_{(A,B)}.c_P.B.m$ precedes the receive event in the trace tr, for some connection identifier c_P. $\mathit{SimpleProxies}(R_i \rightarrow R_j)$

implies $\mathit{SimpleProxy}(P_{(A,B)})(tr)$, and so $\mathit{receive}.P_{(A,B)}.c'_P.A.m$ occurs earlier in the trace, for some connection c'_P.

We use network rule $\mathcal{N}_4$ again; this implies the existence of one of several events earlier in the trace, and, as before, these possibilities are limited by $C \wedge NR^-$, which holds on the channel to the proxy:

(a) $\exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.P_{(A,B)}.m$ **in** tr ;

(b) $\mathit{fake}.A.P_{(A,B)}.c'_P.m$ **in** tr ;

(c) $\exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)} \cdot$
$\mathit{hijack}.A' \rightarrow A.P_{(A',B')} \rightarrow P_{(A,B)}.c'_P.m$ **in** $tr \wedge$
$((P_{(A',B')} = P_{(A,B)}) \vee \mathit{Dishonest}(P_{(A',B')}))$.

The first two of these disjuncts match two of the disjuncts in (†), so we do not (and cannot) trace these back further.

In the third disjunct, if $A' \neq A$ then $P_{(A',B')} \neq P_{(A,B)}$, so $P_{(A',B')}$ is dishonest, and hence A' is dishonest. A must be honest (because the intruder does not re-ascribe his own messages to himself), so this trace simulates one in which the intruder fakes the message with A's identity.

If $B' \neq B$ then the same argument shows that A' is dishonest; again, if $A' \neq A$ then this trace simulates one in which the intruder fakes the message; if $A' = A$ then the intruder has redirected his own message: this simulates a trace in which the intruder sends the message to the correct agent in the first place.

Once we discount the simulating traces, we conclude that $A' = A$ and $B' = B$. This disjunct is now more restrictive than the corresponding disjunct in (†), and hence implies it.

2. The second possibility is that the intruder faked the message with the proxy's identity; this disjunct is already in the correct form for (†).

3. The final possibility is that the intruder hijacked a message sent by the simple proxy $P_{(A',B')}$ to the agent B'. If $P_{(A',B')} = P_{(A,B)}$, then necessarily $A' = A$; if $P_{(A,B)}$ is dishonest, A is also dishonest. Since we already know that either $B' = B$, or B' is dishonest, this disjunct matches that in (†).

We have shown that if an agent B receives a message from A's proxy $P_{(A,B)}$, then the set of events that may precede this receive event is a subset of those allowed by the alternative form of the proxy channel specification $C \wedge NRA^- \wedge NR^-$ on the channel $R_i \rightarrow R_j$. Therefore, $(C \wedge NRA^- \wedge NR^-)(\overrightarrow{Proxy_{(R_i,R_j)}})$ holds. □

While each instance of the theorem can be proved simply, there are 121 instances[3] that must be proved. In order to ease this process we have developed a Haskell [Bir88] script (see Appendix B.4) that performs the proofs automatically. In the rest of this section we describe the script, and relate its various stages to the proof example shown above.

Deriving the full set of trace patterns We first calculate the distinct trace patterns that result in an honest agent receiving a message, via a proxy, from another honest agent or the intruder. A trace pattern is a subtrace consisting of the events leading up to a receive event in which all identities, connection identifiers and message values are representative. For example, a trace pattern may show that an honest agent sends a message to their proxy, the proxy receives it and then sends it on, the intruder then redirects the message to another honest agent, and then the new recipient receives the message; e.g.

$$s \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.P_{(A,B)},\ \mathit{receive}.P_{(A,B)}.c_P.A.m,\ \mathit{send}.P_{(A,B)}.c'_P.B.m,\\ \mathit{hijack}.P_{(A,B)} \rightarrow P_{(A,B')}.B \rightarrow B'.c_{B'}.m,\ \mathit{receive}.B'.c_{B'}.P_{(A,B')}.m \rangle\ .$$

Applying the channel properties We apply the properties of the channels to and from the proxy to eliminate those trace patterns in which the intruder must perform an event that the channel does not allow him to. For example, if the intruder cannot fake on the channel to the proxy, we eliminate those trace patterns in which he fakes a message on this channel.

Determining the resultant channel specification We examine the remaining trace patterns to determine which capabilities the intruder still has. For example, if one of the remaining trace patterns shows that an honest agent A sent a message to an honest agent B, but then B receives that message from the dishonest agent I, then this pattern demonstrates that the intruder can re-ascribe messages with his own identity. When we examine each of the trace patterns we discover which events the intruder can perform; we then find the point in the lattice that corresponds to these remaining events, and collapse this point to a channel in the hierarchy.

We describe each of these three stages in more detail below.

Deriving the full set of trace patterns

To derive the full set of trace patterns, we do not assume that either the channel to the proxy or the channel from the proxy satisfy any secure channel

[3]There are 11 possibilities for the channel to the proxy, and 11 for the channel from the proxy.

specifications. Suppose that tr is a valid system trace such that for all prefixes $tr' \leqslant tr$, $\mathit{SimpleProxies}(R_i \to R_j)(tr')$ holds.

Let A and B be agents playing roles $\hat{R}_i$ and $\hat{R}_j$, $P_{(A,B)}$ be A's proxy to B, c_B a connection and m an application-layer message. Suppose that the event $\mathit{receive}.B.c_B.P_{(A,B)}.m$ occurs in tr; the network rule $\mathcal{N}_4$ implies the existence of one of several events earlier in the trace.

1. $\mathit{send}.P_{(A,B)}.c_P.B.m$ **in** tr for some connection c_P;
2. $\mathit{fake}.P_{(A,B)}.B.c_B.m$ **in** tr ;
3. $\exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)} \cdot$
   $\mathit{hijack}.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m$ **in** tr .

There are three different possibilities: either the proxy sent the message to B, the intruder faked the message to B (with the proxy's identity), or the intruder hijacked a message sent by a different proxy ($P_{(A',B')}$). Each of these events leads to different trace patterns, so we investigate them independently.

1. The event $\mathit{send}.P_{(A,B)}.c_P.B.m$ occurs in the trace for some connection c_P. Since B accepts this message, $\mathit{SimpleProxies}(R_i \to R_j)$ implies that $\mathit{SimpleProxy}(P_{(A,B)})$ holds for tr, and so $P_{(A,B)}$ must previously have received this message. The event $\mathit{receive}.P_{(A,B)}.c'_P.A.m$ occurs earlier in the trace, for some connection c'_P.

   We apply network rule $\mathcal{N}_4$ again; this implies the existence of one of several events earlier in the trace.

   (a) $\mathit{send}.A.c_A.P_{(A,B)}$ **in** tr for some connection c_A; the trace has the following pattern:

   $$tr_1 \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.P_{(A,B)}.m,\ \mathit{receive}.P_{(A,B)}.c'_P.A.m,\ \mathit{send}.P_{(A,B)}.c_P.B.m,\ \mathit{receive}.B.c_B.P_{(A,B)}.m \rangle .$$

   (b) $\mathit{fake}.A.P_{(A,B)}.c'_P.m$ **in** tr ; the trace has the following pattern:

   $$tr_2 \mathrel{\widehat{=}} \langle \mathit{fake}.A.P_{(A,B)}.c'_P.m,\ \mathit{receive}.P_{(A,B)}.c'_P.A.m,\ \mathit{send}.P_{(A,B)}.c_P.B.m,\ \mathit{receive}.B.c_B.P_{(A,B)}.m \rangle .$$

   The intruder does not fake messages with his own identity, so in this trace pattern we assume that A is honest.

   (c) $\exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)} \cdot$
   $\mathit{hijack}.A' \to A.P_{(A',B')} \to P_{(A,B)}.c'_P.m$ **in** tr .

The intruder can only hijack messages that were previously sent ($\mathcal{N}_2$), so A' must have sent the message to her proxy earlier in the trace (from some connection $c_{A'}$). The trace has the following pattern:

$$\begin{aligned} tr_3 \mathrel{\widehat{=}} \langle & \mathit{send}.A'.c'_A.P_{(A',B')}.m, \\ & \mathit{hijack}.A' \to A.P_{(A',B')} \to P_{(A,B)}.c'_P.m, \\ & \mathit{receive}.P_{(A,B)}.c'_P.A.m, \\ & \mathit{send}.P_{(A,B)}.c_P.B.m,\ \mathit{receive}.B.c_B.P_{(A,B)}.m \rangle\,. \end{aligned}$$

We can safely block the intruder from hijacking his own messages (Proposition 4.4.1), so in this trace pattern we assume that A' is honest. If $A' = A$ and $B' = B$, then this trace pattern just shows a replay on the channel to the proxy; because none of our channels prevent replays, we are not interested in whether or not the intruder has this capability. We assume that either $A' \neq A$ or $B' \neq B$.

2. The event $\mathit{fake}.P_{(A,B)}.B.c_B$ occurs in the trace. Since the intruder does not fake with dishonest identities, and since the proxy $P_{(A,B)}$ is honest if and only if A is honest, we assume that A is honest. The trace has the following pattern:

$$tr_4 \mathrel{\widehat{=}} \langle \mathit{fake}.P_{(A,B)}.B.c_B.m,\ \mathit{receive}.B.c_B.P_{(A,B)}.m \rangle\,.$$

3. The event $\mathit{hijack}.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m$ occurs in the trace. We apply $\mathcal{N}_2$ again: the proxy $P_{(A',B')}$ must have sent the message to B' earlier in the trace (in some connection $c_{P'}$). So far, the trace has this pattern:

$$\begin{aligned} s \mathrel{\widehat{=}} \langle & \mathit{send}.P_{(A',B')}.c_{P'}.B'.m, \\ & \mathit{hijack}.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m,\ \mathit{receive}.B.c_B.P_{(A,B)}.m \rangle\,. \end{aligned}$$

As before, we assume that $P_{(A',B')}$ is honest (and hence A' is honest). Now $\mathit{SimpleProxies}(R_i \to R_j)$ applies again and implies that $\mathit{SimpleProxy}(P_{(A',B')})$ holds for tr, and so $P_{(A',B')}$ must previously have received this message from A' (in some connection $c'_{P'}$). The trace now has the following pattern:

$$\begin{aligned} s \mathrel{\widehat{=}} \langle & \mathit{receive}.P_{(A',B')}.c'_{P'}.A',\ \mathit{send}.P_{(A',B')}.c_{P'}.B'.m, \\ & \mathit{hijack}.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m,\ \mathit{receive}.B.c_B.P_{(A,B)}.m \rangle\,. \end{aligned}$$

We apply network rule $\mathcal{N}_4$ again; this implies the existence of one of several events earlier in the trace.

(a) $send.A'.c_{A'}.P_{(A',B')}.m$ **in** tr for some connection $c_{A'}$; the trace has the following pattern:

$$\begin{aligned} tr_5 \mathrel{\widehat{=}} \langle & send.A'.c_{A'}.P_{(A',B')}.m,\ receive.P_{(A',B')}.c'_{P'}.A', \\ & send.P_{(A',B')}.c_{P'}.B'.m, \\ & hijack.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m, \\ & receive.B.c_B.P_{(A,B)}.m \rangle \,. \end{aligned}$$

As before, we assume that A' is honest and that either $A' \neq A$ or $B' \neq B$.

(b) $fake.A'.P_{(A',B')}.c'_{P'}.m$ **in** tr. If the intruder can fake with A''s identity, he can also fake with A's identity; this means that the hijack event on the channel from the proxy is unnecessary. We ignore this trace pattern.

(c) $\exists A'' : \hat{R}_i; B'' : \hat{R}_j; P_{(A'',B'')} : \widehat{Proxy}_{(R_i,R_j)} \cdot$
$hijack.A'' \rightarrow A'.P_{(A'',B'')} \rightarrow P_{(A',B')}.c'_{P'}.m$ **in** tr .
We apply $\mathcal{N}_2$ again: the agent A'' must have sent the message to her proxy in order for the intruder to hijack it. The trace now has the following pattern:

$$\begin{aligned} tr_6 \mathrel{\widehat{=}} \langle & send.A''.c_{A''}.P_{(A'',B'')}.m, \\ & hijack.A'' \rightarrow A'.P_{(A'',B'')} \rightarrow P_{(A',B')}.c'_{P'}.m, \\ & receive.P_{(A',B')}.c'_{P'}.A',\ send.P_{(A',B')}.c_{P'}.B'.m, \\ & hijack.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m, \\ & receive.B.c_B.P_{(A,B)}.m \rangle \,. \end{aligned}$$

If $A'' \neq A'$ and $A' \neq A$ then the intruder has re-ascribed the message to A' on the channel to the proxy; he then re-ascribes it to A on the channel from the proxy. This is unnecessary activity, and this trace pattern simulates one in which he does not hijack on the channel to the proxy. We therefore assume that either $A'' = A'$ or $A' = A$; similarly we assume that either $B'' = B'$ or $B' = B$. We ignore the trace patterns in which $A'' = A'$ and $B'' = B'$, or $A' = A$ and $B' = B$); we assume that A'' is honest.

Applying the channel properties

The Haskell script automatically generates all of the trace patterns described above. It then takes each of the 121 combinations of channels to and from the proxy and uses the channel specifications to eliminate those traces that are not allowed. The traces are calculated in two stages: the set of activity on the channel from the proxy is calculated first, and then the channel specification on the channel from the proxy is applied; then the activity on the channel to the proxy is calculated, and finally the channel specification on the channel to the proxy is applied. The result is a set of trace patterns $\{tr_1, tr_2, \ldots, tr_n\}$ that are allowed on the resultant channel.

Determining the resultant channel specification

In order to determine which channel property the resultant channel satisfies we examine each of the co-ordinate points in the lattice individually. The resultant channel is confidential (the first co-ordinate point) if and only if the channels to and from the proxy are confidential; we call this value c. We determine the remaining three co-ordinate points (no-faking, no-re-ascribing and no-redirecting) by looking at each of the allowed trace patterns.

Each of these trace patterns demonstrates that the intruder can perform a particular event. We present below a summary of these patterns; each case shows the initial and final events in the pattern and then describes which activity this demonstrates. This mapping allows us to assign a tuple (nf, nra, nr) to each trace pattern; this tuple represents the strongest possible specification that this trace allows.

When the final event is $receive.B.c_B.P_{(A,B)}.m$ (agent B receives a message from the honest agent A's proxy):

send$.A.c_A.P_{(A,B)}.m$ this does not demonstrate any hijacking activity: (NF, NRA, NR);

send$.A.c_A.P_{(A,I)}.m$ the intruder redirected a message that was sent to a dishonest agent: (NF, NRA, NR^{-});

send$.A.c_A.P_{(A,B')}.m$ the intruder redirected a message that was sent to an honest agent: $(NF, NRA, \bot)$;

send$.A'.c_{A'}.P_{(A',B)}.m$ the intruder re-ascribed a message to an honest agent: $(NF, \bot, NR)$;

send$.A'.c_{A'}.P_{(A',I)}.m$ the intruder re-ascribed a message to an honest agent, and redirected a message sent to a dishonest agent: $(NF, \bot, NR^{-})$;

send$.A'.c_{A'}.P_{(A',B')}.m$ the intruder re-ascribed a message to an honest agent, and redirected a message that was sent to an honest agent: $(NF, \bot, \bot)$;

send$.I.c_I.P_{(I,B)}.m$ the intruder hijacked his own message to simulate a fake: $(\bot, NRA, NR)$;

fake$.A.P_{(A,B)}.c_P.m$ the intruder faked a message to the proxy: $(\bot, NRA, NR)$;

fake$.P_{(A,B)}.B.c_B.m$ the intruder faked a message from the proxy: $(\bot, NRA, NR)$.

When the final event is $receive.B.c_B.P_{(I,B)}.m$ (agent B receives a message from the dishonest agent I's proxy):

send$.I.c_I.P_{(I,B)}.m$ this does not demonstrate any activity: (NF, NRA, NR);

send$.A.c_A.P_{(A,B)}.m$ the intruder re-ascribed a message to a dishonest agent: (NF, NRA^-, NR);

send$.A.c_A.P_{(A,I)}.m$ this does not demonstrate any activity (the intruder sent a message that was sent to him): (NF, NRA, NR);

send$.A.c_A.P_{(A,B')}.m$ the intruder re-ascribed a message to a dishonest agent, and redirected a message sent to an honest agent; however, this is only possible on a non-confidential channel, so this simulates a learn and send: (NF, NRA, NR).

The result of applying the mapping to each of the trace patterns is a set of tuples of the form (nf_i, nra_i, nr_i) for $i = 1 \ldots n$. We take the value of the confidential co-ordinate and calculate the lattice point of the resultant channel in the following way (where **min** is calculated according to the order on the building blocks — see Chapter 4):

$$(c, nf, nra, nr) = (c, \mathbf{min}_{i=1}^{n}(nf_i), \mathbf{min}_{i=1}^{n}(nra_i), \mathbf{min}_{i=1}^{n}(nr_i)) .$$

Finally we collapse this point to a point in the channel hierarchy. Thus the resultant channel satisfies the channel specification:

$$(c, nf, nra, nr) =\downarrow (c, \mathbf{min}_{i=1}^{n}(nf_i), \mathbf{min}_{i=1}^{n}(nra_i), \mathbf{min}_{i=1}^{n}(nr_i)) .$$

By eliminating trace patterns and calculating the resultant point in the hierarchy in this manner we prove that the alternative specification of the resultant channel holds on all valid system traces in which the channel specifications to and from the proxy hold. The full list of results, and the Haskell script listing are shown in Appendix B.

5.2 Multiplexing proxies

In this section we consider the more general (multiplexing) proxy case. The study of simple proxies shows that by chaining two secure channels through a trusted third party one can sometimes produce a stronger channel. However, in the simple case, we thought of the proxies as 'belonging' to one of the agents communicating; it is highly likely that A trusts her proxies, but should she trust other agents' proxies who send messages to her? In this section we consider more general multiplexing proxies. A multiplexing proxy is a trusted third party who is willing to forward messages from any agent to any other agent.

We assume that all multiplexing proxies are honest. There is nothing to stop the intruder from setting up proxies of his own; however, any message

sent through a dishonest proxy cannot remain confidential, and any message received from a dishonest proxy cannot be authenticated.

When agent A intends to send a message to another agent (B) through a simple proxy she just needs to pick the correct simple proxy to send the message to. The proxy knows whom to forward the message to because it is dedicated to that job. If A is to use a multiplexing proxy, she must communicate her intent (to talk to B) to the proxy. Similarly, when B receives a message from A's proxy, he knows who originally sent the message; when B receives a message from a multiplexing proxy, there must be some communication from the proxy to B to say whom the message is from.

One way to solve this problem would be to build a special transport-layer protocol in which the message sender's protocol agent tells the proxy whom to establish a connection with. The agent may then just send messages to the proxy (just as they would if they were sending the messages directly to the recipient). Similarly, the proxy tells the recipient's protocol agent whom the messages are from. However, this solution is unsuitable for our model for two reasons:

- The whole point of the model is to make the details of the transport-layer protocol abstract; once we start to impose conditions on the transport-layer protocol, we lose the generality of the abstract model;
- When we discussed simple proxies we argued the case for considering the proxies as application-layer entities; we make the same argument here as we wish all details of the discussion to be at the application-layer.

The solution we adopt, therefore, is to annotate the application-layer messages with information about whom they are intended for, and whom they were originally sent by. In order to send a message m to B (via the multiplexing proxy P), agent A concatenates B's identity to the message:

$$\mathit{send}.A.c_A.P.\langle m, B\rangle\,.$$

When P receives this message he concatenates it to A's identity, and sends it on to B:

$$\mathit{send}.P.c_P.B.\langle A, m\rangle\,.$$

This only works if the channel is either confidential or non-fakeable; however, all of our channels satisfy at least one of these two properties, so this method can be used on all of our channels.

We assume that none of the application-layer protocols call for agents to send messages with the same type as the messages described above. If we do not make this assumption, it might be possible for messages created by honest agents for use in the application-layer protocols to be mistaken for messages sent to or from a multiplexing proxy.

Definition 5.2.1. A *multiplexing proxy* is an honest agent who is dedicated to forwarding messages; there is a single proxy role *Proxy*. When two roles communicate through a multiplexing proxy, the following trace specification is satisfied:

$$
\begin{aligned}
&\mathit{Proxies}(R_i \to R_j)(tr) \mathrel{\widehat{=}} \\
&\quad tr \downarrow \{\!|\, \mathit{send}.\hat{R}_i.\mathit{Connection}.\hat{R}_j \,|\!\} = \langle\rangle \wedge \\
&\quad tr \downarrow \{\!|\, \mathit{receive}.\hat{R}_j.\mathit{Connection}.\hat{R}_i \,|\!\} = \langle\rangle \wedge \\
&\quad \forall A, B, P : \mathit{Agent}; c_A : \mathit{Connection}; m : \mathit{Message} \cdot \\
&\qquad \mathit{send}.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \Rightarrow \mathit{Proxy}(P)(tr) \wedge \\
&\quad \forall A, B, P : \mathit{Agent}; c_B : \mathit{Connection}; m : \mathit{Message} \cdot \\
&\qquad \mathit{receive}.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \mathit{Proxy}(P)(tr)\,,
\end{aligned}
$$

where each multiplexing proxy satisfies the following specification:

$$
\begin{aligned}
&\mathit{Proxy}(P)(tr) \mathrel{\widehat{=}} \\
&\quad \mathit{Honest}(P) \wedge \\
&\quad \forall c_P : \mathit{Connection}; A, B, B' : \mathit{Agent}; m, m' : \mathit{Message} \cdot \\
&\qquad \mathit{receive}.P.c_P.A.\langle m, B\rangle \textbf{ in } tr \wedge \mathit{receive}.P.c_P.A.\langle m', B'\rangle \textbf{ in } tr \Rightarrow B = B' \wedge \\
&\quad \forall c_P : \mathit{Connection}; A, A', B : \mathit{Agent}; m, m' : \mathit{Message} \cdot \\
&\qquad \mathit{send}.P.c_P.B.\langle A, m\rangle \textbf{ in } tr \wedge \mathit{send}.P.c_P.B.\langle A', m'\rangle \textbf{ in } tr \Rightarrow A = A' \wedge \\
&\quad \forall c_P : \mathit{Connection}; A : \mathit{Agent} \cdot \exists c'_P : \mathit{Connection}; B : \mathit{Agent} \cdot \\
&\qquad \mathit{send}.P.c_P.\langle A, m\rangle \leqslant \mathit{receive}.P.c'_P.\langle m, B\rangle\,.
\end{aligned}
$$

Each connection that the multiplexing proxies establish is either dedicated to receiving messages from one agent, or sending messages to one agent. Although two individual messages from one agent to another could be sent through different proxies, we assume that all the messages in one connection are sent to (or received from) the same proxy.

We assume that the honest agents only send messages of the form $\langle m, B\rangle$ to proxies, and that they will only receive messages of the form $\langle A, m\rangle$ from proxies.

Each multiplexing proxy can be used by several agents. One can imagine a scenario in which each organisation has a pool of multiplexing proxies: every agent in that organisation communicates with external agents through the proxies, but communicates directly with internal agents (see Figure 5.2).

Whenever a multiplexing proxy receives a message from A for B, he forwards it to B, and tells B that it is from A. In each connection, the proxies only exchange messages with one other agent; they also only allow one third party to be involved in each connection. This restriction on the proxies' behaviour is not necessary for the proof of invariance of the single message channels, but it is necessary to prove that the session properties are invariant under chaining; we discuss how this might be done in Chapter 8.

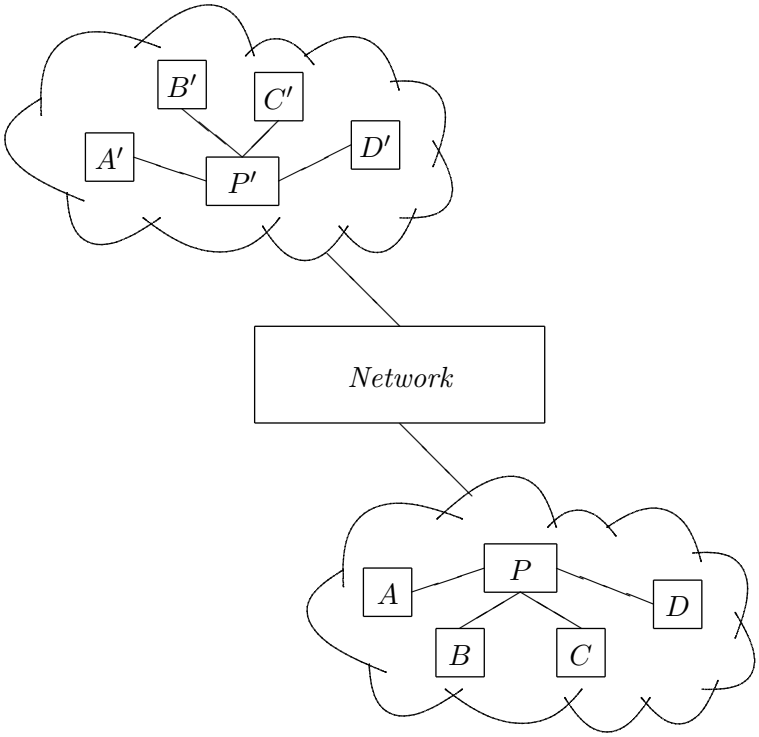


Figure 5.2: Multiplexing proxies.

5.2.1 Secure channels through multiplexing proxies

The public knowledge of the role of each simple proxy was what led to the rather surprising result that the chained form of two channels can be stronger than both channels individually. With the multiplexing proxies we no longer have this public knowledge; B only knows whom the message was originally sent by by examining it and seeing whose identity is attached to it. As we did last time, we consider each of the components of the hierarchy individually in order to discover which properties the channel through a proxy satisfies. In the discussion below we refer to the channel to the proxy as $(R_i \rightarrow \mathit{Proxy})$ and the channel from the proxy as $(\mathit{Proxy} \rightarrow R_j)$; we refer to the overall channel through the proxy as $\mathit{Proxy}(R_i \rightarrow R_j)$.

Confidentiality If either of the channels to or from a proxy is not confidential, then the channel through the proxy is not confidential. Since all multiplexing proxies are honest, the channel through the proxy is confidential if and only if the channels to and from the proxy are confidential.

No faking It is clear that if either of the channels to or from a proxy is fakeable, then the channel through the proxy is fakeable. In order to fake a message from A to B, the intruder must either fake sending the message to the proxy, or from the proxy.

No re-ascribing Unlike the simple proxies, the intruder cannot choose which channel to re-ascribe a message on: he must do so on the channel to the proxy. This is straightforward:

$$tr \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.P.\langle m, B\rangle, \mathit{hijack}.A \rightarrow A'.P.c_P.\langle m, B\rangle\rangle\,.$$

The only identity that the intruder can change by re-ascribing on the channel from the proxy is that of the message sender (the proxy):

$$\begin{aligned} tr \mathrel{\widehat{=}} \langle & \mathit{send}.A.c_A.P.\langle m, B\rangle, \mathit{receive}.P.c_P.A.\langle m, B\rangle, \\ & \mathit{send}.P.c'_P.B.\langle A, m\rangle, \mathit{hijack}.P \rightarrow P'.B.c_B.\langle A, m\rangle\rangle\,. \end{aligned}$$

Because honest agents only accept messages of the form $\langle A, m\rangle$ from proxies, the intruder can only re-ascribe the message to a different proxy: he cannot change the identity of the original sender of the message by re-ascribing the message on the channel from the proxy.

No redirecting The intruder can only redirect a message using the channel from the proxy; this is straightforward:

$$\begin{aligned} tr \mathrel{\widehat{=}} \langle & \mathit{send}.A.c_A.P.\langle m, B\rangle, \mathit{receive}.P.c_P.A.\langle m, B\rangle, \\ & \mathit{send}.P.c'_P.B.\langle A, m\rangle, \mathit{hijack}.P.B \rightarrow B'.c_{B'}.\langle A, m\rangle\rangle\,. \end{aligned}$$

The only identity that the intruder can change by redirecting the message on the channel to the proxy is that of the message recipient: in this case, the proxy;

$$tr \mathrel{\widehat{=}} \langle send.A.c_A.P.\langle m, B\rangle, hijack.A.P \rightarrow P'.c_{P'}.\langle m, B\rangle\rangle\,.$$

Because the only honest agents who receive messages of the form $\langle m, B\rangle$ are proxies, the intruder can only redirect the message to a different proxy.

We note that because the application-layer messages to and from the proxy now contain some routing information, the intruder can redirect or re-ascribe a message (i.e. change the identity of the *original* message sender or the *ultimate* message recipient) by faking a message. In this section we always use the terms no faking, no-re-ascribing and no-redirecting to refer to the properties that block the intruder from performing the fake and hijack events, but we will sometimes talk about the intruder faking a message to re-ascribe or to redirect a message sent from one honest agent to another. In order to do this the intruder must have learned the application-layer message, so this is only possible on non-confidential channels.

The *Proxies* property on the roles R_i and R_j prevents agents playing role R_i from communicating directly with agents playing role R_j. As before, we must reframe the definitions of the authenticated channel building blocks for the channel through a multiplexing proxy because the standard definitions are vacuously satisfied.

Definition 5.2.2 (No faking)**.**

$$\begin{aligned}&NF(Proxy(R_i \rightarrow R_j))(tr) \mathrel{\widehat{=}}\\ &\quad tr \upharpoonright \{|\, fake.A.P.c_A.\langle m, B\rangle,\ fake.P.c_P.B.\langle A, m\rangle \mid\\ &\qquad\qquad A \in \hat{R}_i \wedge P \in \widehat{Proxy} \wedge B \in \hat{R}_j \wedge\\ &\qquad\qquad c_A, c_P \in Connection \wedge m \in Message_{App}\,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.2.3 (No-re-ascribing)**.**

$$\begin{aligned}&NRA(Proxy(R_i \rightarrow R_j))(tr) \mathrel{\widehat{=}}\\ &\quad tr \upharpoonright \{|\, hijack.A \rightarrow A'.P \rightarrow P'.c_{P'}.\langle m, B\rangle \mid\\ &\qquad\qquad A, A' \in \hat{R}_i \wedge P, P' \in \widehat{Proxy} \wedge B \in \hat{R}_j \wedge c_{P'} \in Connection \wedge\\ &\qquad\qquad m \in Message_{App} \wedge A \neq A'\,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.2.4 (No-honest-re-ascribing)**.**

$$\begin{aligned}&NRA^{-}(Proxy(R_i \rightarrow R_j))(tr) \mathrel{\widehat{=}}\\ &\quad tr \upharpoonright \{|\, hijack.A \rightarrow A'.P \rightarrow P'.c_{P'}.\langle m, B\rangle \mid\\ &\qquad\qquad A, A' \in \hat{R}_i \wedge P, P' \in \widehat{Proxy} \wedge B \in \hat{R}_j \wedge c_{P'} \in Connection \wedge\\ &\qquad\qquad m \in Message_{App} \wedge A \neq A' \wedge Honest(A')\,|\} = \langle\rangle\,.\end{aligned}$$

Definition 5.2.5 (No-redirecting)**.**

$$
\begin{array}{l}
NR(Proxy(R_i \rightarrow R_j))(tr) \mathrel{\widehat{=}} \\
\quad tr \upharpoonright \{| \, hijack.P \rightarrow P'.B \rightarrow B'.c_{B'}.\langle A, m\rangle \mid \\
\qquad\qquad A \in \hat{R}_i \wedge P, P' \in \widehat{Proxy} \wedge B, B' \in \hat{R}_j \wedge c_{B'} \in Connection \wedge \\
\qquad\qquad m \in Message_{App} \wedge B \neq B' \, |\} = \langle\rangle \, .
\end{array}
$$

Definition 5.2.6 (No-honest-redirecting)**.**

$$
\begin{array}{l}
NR^{-}(Proxy(R_i \rightarrow R_j))(tr) \mathrel{\widehat{=}} \\
\quad tr \upharpoonright \{| \, hijack.P \rightarrow P'.B \rightarrow B'.c_{B'}.\langle A, m\rangle \mid \\
\qquad\qquad A \in \hat{R}_i \wedge P, P' \in \widehat{Proxy} \wedge B, B' \in \hat{R}_j \wedge c_{B'} \in Connection \wedge \\
\qquad\qquad m \in Message_{App} \wedge B \neq B' \wedge Honest(B) \, |\} = \langle\rangle \, .
\end{array}
$$

As in the proof of the simple chaining theorem, in the proof of the chaining theorem, below, we show that the alternative specifications for each of the channels in the hierarchy through a proxy are satisfied. These forms of the alternative specifications take account of the fact that messages can be faked or hijacked on the channel to the proxy or on the channel from the proxy; the specifications are shown in Appendix B. We present one example below; the alternative form of the channel specification $(C \wedge NRA^{-} \wedge NR^{-})(Proxy(R_i \rightarrow R_j))$ is:

$$
\begin{array}{l}
Alt(C \wedge NRA^{-} \wedge NR^{-})(Proxy(R_i \rightarrow R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow Proxy) \wedge C(Proxy \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad fake.P.B.c_B.\langle A, m\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists P' : \widehat{Proxy}; B' : \hat{R}_j; c_B : Connection \cdot \\
\qquad\qquad hijack.P' \rightarrow P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr \wedge \\
\qquad\qquad ((P' = P) \vee Dishonest(P)) \wedge ((B' = B) \vee Dishonest(B')) \vee \\
\qquad\quad \exists A' : \hat{R}_i; P' : \widehat{Proxy}; c_P : Connection \cdot \\
\qquad\qquad hijack.A' \rightarrow A.P' \rightarrow P.c_P.\langle m, B\rangle \textbf{ in } tr \wedge \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((P' = P) \vee Dishonest(P')) \, .
\end{array}
$$

In Appendix B.2 we always assume that $P' = P$, since honest agents only accept messages from honest proxies, and only send messages to honest proxies, so $Dishonest(P)$ and $Dishonest(P')$ never hold.

5.2.2 Chaining theorem

We make the following observations of the overall channel through a multiplexing proxy.

Observation 5.2.7. The intruder cannot redirect messages using the channel to the proxy. Subject to the collapsing cases described earlier, the channel to the proxy satisfies *NR*.

Observation 5.2.8. The intruder cannot re-ascribe messages using the channel from the proxy. Subject to the collapsing cases described earlier, the channel from the proxy satisfies *NRA*.

Theorem 5.2.9 (Chaining theorem)**.** *If roles R_i and R_j communicate through multiplexing proxies (i.e. Proxies$(R_i \to R_j)$) on secure channels such that:*

$$\begin{aligned} &\mathit{ChannelSpec}_1(R_i \to \mathit{Proxy})\,, \\ &\mathit{ChannelSpec}_2(\mathit{Proxy} \to R_j)\,, \end{aligned}$$

where $\mathit{ChannelSpec}_1$ and $\mathit{ChannelSpec}_2$ are channels in the hierarchy, then the overall channel (through the proxy) satisfies the channel specification:

$$\mathit{ChannelSpec} =\downarrow (\nwarrow_m \mathit{ChannelSpec}_1 \sqcap \nearrow_m \mathit{ChannelSpec}_2)\,;$$

where:

$$\begin{aligned} \nwarrow_m (c, nf, nra, nr) &= (c, nf, nra, NR)\,, \\ \nearrow_m (c, nf, nra, nr) &= (c, nf, NRA, nr)\,, \end{aligned}$$

and $\sqcap$ is the greatest lower bound operator in the full lattice.

The proof of the chaining theorem is in Section 5.2.3.

Corollary 5.2.10. *If roles R_i and R_j communicate through multiplexing proxies (i.e. Proxies$(R_i \to R_j)$) on secure channels such that:*

$$\begin{aligned} &\mathit{ChannelSpec}(R_i \to \mathit{Proxy})\,, \\ &\mathit{ChannelSpec}(\mathit{Proxy} \to R_j)\,, \end{aligned}$$

where ChannelSpec is a channel in the hierarchy, then the overall channel (through the proxy) satisfies a channel specification $\mathit{ChannelSpec}'$ which is such that:

$$\mathit{ChannelSpec} \preccurlyeq \mathit{ChannelSpec}'\,.$$

In particular, $\mathit{ChannelSpec}(\mathit{Proxy}(R_i \to R_j))$ holds.

A simple case analysis in the case of the multiplexing proxies shows that:

$$\mathit{ChannelSpec}' = \mathit{ChannelSpec}\,.$$

Example 5.2.11. The channel to the proxy satisfies $C \wedge NF \wedge NRA^- \wedge NR^-$, and the channel from the proxy satisfies $C \wedge NRA^- \wedge NR^-$.

$$\begin{aligned} \nwarrow_m (C \wedge NF \wedge NRA^- \wedge NR^-) &= C \wedge NF \wedge NRA^- \wedge NR\,, \\ \nearrow_m (C \wedge NRA \wedge NR^-) &= C \wedge NRA \wedge NR^-\,. \end{aligned}$$

The greatest lower bound of these two points is $C \wedge NRA^{-} \wedge NR^{-}$; this is the greatest lower bound of the two channels. This is the same result as for the simple proxies.

The intruder cannot re-ascribe messages to honest agents because the channel from the proxy is only re-ascribable with dishonest identities; even though the channel from the proxy is fakeable, both channels are confidential, so the intruder cannot learn the message and fake it to effect a re-ascribe. The intruder can redirect messages that are sent to him.

Example 5.2.12. The channel to the proxy satisfies $NF \wedge NRA^{-}$, and the channel from the proxy satisfies $NF \wedge NRA^{-} \wedge NR^{-}$.

$$\begin{aligned} \nwarrow_m (NF \wedge NRA^{-}) &= NF \wedge NRA^{-} \wedge NR\,, \\ \nearrow_m (NF \wedge NRA^{-} \wedge NR^{-}) &= NF \wedge NRA \wedge NR^{-}\,. \end{aligned}$$

The greatest lower bound of these two points is $NF \wedge NRA^{-} \wedge NR^{-}$; this channel is stronger than the greatest lower bound of the two channels. This result is different to the simple proxy result (which is just $NF \wedge NRA^{-}$).

Neither channel is confidential, so the overall channel is not confidential. However, because neither channel is fakeable, the intruder cannot overhear messages and fake them to re-ascribe or redirect messages. He cannot therefore redirect messages using the channel to the proxy, and cannot re-ascribe messages using the channel from the proxy.

Example 5.2.13. The channel to the proxy satisfies $C \wedge NF \wedge NRA \wedge NR^{-}$, and the channel from the proxy satisfies $C \wedge NF \wedge NRA^{-} \wedge NR$.

$$\begin{aligned} \nwarrow_m (C \wedge NF \wedge NRA \wedge NR^{-}) &= C \wedge NF \wedge NRA \wedge NR\,, \\ \nearrow_m (C \wedge NF \wedge NRA^{-} \wedge NR) &= C \wedge NF \wedge NRA \wedge NR\,. \end{aligned}$$

The greatest lower bound of these two points is the top channel; this is stronger than both channels.

The intruder cannot redirect messages, nor can he re-ascribe messages on the overall channel because these activities are blocked on the only channel that allows them.

The list of resultant channels for every instance of the chaining theorem is shown in Figure B.2 (in Appendix B.2).

5.2.3 An automated proof of the chaining theorem

As before, each instance of Theorem 5.2.9 is relatively simple to prove; the proof mechanism is identical to that for the simple proxies, only the details of the proof are different. As in the previous section, we first prove an example instance of the theorem, then we describe the changes to the automated proof.

Lemma 5.2.14. *If roles R_i and R_j communicate through multiplexing proxies (i.e. Proxies$(R_i \to R_j)$) on secure channels such that:*

$$(NF \wedge NRA^-)(R_i \to Proxy)\,,$$
$$(NF \wedge NRA^- \wedge NR^-)(Proxy \to R_j)\,,$$

then $(NF \wedge NRA^- \wedge NR^-)(Proxy(R_i \to R_j))$.

Proof. We use the *Proxies* property and the alternative specifications of the channels to and from the proxy to show that the alternative form of $(NF \wedge NRA^- \wedge NR^-)(Proxy(R_i \to R_j))$ holds; i.e.

$$\begin{array}{l}
\forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\quad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad \exists P' : \widehat{Proxy}; B' : \hat{R}_j; c_B : Connection \cdot \\
\qquad\quad hijack.P' \to P.B' \to B.c_B.\langle A, m\rangle \textbf{ in } tr \wedge \qquad (\dagger) \\
\qquad\quad ((P' = P) \vee Dishonest(P)) \wedge ((B' = B) \vee Dishonest(B')) \vee \\
\qquad \exists A' : \hat{R}_i; P' : \widehat{Proxy}; c_P : Connection \cdot \\
\qquad\quad hijack.A' \to A.P' \to P.c_P.\langle m, B\rangle \textbf{ in } tr \wedge \\
\qquad\quad ((A' = A) \vee Dishonest(A)) \wedge ((P' = P) \vee Dishonest(P'))\,,
\end{array}$$

for all valid system traces tr such that for all prefixes $tr' \leqslant tr$:

$$Proxies(R_i \to R_j)(tr')\,,$$
$$(NF \wedge NRA^-)(R_i \to Proxy)(tr')\,,$$
$$(NF \wedge NRA^- \wedge NR^-)(Proxy \to R_j)(tr')\,.$$

Let A and B be agents playing roles R_i and R_j, P be a multiplexing proxy, c_B a connection and m an application-layer message. Suppose that the event $receive.B.c_B.P.\langle m, A\rangle$ occurs in tr; the network rule $\mathcal{N}_4$ implies the existence of one of several events earlier in the trace. The set of possible events is limited by $NF \wedge NRA^- \wedge NR^-$, which holds on the channel from the proxy:

1. $\exists c_P : Connection \cdot send.P.c_P.B.\langle A, m\rangle \textbf{ in } tr$;
2. $\exists P' : \widehat{Proxy}; B' : \hat{R}_j \cdot hijack.P' \to P.B' \to B.c_B.\langle A, m\rangle \textbf{ in } tr \wedge$
   $((P' = P) \vee Dishonest(P)) \wedge ((B' = B) \vee Dishonest(B'))$.

The second of these disjuncts is already in the correct form for $(\dagger)$, so we only need to investigate the first.

Suppose that the event $send.P.c_P.B.\langle A, m\rangle$ precedes the receive event in the trace tr, for some connection identifier c_P. $Proxies(R_i \to R_j)$ implies $Proxy(P)$, and so $receive.P.c'_P.A.\langle m, B\rangle$ occurs earlier in the trace, for some connection c'_P.

We use network rule $\mathcal{N}_4$ again; this implies the existence of one of several events earlier in the trace, and, as before, these possibilities are limited by $NF \wedge NRA^-$, which holds on the channel to the proxy.

1. $\exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.P.\langle m, B\rangle$ **in** tr ;

2. $\exists A' : \hat{R}_i; P' : \widehat{\mathit{Proxy}} \cdot \mathit{hijack}.A' \rightarrow A.P' \rightarrow P.c_P.\langle m, B\rangle \wedge$
   $((A' = A) \vee \mathit{Dishonest}(A)) \wedge ((P' = P) \vee \mathit{Dishonest}(P'))$.

These disjuncts match those in (†).

We have shown that if an agent B receives a message from agent A via a multiplexing proxy, then the set of events that may precede this receive event is equal to those allowed by the alternative form of the proxy channel specification $NF \wedge NRA^- \wedge NR^-$ on the channel $R_i \rightarrow R_j$. Therefore $(NF \wedge NRA^- \wedge NR^-)(\mathit{Proxy}(R_i \rightarrow R_j))$ holds. □

There are, again, 121 instances of this theorem to prove. In order to prove these instances we adapt the automated proof of the simple chaining theorem; we describe these changes below; the full Haskell script is shown in Appendix B.5.

Deriving the full set of trace patterns

We derive the full set of trace patterns in the same way as before: we do not assume that either the channel to the proxy or the channel from the proxy satisfy any secure channel specifications. Suppose that tr is a valid system trace such that for all prefixes $tr' \leqslant tr$, $\mathit{Proxies}(R_i \rightarrow R_j)(tr')$ holds.

Let A and B be agents playing roles $\hat{R}_i$ and $\hat{R}_j$, P be a multiplexing proxy, c_B a connection and m an application-layer message. Suppose that $\mathit{receive}.B.c_B.P.\langle A, m\rangle$ occurs in tr; the network rule $\mathcal{N}_4$ implies the existence of one of several events earlier in the trace.

1. $\mathit{send}.P.c_P.B.\langle A, m\rangle$ **in** tr for some connection c_P;

2. $\mathit{fake}.P.B.c_B.\langle A, m\rangle$ **in** tr ;

3. $\exists P' : \widehat{\mathit{Proxy}}; B' : \hat{R}_j \cdot \mathit{hijack}.P' \rightarrow P.B' \rightarrow B.c_B.\langle A, m\rangle$ **in** tr .

There are three different possibilities: either the proxy sent the message to B, the intruder faked the message to B (with the proxy's identity), or the intruder hijacked a message sent by a different proxy to agent B'. Each of these events leads to different trace patterns, so we investigate them independently.

1. The event $\mathit{send}.P.c_P.B.\langle A, m\rangle$ occurs in the trace for some connection c_P. Since B accepts this message, $\mathit{Proxies}(R_i \rightarrow R_j)$ implies that $\mathit{Proxy}(P)$ holds for tr, and so P must previously have received a message of the form $\langle m, B\rangle$. The event $\mathit{receive}.P.c'_P.A.\langle m, B\rangle$ occurs earlier in the trace, for some connection c'_P.

   We apply network rule $\mathcal{N}_4$ again; this implies the existence of one of several events earlier in the trace.

(a) $\mathit{send}.A.c_A.P.\langle m, B\rangle$ **in** tr for some connection c_A; the trace has the following pattern:

$$tr_1 \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.P.\langle m, B\rangle,\ \mathit{receive}.P.c'_P.A.\langle m, B\rangle,\\ \mathit{send}.P.c_P.B.\langle A, m\rangle,\ \mathit{receive}.B.c_B.P.\langle A, m\rangle\rangle\,.$$

(b) $\mathit{fake}.A.P.c'_P.\langle m, B\rangle$ **in** tr ; the trace has the following pattern:

$$tr_2 \mathrel{\widehat{=}} \langle \mathit{fake}.A.P.c'_P.\langle m, B\rangle,\ \mathit{receive}.P.c'_P.A.\langle m, B\rangle,\\ \mathit{send}.P.c_P.B.\langle A, m\rangle,\ \mathit{receive}.B.c_B.P.\langle A, m\rangle\rangle\,.$$

The intruder does not fake messages with his own identity, so in this trace pattern we assume that A is honest.

(c) $\exists A' : \hat{R}_i; P' : \widehat{Proxy} \cdot \mathit{hijack}.A' \rightarrow A.P' \rightarrow P.c'_P.\langle m, B\rangle$ **in** tr . The intruder can only hijack messages that were previously sent ($\mathcal{N}_2$), so A' must have sent the message to P' earlier in the trace (in some connection $c_{A'}$). The trace has the following pattern:

$$tr_3 \mathrel{\widehat{=}} \langle \mathit{send}.A'.c_{A'}.P'.\langle m, B\rangle,\\ \mathit{hijack}.A' \rightarrow A.P' \rightarrow P.c'_P.\langle m, B\rangle,\ \mathit{receive}.P.c'_P.A.\langle m, B\rangle,\\ \mathit{send}.P.c_P.B.\langle A, m\rangle,\ \mathit{receive}.B.c_B.P.\langle A, m\rangle\rangle\,.$$

We can safely block the intruder from hijacking his own messages (Proposition 4.4.1), so in this trace pattern we assume that A' is honest. The intruder does not gain anything by redirecting the message to a different proxy (since all proxies are honest, and all proxies behave in the same way), so we assume that $P' = P$. If $A' = A$ then this trace pattern just shows a replay on the channel to the proxy; because none of our channels prevent replays, we are not interested in whether or not the intruder has this capability; we assume that $A' \neq A$.

2. The event $\mathit{fake}.P.B.c_B.\langle A, m\rangle$ occurs in the trace; the trace has the following pattern:

$$tr_4 \mathrel{\widehat{=}} \langle \mathit{fake}.P.B.c_B.\langle A, m\rangle,\ \mathit{receive}.B.c_B.P.\langle A, m\rangle\rangle\,.$$

In this trace pattern, we assume that A is honest.

3. The event $\mathit{hijack}.P' \rightarrow P.B' \rightarrow B.c_B.\langle A, m\rangle$ occurs in the trace. Since all proxies are honest, and all behave in the same way, the intruder does not gain anything by changing the identity of the sender of the message; we assume that $P' = P$. We apply $\mathcal{N}_2$ again: the proxy P must have sent the message to B' earlier in the trace (in some connection c_P). So far, the trace has the pattern:

$$s \mathrel{\widehat{=}} \langle \mathit{send}.P.c_P.B'.\langle A, m\rangle,\\ \mathit{hijack}.P.B' \rightarrow B.c_B.\langle A, m\rangle,\ \mathit{receive}.B.c_B.P.\langle A, m\rangle\rangle\,.$$

P is a multiplexing proxy, so $\mathit{Proxies}(R_i \to R_j)$ applies again, and implies that $\mathit{Proxy}(P)$ holds for tr, and so P must previously have received a message of the form $\langle m, B' \rangle$ from A (in some connection c'_P). The trace now has the following pattern:

$$s \mathrel{\widehat{=}} \langle \mathit{receive}.P.c'_P.A.\langle m, B' \rangle,\ \mathit{send}.P.c_P.B'.\langle A, m \rangle,\\ \mathit{hijack}.P.B' \to B.c_B.\langle A, m \rangle,\ \mathit{receive}.B.c_B.P.\langle A, m \rangle \rangle\,.$$

We apply network rule $\mathcal{N}_4$ again; this implies the existence of one of several events earlier in the trace.

(a) $\mathit{send}.A.c_A.P.\langle m, B' \rangle$ **in** tr for some connection c_A; the trace has the following pattern:

$$tr_5 \mathrel{\widehat{=}} \langle \mathit{send}.A.c_A.P.\langle m, B' \rangle,\\ \mathit{receive}.P.c'_P.A.\langle m, B' \rangle,\ \mathit{send}.P.c_P.B'.\langle A, m \rangle,\\ \mathit{hijack}.P.B' \to B.c_B.\langle A, m \rangle,\ \mathit{receive}.B.c_B.P.\langle A, m \rangle \rangle\,.$$

As before, we assume that A is honest and that $B' \neq B$.

(b) $\mathit{fake}.A.P.c'_P.\langle m, B' \rangle$ **in** tr. If the intruder can fake the message $\langle m, B' \rangle$, he can also fake the message $\langle m, B \rangle$; this means that the hijack event on the channel from the proxy is unnecessary. We ignore this trace pattern.

(c) $\exists A' : \hat{R}_i; P' : P\widehat{roxy} \cdot \mathit{hijack}.A' \to A.P' \to P.c'_P.\langle m, B' \rangle$ **in** tr. As before, we assume that $P' = P$. We apply $\mathcal{N}_2$ again: the agent A' must have sent the message to P (in some connection $c_{A'}$) in order for the intruder to hijack it. The trace now has the following pattern:

$$tr_5 \mathrel{\widehat{=}} \langle \mathit{send}.A'.c_{A'}.P.\langle m, B' \rangle,\ \mathit{hijack}.A' \to A.P.c'_P.\langle m, B' \rangle,\\ \mathit{receive}.P.c'_P.A.\langle m, B' \rangle,\ \mathit{send}.P.c_P.B'.\langle A, m \rangle,\\ \mathit{hijack}.P.B' \to B.c_B.\langle A, m \rangle,\ \mathit{receive}.B.c_B.P.\langle A, m \rangle \rangle\,.$$

We assume that A' is honest and that $A' \neq A$ and $B' \neq B$.

Applying the channel properties

The properties on the channels to and from the proxy are applied in exactly the same way as for the simple proxies. The result of this stage of the automated proof is a set of trace patterns $\{tr_1, tr_2, \ldots, tr_n\}$ that are allowed on the resultant channel.

Determining the resultant channel specification

We determine the channel property that the resultant channel satisfies in the same way as we do in the automated proof of the simple chaining theorem. The resultant channel is confidential if and only if the channels to and from the proxy are confidential; we call this value c. We determine the remaining three co-ordinate points in the lattice by looking at the allowed trace patterns.

Each of these trace patterns demonstrates that the intruder can perform a particular event. We present below a summary of these patterns; each case shows the initial and final events in the pattern, and then describes which activity this demonstrates. This mapping allows us to assign a tuple (nf, nra, nr) to each trace pattern; this tuple represents the strongest possible specification that this trace allows.

When the final event is $receive.B.c_B.P.\langle A, m\rangle$ (agent B receives a message from the proxy that appears to have been sent on the honest agent A's behalf):

send$.A.c_A.P.\langle m, B\rangle$ this does not demonstrate any activity: (NF, NRA, NR);

send$.A.c_A.P.\langle m, I\rangle$ the intruder redirected a message that was intended for a dishonest agent: (NF, NRA, NR^-);

send$.A.c_A.P.\langle m, B'\rangle$ the intruder redirected a message that was intended for an honest agent: $(NF, NRA, \bot)$;

send$.A'.c_{A'}.P.\langle m, B\rangle$ the intruder re-ascribed a message to an honest agent: $(NF, \bot, NR)$;

send$.A'.c_{A'}.P.\langle m, I\rangle$ the intruder re-ascribed a message to an honest agent, and redirected a message that was intended for a dishonest agent: $(NF, \bot, NR^-)$;

send$.A'.c_{A'}.P.\langle m, B'\rangle$ the intruder re-ascribed a message to an honest agent, and redirected a message that was intended for an honest agent: $(NF, \bot, \bot)$;

send$.I.c_I.P.\langle m, B\rangle$ the intruder hijacked his own message to simulate a fake: $(\bot, NRA, NR)$;

fake$.A.P.c_P.\langle m, B\rangle$ the intruder faked a message to the proxy: $(\bot, NRA, NR)$;

fake$.P.B.c_B.\langle A, m\rangle$ the intruder faked a message from the proxy: $(\bot, NRA, NR)$.

When the final event is $\mathit{receive}.B.c_B.P.\langle I, m\rangle$ (agent B receives a message from the proxy that appears to have been sent on the dishonest agent I's behalf):

$\boldsymbol{send}.I.c_I.P.\langle m, B\rangle$ this does not demonstrate any activity: (NF, NRA, NR);

$\boldsymbol{send}.A.c_A.P.\langle m, B\rangle$ the intruder re-ascribed a message to a dishonest agent: (NF, NRA^-, NR);

$\boldsymbol{send}.A.c_A.P.\langle m, I\rangle$ this does not demonstrate any activity (the intruder sent a message that was sent to him): (NF, NRA, NR);

$\boldsymbol{send}.A.c_A.P.\langle m, B'\rangle$ the intruder re-ascribed a message to a dishonest agent, and redirected a message that was intended for an honest agent; however, this is only possible on a non-confidential channel, so this simulates a learn and send: (NF, NRA, NR).

The result of applying the mapping to each of the trace patterns is a set of tuples of the form (nf_i, nra_i, nr_i) for $i = 1 \ldots n$. We take the value of the confidential co-ordinate and calculate the resultant channel specification in the following way (where **min** is calculated according to the order on the building blocks — see Chapter 4):

$$(c, nf, nra, nr) =\downarrow (c, \mathbf{min}_{i=1}^{n}(nf_i), \mathbf{min}_{i=1}^{n}(nra_i), \mathbf{min}_{i=1}^{n}(nr_i)) .$$

By eliminating trace patterns and calculating the resultant point in the hierarchy in this manner, we prove that the alternative specification of the resultant channel holds on all valid system traces in which the channel specifications to and from the proxy hold. The full list of results, and the Haskell script listing are shown in Appendix B.

5.3 Related work

In [MS94], the authors describe a calculus for secure channel establishment. They define channels that offer confidentiality ($\rightarrow\!\bullet$), authentication of the message sender ($\bullet\!\rightarrow$), or both ($\bullet\!\rightarrow\!\bullet$). The authors show that if user B trusts a third party T, and there are channels from another agent A such that $A \bullet\!\rightarrow T \bullet\!\rightarrow B$, then the agents A and B can establish a new channel $A \bullet\!\rightarrow B$. The authors also show that confidential channels can be chained, provided that the message sender trusts the third party. These two results agree with our chaining theorems; though our results go further as we show that many more channels can be chained. However, we cannot reason about channels when only one agent trusts the third party, as the authors of [MS94] can.

In [Boy93], Boyd defines two different types of channel: *Confidentiality*, where only the intended user (or set of users) can read the message; and

Authentication, where only the expected user (or set of users) can write the message. In Boyd's setup channels are established by utilising existing channels, or by propagating new channels between the two users wishing to communicate, often via a trusted third party (a proxy in our notation). Boyd shows that if a user A has an authenticated channel to a third party T, and T has an authenticated channel to a user B (and if B trusts T), then an authenticated channel from A to B can be established. This agrees with our (multiplexing) result that authenticated channels can be chained; as before though, our results go further as they show that many more channels can be chained.

Some authors have tried to solve the chaining problem by modifying the secure transport layer protocol. In [SBL06] the authors propose a variant of TLS in which three connections are established: a direct connection between client and server, and two direct connections between the client and a proxy, and between the proxy and the server. The direct connection can be used for highly confidential data, while the proxy channel can be used for data that does not have to remain secret. In [KCC01] the authors propose adding end-to-end encryption to chains of WTLS and TLS connections so that data sent via a proxy remains confidential. However, in both these cases, data can be passed through the proxy without the proxy being able to read it; the proxy can then no longer perform any application-layer jobs it might have (such as virus scanning).

5.4 Conclusions

In this chapter we investigated chaining our secure channel properties through a trusted third party (a proxy). We showed, in two different cases, that our channel properties are invariant under chaining, and that the overall channel property through a proxy is at least as strong as the greatest lower bound of the channels to and from the proxy. In some cases the overall channel is stronger than both channels.

In the case of simple proxies this elevation of the overall channel property was caused by the trust relations between the agents and the proxies: honest agents only send messages to their proxies, and all agents know which proxies are dedicated to sending messages to them. In the case of the multiplexing proxies this elevation is due to the extra information added to the application-layer messages.

In both cases we demonstrated a proof technique for showing that the channel through a proxy satisfies the properties described by the general chaining theorems, and we described a Haskell programme that automates each of the 121 proofs of correctness of the theorems. Finally, we compared our chaining results to similar results discovered and proved by other researchers.

Chapter 6

CSP models of secure channels

In Chapter 3 we defined a framework for specifying secure channel properties; we set out a hierarchy of 11 confidential and authenticated channel properties, and a hierarchy of 13 independent session and stream properties. One of the key reasons for developing these channel properties was to study security protocols that were designed with a particular secure transport layer in mind.

In this chapter we describe our abstract models of the channel properties. These abstract models simply capture the limitations that the channel properties impose on the intruder, without any of the complexity of modelling the secure transport layer protocol itself.

In Section 6.1 we describe and characterise (by its honest traces) the existing Casper model. We show that the standard model is equivalent to, i.e. has the same honest traces as, the bottom channel from Chapter 3. In Section 6.2 we describe the models of the new channels; these models are built around the existing Casper structure so that only minimal changes are required to the input scripts, and only small changes are made by the compiler to the resulting CSP output script. In this section we prove that these models are sound and complete (i.e. that they capture all honest traces of the relevant channel property and no more), so we conclude that they can usefully be used for protocol analysis.

In Section 6.3 we describe the alterations to the Casper model that are necessary to implement some of the session and stream properties, and we argue that these alterations correctly realise the properties. In Section 6.4, we describe the syntax for Casper input files to use the new channel models. We use the new Casper models for our case studies in Chapter 7. Finally, in Section 6.5 we conclude and summarise our findings.

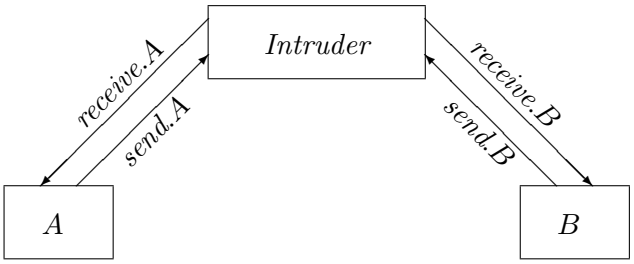


Figure 6.1: The standard Casper network model.

6.1 Casper model

Casper [Low98] is a compiler for the analysis of security protocols; Casper produces a CSP model of a security protocol from an abstract description of the protocol. The CSP model can be checked for security (trace) properties by the model checker FDR [FSE05]. In this section we describe the Casper model, and we formulate a description of the set of traces of the Casper system. We use this formulation to show that the standard Casper model is equivalent to the bottom channel specification.

A Casper input script describes a single application-layer protocol, and the CSP output script models a finite system of honest agents who run the protocol over a network controlled by an intruder. The intruder observes all messages entering the network, and controls all messages leaving the network, so the intruder controls all communication paths through the network; see Figure 6.1. The intruder can also take part in protocol runs using his own identity: the honest agents can send messages to him, and he can cause them to receive messages from him.

We recall the definition of an application-layer security protocol (from Chapter 4): an application-layer security protocol $\mathcal{P}$ is represented by a triple $(\mathcal{R}, \mathcal{M}, \mathcal{T})$. $\mathcal{R}$ is the set of roles in the protocol; $\mathcal{M}$ is the set of messages (labelled sequences of values taken from $Message_{App}$); $\mathcal{T}$ is the ordered sequence of message transmissions in the protocol.

We use the Yahalom protocol [BAN90] as a running example in this section to demonstrate how Casper models are constructed:

$$
\begin{array}{ll}
Message\ 1 & a \rightarrow b : a, na \\
Message\ 2 & b \rightarrow s : b, \{a, na, nb\}_{k_{bs}} \\
Message\ 3 & s \rightarrow a : \{b, k_{ab}, n_a, n_b\}_{k_{as}}, \{a, k_{ab}\}_{k_{bs}} \\
Message\ 4 & a \rightarrow b : \{a, k_{ab}\}_{k_{bs}}, \{n_b\}_{k_{ab}} .
\end{array}
$$

There are three roles in this protocol: the initiator $(a : R_1)$, the responder $(b : R_2)$ and the server $(s : R_3)$. Using the notation introduced above

this protocol is described as:

$$
\begin{aligned}
\mathcal{P} &= (\mathcal{R}, \mathcal{M}, \mathcal{T}) \,, \\
\mathcal{R} &= \{R_1, R_2, R_3\} \,, \\
\mathcal{M} &= \{(Msg_1, \langle a, na\rangle) \mid a : Agent, na : Nonce\} \cup \\
&\quad \{(Msg_2, \langle b, \{a, na, nb\}_{k_{bs}}\rangle) \mid \\
&\qquad a, b : Agent; n_a, n_b : Nonce; k_{bs} : SymmetricKey\} \cup \\
&\quad \{(Msg_3, \langle \{b, k_{ab}, n_a, n_b\}_{k_{as}}, \{a, k_{ab}\}_{k_{bs}}\rangle) \mid \\
&\qquad a, b : Agent; n_a, n_b : Nonce; k_{ab}, k_{as}, k_{bs} : SymmetricKey\} \cup \\
&\quad \{(Msg_4, \langle \{a, k_{ab}\}_{k_{bs}}, \{n_b\}_{k_{ab}}\rangle) \mid \\
&\qquad a : Agent; n_b : Nonce; k_{ab}, k_{bs} : SymmetricKey\} \,, \\
\mathcal{T} &= \langle (R_1, R_2, Msg_1), (R_2, R_3, Msg_2), (R_3, R_1, Msg_3), (R_1, R_2, Msg_4)\rangle \,.
\end{aligned}
$$

For each of these roles Casper builds a process that sends and receives the messages of the protocol; each process is parameterised by the facts that are necessary to perform the particular role in the protocol. For example, the initiator process is as follows:

$$
\begin{aligned}
&INITIATOR(a, n_a, s) = \\
&\quad \Box\, b : Responder \bullet env.a.(Env_0, b) \rightarrow \\
&\quad send.a.b.(Msg_1, \langle a, n_a\rangle) \rightarrow \\
&\quad \Box\, s : Server \Box\, k_{ab} : Key \bullet \\
&\qquad receive.s.a.(Msg_3, \langle \{b, k_{ab}, n_a, n_b\}_{k_{as}}, m\rangle) \rightarrow \\
&\quad send.a.b.(Msg_4, \langle m, \{n_b\}_{k_{ab}}\rangle) \rightarrow STOP \,.
\end{aligned}
$$

The initiator process is parameterised by the identity of the agent playing the role (a), the values of all facts necessary to play the role (n_a), and the identities of third party agents that are fixed by the protocol structure (in this case, just the server's identity s is fixed).

We note that the values of facts in received messages (such as nonces) are checked automatically because the process does not accept a message in which the values are incorrect. The second component of the third message is just treated as a value m from $Message_{App}$; this field in the protocol is encrypted with b's shared key with s, so a cannot check the values in it.

As well as performing a send or receive event for every protocol message that the initiator takes part in, the initiator process receives an initial event from the environment; this event tells the initiator which agent to run the protocol with. Whether values of facts and identities of other agents are passed to the process as parameters or in an initial message from the environment is controlled by the Casper input script.

Casper builds a process like the one above for every role in the protocol. Each of these processes can be initiated with an agent's identity (and the values of the facts that process needs to know to run the protocol). For any protocol $\mathcal{P}$ we write $\mathcal{P}_i(A)$ for the process for role R_i initiated with agent A's identity. $traces(\mathcal{P}_i(A))$ is the complete set of traces that process

could perform, and $traces(\mathcal{P}_A)$ is the set of traces that all the processes initiated with agent A's identity perform (these processes are interleaved, so this is the set of interleavings of $traces(\mathcal{P}_i(A))$ for all the appropriate i).

The set of traces of the honest agents in the Casper system is therefore described by the following set:

$$\{tr \in \{|\, send.Honest, receive.Honest \,|\}^* \mid \\ \forall A : Honest \cdot tr \upharpoonright \{|\, send.A, receive.A \,|\}^* \in traces(\mathcal{P}_A)\}\,.$$

Definition 6.1.1 (Protocol traces)**.** We write $traces(\mathcal{P})$ for the set of all possible traces of the application-layer protocol $\mathcal{P}$:

$$traces(\mathcal{P}) \mathrel{\widehat{=}} \\ \{tr \in \{|\, send, receive \,|\}^* \mid \\ \forall A : Agent \cdot tr \upharpoonright \{|\, send.A, receive.A \,|\} \in traces(\mathcal{P}_A)\}\,.$$

The standard Casper agent processes do not communicate in explicit connections: they just send and receive messages to and from other agents. In Section 6.3 we build explicit connection identifiers into Casper to model the session and stream channels. Until then, we assume that the agents do communicate in connections. Connection identifiers can easily be mapped onto the traces performed by the agents in the following way:

1. Connection identifiers are added to the events in the same way as they appear in the events in Chapter 3;

2. Every time an agent sends a message to another agent for the first time they start a new connection, and so a new connection identifier is introduced;

3. Every time an agent receives a message from another agent for the first time they start a new connection, and so a new connection identifier is introduced.

Casper also builds an intruder process that controls the communication between the honest agent processes. The intruder synchronises on all of the send events performed by the honest agents, and learns the values of the messages. After every send event that he overhears the intruder uses the deduction rules described in Chapter 2 to close the set of his previous knowledge and the new message under the deduction relation. At any time the intruder can say a value that he has learned to cause an honest agent to receive a message from any other agent's identity.

This intruder process can act in an entirely passive manner, just allowing message transmissions to be received unaltered, or he can block, fake, redirect, re-ascribe and alter messages before passing them on. For more details on the intruder process and the deduction relation see [RG97, Low98, RSG+01].

As in the formal model described in Chapter 3, the intruder in the Casper model has a set of initial knowledge *IIK* which he can use in his deductions. In the standard Casper model, the intruder learns every sent message (i.e. no channels are confidential), so after any trace the intruder knows everything that he can deduce from his initial knowledge and what he has overheard:

$$\begin{aligned}&\mathit{CasperIntruderKnows}_{IIK}(tr) \mathrel{\widehat{=}} \\ &\quad \{m \mid \mathit{IIK} \cup tr \downarrow \{\!|\, \mathit{send.Honest.Connection.Agent} \,|\!\} \vdash m\}\,.\end{aligned}$$

In the Casper models the intruder does not send messages with his own identity (he just causes honest agents to receive them from him), nor does he receive messages that were sent to him (he learns the content of the messages from overhearing the send events). The complete set of traces of a Casper system is therefore described by the following set:

$$\begin{aligned}&\mathit{CasperTraces}_{IIK} \mathrel{\widehat{=}} \\ &\quad \{tr \in \{\!|\, \mathit{send.Honest}, \mathit{receive.Honest} \,|\!\}^* \mid \\ &\qquad \forall A : \mathit{Honest} \cdot tr \upharpoonright \{\!|\, \mathit{send.A}, \mathit{receive.A} \,|\!\} \in \mathit{traces}(\mathcal{P}_A)\} \wedge \\ &\qquad \forall B : \mathit{Honest}; c_B : \mathit{Connection}; A : \mathit{Agent}; m : \mathit{Message}_{App}; tr' : \mathit{Trace} \cdot \\ &\qquad\quad tr' \frown \langle \mathit{receive}.B.c_B.A.m \rangle \leqslant tr \Rightarrow \\ &\qquad\qquad m \in \mathit{CasperIntruderKnows}_{IIK}(tr')\}\,.\end{aligned}$$

The above description of the traces of the Casper system is based on the description of Casper traces from [BL03].

In order to prove that the models we have developed for Casper are sound and complete we show that the set of traces of the specification system defined in Chapter 3 is equal to the set of traces of the Casper system, subject to the limitations that the application protocol imposes on the honest agents' behaviour.

In Chapter 3 we defined the set of $\mathit{ValidSystemTraces}_{IIK}$; this is the set of traces composed of application-layer send and receive events and transport-layer send, receive, fake and hijack events that follow the rules $\mathcal{N}_1$–$\mathcal{N}_4$, $\mathcal{A}_1$ and $\mathcal{A}_2$. In this chapter we use the simulation relation to show that the Casper system of any channel specification is equivalent to the system described in Chapter 3.

In order to simplify the job of showing the simulation relations hold we formulate below a precise definition of $\mathit{HonestTraces}_{IIK}$: the projection of the set of valid traces onto the events performed by the honest agents. The network rules $\mathcal{N}_1$, $\mathcal{N}_2$ and $\mathcal{N}_3$ restrict the intruder's behaviour, and they all hold vacuously on an honest trace. Similarly, rules $\mathcal{A}_1$ and $\mathcal{A}_2$ relate application-layer events to transport-layer events, and since transport-layer events do not appear in honest traces these rules hold vacuously.

We restate a slightly modified form of rule $\mathcal{N}_4$ below; this rule does not necessarily hold for every honest trace, so we must consider each of the

consequences of the implication separately:

$$\begin{array}{l}\mathcal{N}_4(tr) \mathrel{\widehat{=}} \\ \quad \forall B : \mathit{Honest}; c_B : \mathit{Connection}; A : \mathit{Agent}; m : \mathit{Message}_{\mathit{App}}; tr' : \mathit{Trace} \cdot \\ \quad\quad tr'^\frown\langle \mathit{receive}.B.c_B.A.m\rangle \leqslant tr \Rightarrow \exists A', B' : \mathit{Agent}; c_A : \mathit{Connection} \cdot \\ \quad\quad\quad \mathit{send}.A.c_A.B.m \textbf{ in } tr' \vee \\ \quad\quad\quad \mathit{fake}.A.B.c_B.m \textbf{ in } tr' \vee \\ \quad\quad\quad \mathit{hijack}.A' \rightarrow A.B' \rightarrow B.c_B.m \textbf{ in } tr' \,.\end{array}$$

1. If a $\mathit{send}.A.c_A.B.m$ event precedes the receive event in the trace then either agent A is honest (in which case the event appears in the honest trace), or agent A is dishonest (in which case the event does not appear in the honest trace). In this second case, we use rule $\mathcal{N}_3$ to conclude that $m \in \mathit{IntruderKnows}_{IIK}(tr')$.

2. If a $\mathit{fake}.A.B.c_B.m$ event precedes the receive event in the trace then this event does not appear in the honest trace; as above, we conclude that $m \in \mathit{IntruderKnows}_{IIK}(tr')$.

3. If a $\mathit{hijack}.A' \rightarrow A.B' \rightarrow B.c_B.m$ event precedes the receive event in the trace then there must previously have been a $\mathit{send}.A'.c_{A'}.B'.m$ event in the trace, for some connection $c_{A'}$ (by rule $\mathcal{N}_2$). In this case we assume that A' is honest, because we can disregard those traces in which the intruder hijacks his own messages (Proposition 4.4.1).

Hence we define:

$$\begin{array}{l}\mathit{HonestTraces}_{IIK} \mathrel{\widehat{=}} \\ \quad \{tr \in \{|\mathit{send}.\mathit{Honest}, \mathit{receive}.\mathit{Honest}|\}^* \mid \\ \quad\quad \forall B : \mathit{Honest}; c_B : \mathit{Connection}; A : \mathit{Agent}; m : \mathit{Message}_{\mathit{App}}; tr' : \mathit{Trace} \cdot \\ \quad\quad\quad tr'^\frown\langle \mathit{receive}.B.c_B.A.m\rangle \leqslant tr \Rightarrow \\ \quad\quad\quad\quad \exists A', B' : \mathit{Agent}; c_{A'} : \mathit{Connection} \cdot \mathit{send}.A'.c_{A'}.B'.m \textbf{ in } tr' \vee \\ \quad\quad\quad\quad m \in \mathit{IntruderKnows}_{IIK}(tr')\} \,.\end{array}$$

Theorem 6.1.2. *The standard* Casper *model is equivalent to the bottom channel specification $\perp$ with respect to any application layer protocol $\mathcal{P}$. In other words, if every channel in the formal model satisfies the specification $\perp$, then:*

$$\forall IIK \subseteq \mathit{Message}_{\mathit{App}} \cdot \mathit{HonestTraces}_{IIK} \cap \mathit{traces}(\mathcal{P}) = \mathit{CasperTraces}_{IIK} \,.$$

Proof. The theorem clearly holds once we observe that on the bottom channel (which is non-confidential):

$$\forall IIK \subseteq \mathit{Message}_{\mathit{App}} \cdot \mathit{send}.A'.c_{A'}.B'.m \textbf{ in } tr' \Rightarrow \mathit{IntruderKnows}_{IIK}(tr') \,,$$

and since

$$IntruderKnows_{IIK}(tr) \mathrel{\widehat=} \{m \mid (IIK \cup SentToIntruder(tr) \cup SentOnNonConfidential(tr)) \vdash m\},$$

and all channels are non-confidential:

$$\forall IIK \subseteq Message_{App}; tr : Trace \cdot IntruderKnows_{IIK}(tr) = CasperIntruderKnows_{IIK}(tr).$$

□

6.2 Authenticated and confidential channels

In the previous section we described the standard Casper model, and we proved that it is equivalent to the bottom channel specification. In this section we describe the changes to Casper to model the new channels, and we prove that these models are equivalent to the channel specifications. We prove this equivalence by showing that the sets of honest traces of the Casper system are equal to the sets of honest traces of the specification.

In the standard Casper model the intruder's knowledge increases when he overhears messages sent by honest agents: an event of the form $send.A.c_A.B.m$ for honest A is renamed to a $hear.m$ event. The new message m is then used to increase the set of the intruder's knowledge; i.e.

$$CasperIntruderKnows_{IIK}(tr^\frown\langle send.A.c_A.B.m\rangle) = \{m' \mid IIK \cup CasperIntruderKnows_{IIK}(tr) \cup \{m\} \vdash m'\}.$$

Every message in the intruder's knowledge is stored in terms of its components (i.e. the parts of the message that the intruder can break down and reconstruct); these are known as *facts*. The intruder can combine facts to create new messages, and, for any message m that he knows, the intruder can perform a $say.m$ event; this is renamed to an event of the form $receive.B.c_B.A.m$ for honest B.

To model the new channels we create several new facts that the intruder can learn; these facts record the message that was sent (as the old facts do) but they also record who the message was originally sent by and to whom it was sent. These new facts have different deduction rules (depending on whether or not the channel is confidential), and the renaming rules for the intruder to hear and say these new facts are restricted to reflect the properties of the channel.

The new facts are used on channels as follows:

SentToC.(B, m) for confidential channels that allow re-ascribing but specify NR or NR^-; note that there is not a non-confidential form of this fact because there are no non-confidential channels that allow re-ascribing;

SentBy.(A, m) for channels that allow redirecting but specify *NRA* or NRA^-; note that there is not a confidential form of this fact because it is only used on channels that allow redirecting (and hence cannot be confidential);

SentByTo.(A, B, m) for channels that specify *NRA* or NRA^- and *NR* or NR^-;

SentByToC.(A, B, m) for confidential channels that specify *NRA* or NRA^- and *NR* or NR^-.

The deduction rules for these facts are straightforward:

$$\begin{aligned}&\forall A, B : Agent; m : Message \cdot \\ &\quad SentBy.(A, m) \vdash m \wedge \\ &\quad SentByTo.(A, B, m) \vdash m\,.\end{aligned}$$

The intruder cannot use the confidential facts ($SentToC.(B, m)$ and $SentByToC.(A, B, m)$) to deduce any further facts.

6.2.1 Channel models

In this section we describe the renaming rules for the send/hear and say/receive events for the new channel models.[1] For each channel we describe the renaming rules, and we describe how these relate to the events that the intruder can and cannot perform.

$\perp$ The renaming relation is as follows:

$$\{(hear.m, send.A.B.m) \mid A : Honest; B : Agent; m : Message\} \cup$$

$$\{(say.m, receive.A.B.m) \mid A : Agent; B : Honest; m : Message\}\,.$$

The intruder overhears all messages sent on the network and can send messages with any agent's identity; this reflects the fact that the intruder can fake messages, and can hijack messages by creating a receive event with any pair of agent identities after hearing a message that was previously sent.

NF $\wedge$ ***NRA***$^-$ The renaming relation is as follows:

$$\begin{aligned}&\{(hear.SentBy.(A, m), send.A.B.m) \mid \\ &\quad A : Honest; B : Agent; m : Message\} \cup\end{aligned}$$

$$\{(say.m, receive.I.B.m) \mid I : Dishonest; B : Honest; m : Message\} \cup$$

$$\begin{aligned}&\{(say.SentBy.(A, m), receive.A.B.m) \mid \\ &\quad A, B : Honest; m : Message\}\,.\end{aligned}$$

[1] In this section we omit the connection identifiers from the honest agents' events; as before, these can easily be added to the traces later.

The intruder overhears all messages sent on the network, but he can only send messages with his own identity. The intruder can redirect all previously sent messages, and he can re-ascribe messages to dishonest identities by learning them and sending them.

$\boldsymbol{NF} \wedge \boldsymbol{NRA}^{-} \wedge \boldsymbol{NR}^{-}$ The renaming relation is as follows:

$$\{(\mathit{hear.SentByTo.}(A, B, m), \mathit{send.A.B.}m) \mid A : \mathit{Honest}; B : \mathit{Agent}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.}m, \mathit{receive.}I\mathit{.B.}m) \mid I : \mathit{Dishonest}; B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByTo.}(A, B, m), \mathit{receive.A.B.}m), A, B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByTo.}(A, I, m), \mathit{receive.A.B.}m) \mid A, B : \mathit{Honest}; I : \mathit{Dishonest}; m : \mathit{Message}\}.$$

The intruder overhears all messages sent on the network, but he can only send messages with his own identity. The intruder can re-ascribe previously sent messages to a dishonest identity, but he can only redirect messages that were sent to him. The intruder can only re-ascribe such messages to himself, so to re-ascribe *and* redirect a message he learns the message and sends it with his own identity.

$\boldsymbol{NF} \wedge \boldsymbol{NRA}^{-} \wedge \boldsymbol{NR}$ The renaming relation is as follows:

$$\{(\mathit{hear.SentByTo.}(A, B, m), \mathit{send.A.B.}m) \mid A : \mathit{Honest}; B : \mathit{Agent}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.}m, \mathit{receive.}I\mathit{.B.}m) \mid I : \mathit{Dishonest}; B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByTo.}(A, B, m), \mathit{receive.A.B.}m), A, B : \mathit{Honest}; m : \mathit{Message}\}.$$

The intruder overhears all messages sent on the network, but he can only send messages with his own identity. The intruder can re-ascribe previously sent messages to a dishonest identity, but he cannot redirect messages.

$\boldsymbol{C} \wedge \boldsymbol{NR}^-$ The renaming relation is as follows:

$$\{(hear.SentToC.(B, m), send.A.B.m) \mid A, B : Honest; m : Message\} \cup$$

$$\{(hear.m, send.A.I.m) \mid A : Honest; I : Dishonest; m : Message\} \cup$$

$$\{(say.m, receive.A.B.m) \mid A : Agent; B : Honest; m : Message\} \cup$$

$$\{(say.SentToC.(B, m), receive.A.B.m) \mid A : Agent; B : Honest; m : Message\} .$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, fake messages, and he can redirect messages that were sent to him by faking them. The intruder cannot redirect messages that were sent to honest agents (but he can re-ascribe them).

$\boldsymbol{C} \wedge \boldsymbol{NRA}^- \wedge \boldsymbol{NR}^-$ The renaming relation is as follows:

$$\{(hear.SentByToC.(A, B, m), send.A.B.m) \mid A, B : Honest; m : Message\} \cup$$

$$\{(hear.m, send.A.I.m) \mid A : Honest; I : Dishonest; m : Message\} \cup$$

$$\{(say.m, receive.A.B.m) \mid A : Agent; B : Honest; m : Message\} \cup$$

$$\{(say.SentByToC.(A, B, m), receive.A.B.m) , (say.SentByToC.(A, B, m), receive.I.B.m) \mid A, B : Honest; I : Dishonest; m : Message\} .$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, fake messages, and he can redirect messages that were sent to him by faking them. The intruder can only re-ascribe messages that were sent to honest agents with his own identity, and he cannot redirect them.

$\boldsymbol{C} \wedge \boldsymbol{NRA} \wedge \boldsymbol{NR}^-$ The renaming relation is as follows:

$$\{(hear.SentByToC.(A, B, m), send.A.B.m) \mid A, B : Honest; m : Message\} \cup$$

$$\{(hear.m, send.A.I.m) \mid A : Honest; I : Dishonest; m : Message\} \cup$$

$$\{(say.m, receive.A.B.m) \mid A : Agent; B : Honest; m : Message\} \cup$$

$$\{(say.SentByToC.(A, B, m), receive.A.B.m) \mid A, B : Honest; m : Message\} .$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, fake messages, and he can redirect messages that were sent to him by faking them. The intruder cannot re-ascribe or redirect messages that were sent to honest agents, he can only replay them.

$\boldsymbol{C} \wedge \boldsymbol{NF} \wedge \boldsymbol{NRA}^{-} \wedge \boldsymbol{NR}^{-}$ The renaming relation is as follows:

$$\{(\mathit{hear.SentByToC.}(A, B, m), \mathit{send.A.B.}m) \mid A, B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{hear.SentByTo.}(A, I, m), \mathit{send.A.I.}m) \mid A : \mathit{Honest}; I : \mathit{Dishonest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.}m, \mathit{receive.I.B.}m) \mid I : \mathit{Dishonest}; B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByToC.}(A, B, m), \mathit{receive.A.B.}m), (\mathit{say.SentByToC.}(A, B, m), \mathit{receive.I.B.}m) \mid A, B : \mathit{Honest}; I : \mathit{Dishonest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByTo.}(A, I, m), \mathit{receive.A.B.}m) \mid A, B : \mathit{Honest}; I : \mathit{Dishonest}; m : \mathit{Message}\}.$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, but he cannot fake messages. The intruder can only re-ascribe messages with his own identity, and he can only redirect messages that were sent to him. To re-ascribe *and* redirect a message he learns the message and sends it with his own identity.

$\boldsymbol{C} \wedge \boldsymbol{NF} \wedge \boldsymbol{NRA} \wedge \boldsymbol{NR}^{-}$ The renaming relation is as follows:

$$\{(\mathit{hear.SentByToC.}(A, B, m), \mathit{send.A.B.}m) \mid A, B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{hear.SentByTo.}(A, I, m), \mathit{send.A.I.}m) \mid A : \mathit{Honest}; I : \mathit{Dishonest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.}m, \mathit{receive.I.B.}m) \mid I : \mathit{Dishonest}; B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByToC.}(A, B, m), \mathit{receive.A.B.}m) \mid A, B : \mathit{Honest}; m : \mathit{Message}\} \cup$$

$$\{(\mathit{say.SentByTo.}(A, I, m), \mathit{receive.A.B.}m) \mid A, B : \mathit{Honest}; I : \mathit{Dishonest}; m : \mathit{Message}\}.$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, but he cannot fake messages. The intruder

cannot re-ascribe messages, and he can only redirect messages that were sent to him.

$\boldsymbol{C \wedge NF \wedge NRA^{-} \wedge NR}$ The renaming relation is as follows:

$$\{(hear.SentByToC.(A, B, m), send.A.B.m) \mid A, B : Honest; m : Message\} \cup$$

$$\{(hear.m, send.A.I.m) \mid A : Honest; I : Dishonest; m : Message\} \cup$$

$$\{(say.m, receive.I.B.m) \mid I : Dishonest; B : Honest; m : Message\} \cup$$

$$\{(say.SentByToC.(A, B, m), receive.A.B.m), (say.SentByToC.(A, B, m), receive.I.B.m) \mid A, B : Honest; I : Dishonest; m : Message\}.$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, but he cannot fake messages. The intruder can only re-ascribe messages with his own identity, and he cannot redirect messages.

$\boldsymbol{C \wedge NF \wedge NRA \wedge NR}$ The renaming relation is as follows:

$$\{(hear.SentByToC.(A, B, m), send.A.B.m) \mid A, B : Honest; m : Message\} \cup$$

$$\{(hear.m, send.A.I.m) \mid A : Honest; I : Dishonest; m : Message\} \cup$$

$$\{(say.m, receive.I.B.m) \mid I : Dishonest; B : Honest; m : Message\} \cup$$

$$\{(say.SentByToC.(A, B, m), receive.A.B.m) \mid A, B : Honest; I : Dishonest; m : Message\}.$$

The intruder cannot learn messages sent to honest agents. He can send messages with his own identity, but he cannot fake, re-ascribe or redirect messages.

6.2.2 Soundness and completeness

In Chapter 3 we assume that the intruder cannot use the hijack event to play a message from one channel on another one: the hijack event does not allow the intruder to change the roles of the agents, only their identities. In order to prove the equivalence of the Casper models to the formal channel specifications we must assume a property of the application-layer protocols that prevents the intruder from hijacking messages from one channel to

another. If we do not assume that application-layer protocols have this property then the proofs in this section become soundness proofs, rather than equivalence proofs. We recall the definition of *disjoint messages* from Chapter 4.

Definition 6.2.1 (Disjoint messages)**.** An application-layer protocol $\mathcal{P} = (\mathcal{R}, \mathcal{M}, \mathcal{T})$ has disjoint messages if the sets of possible values of encrypted components of different messages are disjoint:

$$\begin{array}{l}\forall (Msg_i, M_i), (Msg_j, M_j) \in \mathcal{M} \cdot \\ \quad \forall m_i \in EncryptedComponents(M_i); m_j \in EncryptedComponents(M_j) \cdot \\ \quad\quad m_i = m_j \Rightarrow i = j\,.\end{array}$$

The disjoint messages property of an application-layer protocol ensures that even with the bottom secure channel property, the intruder cannot take messages from one channel and play them on the other. This is in fact a stronger disjointness property than we need to show the equivalence of the single-message channel properties, but we use this strong property in Section 6.3 for the session and stream properties.[2]

Confidentiality The only difference between a confidential and a non-confidential channel in Casper is whether or not the intruder can learn the application-layer message from the fact that he learns from hearing the message on the transport layer. By inspection of the renaming relations and the deduction relations described earlier in this section we see that the intruder can only learn messages sent on non-confidential channels, or sent to him. In particular, it is clear that after any Casper trace tr on a mixture of confidential and non-confidential channels, the intruder's knowledge is described by $IntruderKnows_{IIK}(tr)$. In other words, the intruder's knowledge is exactly the same as it is in the formal model.

Authentication For each of the secure channels in the hierarchy we prove the following theorem:

Theorem 6.2.2 (Equivalence of channel models)**.** *The* Casper *model of the channel property P is equivalent to the formal model of the channel property. In other words, given a channel specification $P(R_i \rightarrow R_j)$ and an application-layer protocol $\mathcal{P} = (\mathcal{R}, \mathcal{M}, \mathcal{T})$:*

$$\begin{array}{l}\forall IIK \subseteq Message_{App} \cdot HonestTraces_{IIK}(P(R_i \rightarrow R_j)) \cap traces(\mathcal{P}) = \\ \quad CasperTraces_{IIK}(P(R_i \rightarrow R_j))\,,\end{array}$$

[2]For the equivalence of the single message channels it is sufficient that the messages sent between different roles on channels that satisfy the same property are disjoint; there may be intersections in the sets of messages sent by one role to another on the same channel.

where $HonestTraces_{IIK}(P(R_i \rightarrow R_j))$ *is the set of valid system traces that satisfy the property* P *on the channel* $R_i \rightarrow R_j$ *and the property* $\bot$ *on all other channels, and* $CasperTraces_{IIK}(P(R_i \rightarrow R_j))$ *is the set of traces of the* Casper *system with the model of specification* P *on the channel* $R_i \rightarrow R_j$ *and the standard* Casper *model on all other channels.*

In each system we consider the channel specification P applied to the channel $R_i \rightarrow R_j$, while every other channel satisfies the bottom specification. Because messages cannot be hijacked from one channel to another, and because including stronger channel specifications on other channels only restricts the traces on the other channels, the proof still holds when the other channels satisfy stronger properties.

In order to demonstrate the proof technique, we prove the theorem for the channel $C \wedge NF \wedge NRA^{-} \wedge NR^{-}$.

Proof. We first calculate the honest traces of the channel specification using $HonestTraces_{IIK}$ and the alternative specification of the channel $C \wedge NF \wedge NRA^{-} \wedge NR^{-}$. We recall the alternative specification:

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA^{-} \wedge NR^{-})(R_i \rightarrow R_j) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App}; tr' : Trace \cdot \\
\qquad tr' \frown \langle receive.B.c_B.A.m \rangle \leqslant tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr' \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \rightarrow A.B' \rightarrow B.c_B.m \textbf{ in } tr' \wedge \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B'))\,.
\end{array}
$$

1. If a $send.A.c_A.B.m$ event precedes the receive event in the trace then either agent A is honest (in which case the event appears in the honest trace), or agent A is dishonest (in which case the event does not appear in the honest trace). In the second case we conclude, as before, that $m \in IntruderKnows_{IIK}(tr')$;

2. If a $hijack.A' \rightarrow A.B' \rightarrow B.c_B.m$ event precedes the receive event in the trace then there are four possibilities:

   (a) $A = A' \wedge B = B'$: in this case A is honest (because the intruder does not hijack his own send events to replay them), so we conclude that there is a $send.A.c_A.B.m$ event in the trace;

   (b) $A \neq A' \wedge B = B'$: in this case A is dishonest; there must have been a $send.A'.c_{A'}.B.m$ event earlier in the trace;

   (c) $A = A' \wedge B \neq B'$: in this case B' is dishonest; there must have been a $send.A.c_A.B'.m$ event earlier in the trace;

   (d) $A \neq A' \wedge B \neq B'$: in this case both A and B' are dishonest; we can safely block the intruder from performing a hijack event where

he redirects a message that was sent to him and re-ascribes it to himself (Proposition 4.4.3), so we do not allow this trace in the honest traces of the specification.

The honest trace formulation of the alternative specification is thus:

$$
\begin{array}{l}
\mathit{Alt}(C \wedge \mathit{NF} \wedge \mathit{NRA}^- \wedge \mathit{NR}^-)(R_i \rightarrow R_j) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{App}; tr' : \mathit{Trace} \cdot \\
\qquad tr'^\frown\langle \mathit{receive}.B.c_B.A.m \rangle \leqslant tr \Rightarrow \\
\qquad\quad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \textbf{ in } tr' \vee \\
\qquad\quad \mathit{Dishonest}(A) \wedge m \in \mathit{IntruderKnows}_{IIK}(tr') \vee \\
\qquad\quad \mathit{Dishonest}(A) \wedge \exists A' : \hat{R}_i; c_{A'} : \mathit{Connection} \cdot \\
\qquad\qquad \mathit{send}.A'.c_{A'}.B.m \textbf{ in } tr \vee \\
\qquad\quad \exists c_A : \mathit{Connection}; B' : (\mathit{Dishonest}, R_j) \cdot \mathit{send}.A.c_A.B'.m \textbf{ in } tr\,.
\end{array}
$$

In this specification, the existence of a send event by an agent A implies that that agent is honest (because if it were dishonest the event would not appear in the honest trace).

In order to obtain a complete description of the honest traces of this specification we take the definition of $\mathit{HonestTraces}_{IIK}$ from before, and add the alternative specification (above) as a condition for the receive events on the channel $R_i \rightarrow R_j$:

$$
\begin{array}{l}
\mathit{HonestTraces}_{IIK}(P(R_i \rightarrow R_j)) \mathrel{\widehat{=}} \\
\quad \{tr \in \{\mid \mathit{send}.\mathit{Honest}, \mathit{receive}.\mathit{Honest} \mid\}^* \mid \\
\qquad \forall B : \mathit{Honest}; c_B : \mathit{Connection}; A : \mathit{Agent}; m : \mathit{Message}_{App}; tr' : \mathit{Trace} \cdot \\
\qquad\quad tr'^\frown\langle \mathit{receive}.B.c_B.A.m \rangle \leqslant tr \Rightarrow \\
\qquad\qquad \exists A', B' : \mathit{Agent}; c_{A'} : \mathit{Connection} \cdot \mathit{send}.A'.c_{A'}.B'.m \textbf{ in } tr' \vee \\
\qquad\qquad m \in \mathit{IntruderKnows}_{IIK}(tr') \wedge \\
\qquad \forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{App}; tr' : \mathit{Trace} \cdot \\
\qquad\quad tr'^\frown\langle \mathit{receive}.B.c_B.A.m \rangle \leqslant tr \Rightarrow \\
\qquad\qquad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \textbf{ in } tr' \vee \\
\qquad\qquad \mathit{Dishonest}(A) \wedge m \in \mathit{IntruderKnows}_{IIK}(tr') \vee \\
\qquad\qquad \mathit{Dishonest}(A) \wedge \exists A' : \hat{R}_i; c_{A'} : \mathit{Connection} \cdot \\
\qquad\qquad\quad \mathit{send}.A'.c_{A'}.B.m \textbf{ in } tr \vee \\
\qquad\qquad \exists c_A : \mathit{Connection}; B' : (\mathit{Dishonest}, R_j) \cdot \mathit{send}.A.c_A.B'.m \textbf{ in } tr\}\,.
\end{array}
$$

This is the set of all valid honest traces with the bottom channel specification on all channels, with the additional restrictions of the channel property $C \wedge \mathit{NF} \wedge \mathit{NRA}^- \wedge \mathit{NR}^-$ on the channel $R_i \rightarrow R_j$. In order to prove the theorem we must show that this set of honest traces is equal to the set of traces of the Casper system; to do this, we consider the full set of traces of the Casper system.

First we recall the description of the set of traces of the Casper system:

$$
\begin{array}{l}
\mathit{CasperTraces}_{IIK} \mathrel{\widehat{=}} \\
\quad \{tr \in \{\!|\, \mathit{send.Honest}, \mathit{receive.Honest} \,|\!\}^* \mid \\
\qquad \forall A : \mathit{Honest} \cdot tr \upharpoonright \{\!|\, \mathit{send.A}, \mathit{receive.A} \,|\!\}^* \in \mathit{traces}(P_A)\} \wedge \\
\qquad \forall B : \mathit{Honest}; c_B : \mathit{Connection}; A : \mathit{Agent}; m : \mathit{Message}; tr' : \mathit{Trace} \cdot \\
\qquad\quad tr' \frown \langle \mathit{receive.B}.c_B.\mathit{A}.m \rangle \leqslant tr \Rightarrow \\
\qquad\qquad m \in \mathit{CasperIntruderKnows}_{IIK}(tr')\} \,.
\end{array}
$$

On the channel $R_i \rightarrow R_j$ the model of the intruder differs from the standard model, so we can be more specific in the formulation of $\mathit{CasperTraces}_{IIK}$ on the channel $R_i \rightarrow R_j$. We use the intruder's renaming and the deduction relation rules for the new facts to discover which events must have happened earlier in a trace in which a $\mathit{receive.B}.c_B.\mathit{A}.m$ event occurs on the channel $R_i \rightarrow R_j$.

Suppose first that the honest agent $B : \hat{R}_i$ receives the message m in connection c_B, apparently from agent $A : \hat{R}_i$. For this to happen, the intruder must be able to perform the receive event. By tracing the renaming relation backwards we observe that the intruder must be able to perform one of the following events:

1. $\mathit{say}.m$: the intruder cannot fake on this channel, so A must be dishonest; in this case, $m \in \mathit{CasperIntruderKnows}_{IIK}(tr')$;

2. $\mathit{say.SentByToC.}(A, B, m)$: in this case A must be honest; this fact can only be learned when the intruder hears it, so the event $\mathit{send.A}.c_A.B.m$ must occur earlier in the trace for some connection c_A;

3. $\mathit{say.SentByToC.}(A', B, m)$: if $A \neq A'$ then A must be dishonest; this fact can only be learned when the intruder hears it, so the event $\mathit{send.A'}.c_{A'}.B.m$ must occur earlier in the trace for some connection $c_{A'}$; when $A = A'$ this is covered by the case above;

4. $\mathit{say.SentByTo.}(A, B', m)$: in this case B' must be dishonest; this fact can only be learned when the intruder hears it, so the event $\mathit{send.A}.c_A.B'.m$ must occur earlier in the trace for some connection c_A; again, when $B = B'$ this is covered by the case above.

We add these possible prior events to the description of $\mathit{CasperTraces}_{IIK}$

to form $CasperTraces_{IIK}(P(R_i \rightarrow R_j))$:

$$
\begin{array}{l}
CasperTraces_{IIK}(P(R_i \rightarrow R_j)) \mathrel{\widehat{=}} \\
\quad \{tr \in \{\!|\, send.Honest, receive.Honest \,|\!\}^* \mid \\
\qquad \forall A : Honest \cdot tr \upharpoonright \{\!|\, send.A, receive.A \,|\!\}^* \in traces(\mathcal{P}_A)\} \wedge \\
\qquad \forall B : Honest; c_B : Connection; A : Agent; m : Message; tr' : Trace \cdot \\
\qquad\quad tr'^\frown\langle receive.B.c_B.A.m\rangle \leqslant tr \Rightarrow \\
\qquad\qquad m \in CasperIntruderKnows_{IIK}(tr') \wedge \\
\qquad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message; tr' : Trace \cdot \\
\qquad\quad tr'^\frown\langle receive.B.c_B.A.m\rangle \leqslant tr \Rightarrow \\
\qquad\qquad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr' \vee \\
\qquad\qquad Dishonest(A) \wedge m \in CasperIntruderKnows_{IIK}(tr') \vee \\
\qquad\qquad Dishonest(A) \wedge \exists A' : \hat{R}_i; c_{A'} : Connection \cdot \\
\qquad\qquad\quad send.A'.c_{A'}.B.m \textbf{ in } tr' \vee \\
\qquad\qquad \exists c_A : Connection; B' : (Dishonest, R_j) \cdot send.A.c_A.B'.m \textbf{ in } tr'\} \,.
\end{array}
$$

As with the proof of equivalence for the bottom channel specification we observe that if a $send.A'.c_A.B'.m$ appears in the trace on any channel other than $R_i \rightarrow R_j$ then the intruder knows that message, and that the intruder's knowledge after any trace is the same in the formal and Casper models. It is clear that when we restrict the honest traces of the channel specification to the traces of the application-layer protocol $\mathcal{P}$ the two sets are equal, and hence the two models are equivalent. □

6.3 Session and stream channels

In Section 6.2 we described how implicit connection identifiers can be mapped onto the traces of the Casper agent processes: each instantiation of each agent receives a new connection identifier for every agent process it communicates with. Our model of the session channels works in exactly the same way, except that the connection identifiers are made explicit. In this section we describe the session and stream models, and we argue that they are sound and complete.

6.3.1 Session channel models

To model the session channels we add explicit connection identifiers to the new facts on the relevant channels, and to the communication events of the honest agent processes.

For connections where an agent acts as the message sender the agent gets a new connection identifier to use for the connection; this connection identifier is passed to the agent process as a parameter (just as the values such as nonces are passed as parameters). The agent uses the same connection identifier for every message he sends in the connection.

When an agent acts as the message receiver the connection identifier is chosen by external choice when he receives the first message in the connection (just as the values of new facts that originate at other agents are chosen). After receiving the first message in a connection the agent process only receives further messages if they have the same connection identifier.

To illustrate the technique, we update the *INITIATOR* process from Section 6.1 to show how the connection identifiers are used (in this case every message in the protocol is sent on a session channel).

$$
\begin{array}{l}
INITIATOR(a, c_a, n_a, s) = \\
\quad \Box\, b : Responder \bullet env.a.(Env_0, b) \rightarrow \\
\quad send.a.c_a.b.(Msg_1, \langle a, n_a \rangle) \rightarrow \\
\quad \Box\, s : Server \Box\, k_{ab} : Key \Box\, c_s : Connection \bullet \\
\qquad receive.s.a.c_s.(Msg_3, \langle \{b, k_{ab}, n_a, n_b\}_{k_{as}}, m \rangle) \rightarrow \\
\quad send.a.c_a.b.(Msg_4, \langle m, \{n_b\}_{k_{ab}} \rangle) \rightarrow STOP\,.
\end{array}
$$

The connection identifiers are added to the facts in the natural way, so the new facts now have the following forms: $SentToC.(B, c_A, m)$, $SentBy.(A, c_A, m)$, $SentByTo.(A, c_A, B, m)$ and $SentByToC.(A, c_A, B, m)$.[3] The deduction rules are exactly the same as before, so they do not allow the intruder to change the connection identifier associated with the message in the new facts: the intruder cannot hijack messages from one connection to another.

Each honest agent needs a new connection identifier for every connection they establish, so the Casper compiler can calculate how many connection identifiers are necessary for any particular instantiation of honest agent processes. The intruder also has a connection identifier to use when he sends or fakes messages, but the intruder only uses one connection identifier for every message he sends or fakes.

Because agents only send (and receive) messages in the same connection with the same connection identifier, this mechanism correctly implements the session property. In order to show this we consider the receives-from relation $\mathcal{R}$ established by the traces of the Casper system and we show that it is left-total: i.e. every agent's connection receives messages from a single connection (or from the intruder). This is trivially true once we observe that each agent sends all their messages with the same connection identifier, the intruder can only fake messages with his connection identifier, the intruder cannot change the connection identifier in the new facts, and each honest agent only receives messages in a single connection if they all have the same connection identifier.

If an honest agent receives a single message in a session then that message must either have been faked to him by the intruder, or hijacked to him

[3]Note that the connection identifier in the *SentToC* fact is the connection identifier chosen by the message sender, but this does not affect whom the intruder re-ascribes the message to.

(by the intruder saying one of the new facts). In this case, there is another connection (either the original sender's connection or the intruder's connection identifier) such that the *receives-from* relation holds. If an honest agent receives more than one message in a single connection, but there does not exist a single connection (either honest or faked) that sent all the messages that agent received, then the intruder must have been able to fake with an honest agent's connection identifier (which he cannot do), or change the connection identifier associated with one of the facts he has learned (which he cannot do).

To model an injective session channel we need to restrict the honest agents' ability to start receiving messages from a connection. In the session channel model the honest agents perform an external choice over all possible connection identifiers before they receive the first message in a session. Once an honest agent has started to receive messages from a particular (honest) connection identifier, we must prevent other honest agents from receiving the messages associated with that connection identifier, and we must also prevent the same agent from receiving the messages in that connection again.

We construct a connection identifier manager process that synchronises with the honest agents on the first receive event in any new connection. Once an instantiation of an honest agent process has accepted an honest connection identifier no other agent process can accept that connection identifier. This manager process prevents the intruder from replaying sessions, so each session can be received at most once; this ensures the injectivity of the *receives-from* relation. The manager process only synchronises on the events with honest agents' connection identifiers.

For the models of session channels the honest agents use a different connection identifier for each channel. For example, suppose a protocol involves just two roles (R_i and R_j) communicating in a single session. In the model we have described the agent playing role R_i has a connection identifier for the messages that he sends to the agent playing role R_j, and he accepts messages from the agent playing role R_j that are associated with a different connection identifier. This means that the two agents in a session can have different views of the connection identifiers they are each using, so they may not actually be communicating in a symmetric session.

To model a symmetric session channel we change the models of the honest agents so that they perform and synchronise on a $pair.c_A.c_B$ event during each session they establish. This event can only be performed by the honest agents (except that any connection identifier can always be paired with the intruder's connection identifier), so once one agent has a complete view of both connection identifiers the intruder can only connect that agent's channels to another agent who has the same view. The pair event is performed according to the following schedule:

- The initiator performs the pair event after he sends his first message,

but before he receives the next message;

- The recipient performs the pair event immediately after receiving the first message.

Once the recipient has received the first message he has a complete view of the two connection identifiers because he knows what his connection identifier is for the next message he sends. However, the recipient cannot send this message until the agent who sent the first message performs the pair event. Because the intruder cannot change the connection identifier associated with a message, both agents 'own' one of the connection identifiers in the pair event, and so they are the only two agents who can perform the event. This ensures that a symmetric session is established, even if the agents cannot be sure of each other's identity.

6.3.2 Stream channel models

In Chaper 4 we showed that for many of the application-layer protocols that we are interested in (those with the *no speaking out of turn* property) it is safe to use a session channel rather than a stream channel because progress in the protocol is controlled by all the agents taking part. In particular, the order in which the messages between any two agents arrive is fixed by the protocol because the agents cannot make progress (send messages) until they receive the messages they expect to.

Any protocol that satisfies the *no speaking out of turn* property may be analysed in Casper with stream channels modelled simply by adding message numbers to the new facts that the intruder learns, then using the session channel models. The message numbers prevent the intruder from changing the order in which messages are received. However, for any protocol that satisfies the disjoint messages property, described in Section 6.2, the message numbers are not necessary because the intruder cannot play one message as another.

The mutual and synchronised stream properties from the end of Chapter 3 are often enforced by the application-layer protocol being studied, so building models of the stronger stream channels is not necessary.

6.4 Changes to Casper input scripts

In this chapter we described the Casper models of the secure channel specifications from Chapter 3. In this final section we describe the necessary changes to a Casper input script to use these new models in a standard Casper analysis.

The user encodes the protocol under consideration as usual; the only necessary addition to the input script is a `#Channels` section. This section

can be used to specify the old (secret, authenticated and direct) channels of [BL03], or to specify the use of the new channel models. The new models and the old models cannot be used together in the same script.

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

```
#Channels

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

The channel properties must be listed for each message individually, and must be listed in the order `C`, `NF`, `NRA(-)`, `NR(-)`, with the message number and the channel properties separated by any combination of white space characters.

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

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

The messages in the session (or stream) should be listed in order. However, the session and stream channel properties can be listed above or below the individual channel properties, and different messages sent by the same role in a single session do not have to satisfy the same channel property.

6.5 Conclusions

In this chapter we described abstract CSP models of our channel properties; these models capture the properties of the channels (such as no-redirecting) rather than modelling concrete transport layer protocols. We have built these models into Casper. We proved that these CSP models are equivalent to the formal channel properties described in Chapter 3, even though the model of the intruder is slightly different. We proved this soundness and completeness result using the equivalence relation defined in Chapter 4.

Finally, we described the simple changes that need to be made to a Casper input script in order to use these new channel models.

Chapter 7

Case Studies

One of the reasons for developing the secure channel specifications presented in this thesis, and the primary motivation behind the development of the models described in Chapter 6, is to study security protocols that rely on secure transport layers. In Chapter 1 we suggested that many protocols of this type can be found in web-based applications. In this chapter we study two web-based single sign on protocols that use secure channels (either SSL or TLS) as well as more traditional security techniques (such as MACs and signatures).

A typical computer user has several relationships with service providers, and must be able to authenticate himself to these service providers in order to request resources from them, or to make use of services that they offer. Most service providers want to be in full control of the authentication of their users, and so the average user has to remember many usernames and passwords for use at different service providers. A study of over half a million internet users conducted over three months in 2007 found that, on average, each user had 25 username accounts at different service providers, and typed 8 of these per day. Yet each user only had, on average, 6.5 different passwords [FH07].

In order to make things easier for themselves, many users record their passwords somewhere close to hand, or use the same password (and, where possible, the same username) at every service provider. This is not an ideal solution to the problem: secure authentication of a user at one service provider now depends on the security of other service providers. Typically, different service providers do not have the same requirements for the authentication of their users, and so, for example, a user's internet banking account may easily be hacked if the user chooses the same login details at a less secure service provider, such as an internet forum.

In Section 7.1 we describe the single sign-on paradigm. Single sign-on systems have been developed to ease the burden of memory on the users of computer systems without compromising the security of the systems.

In Section 7.2 we describe our use of the secure channel specifications of Chapter 3 to model SSL and TLS connections.

In the next two sections we report on our analysis of two single sign-on systems that have been designed specifically for web-based deployment. The SAML Web Browser Single Sign-On Profile was developed by OASIS [OAS05b] to authenticate users to service providers by means of trusted, centrally managed identity providers. In contrast to this, the OpenID Authentication [FRH+08] scheme uses decentralised, potentially untrusted OpenID providers to authenticate users to relying parties. It is up to each user and each relying party to decide whether or not they trust a particular OpenID provider. In Section 7.3 we describe the SAML protocols, and in Section 7.4 we describe the OpenID protocols.

We have found attacks against both sets of protocols when the recommendations in their specifications are not followed exactly. In Section 7.5 we describe a new single sign-on protocol that we have designed to be as concise as possible. Our protocol is correct when run over SSL and TLS connections, and relies heavily on our understanding of the properties of these channels to function correctly and securely. We give the exact requirements for the channels in Section 7.5. The short protocol development and description process emphasises the benefits of the channel models for protocol development.

Finally in Section 7.6 we conclude and summarise our findings.

7.1 Single Sign-On

In a single sign-on system a user who has a login session with one server and wishes to access a resource from another server can be authenticated to the new server by means of their existing authenticated session. The first server is often known as an identity provider; the second as a service provider or resource provider. The identity provider asserts to the service provider that the user is known to him, and has been authenticated. If the service provider trusts the identity provider then they trust that the authentication is valid, and so they create a new session for the user. The user does not have to re-authenticate himself to the service provider.

There are many situations when this form of identity federation can prove useful; for example:

- A company's portal software and the order systems of suppliers to that company. An employee of the company can sign in to the portal, and then place orders with the suppliers without needing to authenticate himself to each one individually.

- A web-based directory service where a user can register and provide their information. When they wish to access a resource at a third

party website they can be authenticated by their existing session with the directory service, and their details transferred automatically.

- An academic institution where teachers access several online systems (e.g. email, class attendance, student reports, applications). Each server can act as both identity provider and service provider so that once a teacher has signed on to one of the systems, they do not need to sign on to the others.

Single sign-on offers a considerable advantage to the users of a system: they only have to authenticate themselves to one server, and so they only have to remember one password. If a user requests resources from multiple service providers within the lifetime of a session with an identity provider, he does not have to authenticate himself to each service provider individually.

One way to achieve this sort of identity federation would be to give each user the same credentials and shared secrets with all of the service providers. However, this would allow any service provider to impersonate any user to any other service provider, and so, in this scenario, the user must trust many more servers than he has to with a single sign-on solution, where he only has to trust the identity provider.

The SAML and OpenID single sign-on protocols both specify means of protecting the application-layer messages by adding security constructs (such as signatures and MACs) to the protocol messages. However, they also allow security properties from the transport layer to be used to strengthen, or even to replace the application-layer message protection. Specifically, if users can establish unilateral TLS connections to authenticated servers, and the servers can initiate and respond to TLS connections with unilateral or bilateral authentication, then the confidentiality and authentication properties of TLS can be lifted to the application-layer messages. Rather than model these secure channels explicitly, we use the models from Chapters 3 and 6 to describe the properties we expect the network to satisfy, and then limit the abilities of the intruder to respect these properties.

7.2 Modelling TLS

In Chapter 2 we described the TLS protocol, and in Chapter 3 we described which of the secure channel properties we believe TLS (and SSL) satisfy. In this section we relate the properties of SSL and TLS to the secure channel properties, and we describe how we use the models from Chapter 6 to model SSL and TLS in Casper.

Many researchers have analysed SSL and TLS and have found that they allow agents to establish authenticated sessions, and to transfer data securely and confidentially; see e.g. [WS96, MSS98, Pau99, KL08]. SSL allows an unauthenticated client to establish a secure session with an authenticated

server: no guarantees about the client's identity are provided, but the client is assured of the server's identity. TLS can be run in two modes:

- In the unilateral authentication mode the server is authenticated to the client, but the client is not authenticated to the server; this mode is similar to SSL;
- In the bilateral authentication mode both client and server are authenticated to one another.

Kamil and Lowe have shown that bilateral TLS establishes strong stream channels [KL08] that satisfy the strongest channel specification: $C \wedge NF \wedge NRA \wedge NR$. We believe that unilateral TLS and SSL also establish strong stream channels, but that a weaker authentication specification is satisfied. Because these protocols are asymmetric, the channel specifications satisfied by the channels from the client to the server and from the server to the client are different.

- On the channel from the client to the server the client can be sure whom he is sending messages to. We use our model of $C \wedge NR^{-}$ for this channel: the channel is confidential, but the intruder could establish a session pretending to be an honest agent.
- On the channel from the server to the client the client can be sure whom he is receiving messages from. However, the server cannot be sure whom he is sending messages to, so the intruder could be receiving messages that the server believes he is sending to an honest agent. We use our model of $NF \wedge NRA^{-}$ for this channel.

  If the intruder pretends to be an honest agent and establishes a session with an honest server he can learn messages that were not intended for him. If an honest agent and an honest server establish an SSL or unilateral TLS session the intruder can overhear the transport-layer messages they exchange, but he cannot deduce the application-layer messages. Although the strong confidentiality definition of Chapter 3 is not satisfied, a weaker notion of confidentiality is satisfied. This weaker notion does not collapse on redirectable channels, so we model sessions over SSL or unilateral TLS as confidential, but we allow the intruder to redirect messages, and hence to learn the messages sent to an honest client if and only if he is faking the client role in the session.

We also use the symmetric session channel model for SSL and TLS connections; the protocols in this chapter all satisfy the *no speaking out of turn* property, so we do not need to use the stream channel models.

HTTP/1.1 [FGM+99] supports persistent connections; rather than establishing a new connection for each request, clients can keep their connection to a server open. This means that HTTP/1.1 could potentially provide

session continuation: all messages in a single session could be sent over a single connection. However, most web browsers are designed to close connections after retrieving a page and all the documents (such as stylesheets, images, etc.) associated with it. Most modern web browsers fully support TLS session resumption, so multiple connections can be unified by this means.

Paulson has proved that the TLS session resumption mechanism is secure even if previous keys from the session being resumed have been compromised [Pau99]. We argue therefore that we can treat multiple TLS sessions between two agents as being equivalent to a single session.

7.3 SAML Single Sign-On

The Security Assertion Markup Language (SAML) — developed by the Security Services Technical Committee of OASIS (the Organization for the Advancement of Structured Information Standards) — defines a framework for exchanging security assertions between SAML authorities. A security assertion is a package of information that carries statements made by a SAML authority about a subject (typically a user or another SAML authority). In this section we describe our model and analysis of the SAML Single Sign-On protocols, and we report on the attacks we have found against these protocols.

The Web Browser Single Sign-On Profile is one of several use cases described by version 2.0 of SAML [OAS05c, OAS05b, OAS07]. It outlines several protocols (based on variations of message flow and the bindings used to transport the protocol messages) that allow a user to authenticate himself to a SAML authority (the service provider) by means of an existing authenticated session with another SAML authority (the identity provider).

The SAML protocols are designed for web deployment; they must be functional for any user, so the only capabilities that can be assumed of the user's software are those that are standard for any web browser:

- The ability to create GET and POST requests and to process the response from such requests;
- The ability to accept, store and provide cookies (when requested);[1]
- The ability to initiate unilateral TLS/SSL sessions (and to verify the validity of a server's certificate).

[1] A cookie is a small text file that an HTTP server can ask a web browser to retain, and to present whenever the user requests a page from that server again.

7.3.1 SAML identities and bindings

The core building blocks of SAML are security assertions: authentication statements about subjects (users or SAML authorities). Version 2.0 of SAML specifies several different bindings and messages for exchanging assertions, but for our analysis of the Web Browser Single Sign-On Profile we only need to consider the SOAP, HTTP redirect, HTTP post and HTTP artifact bindings, and the authentication request and response messages.

In the SAML protocols identities can take many forms; for example, an identity may be an email address, an X.509 subject name, a Windows domain qualified name or a Kerberos principal name. In our analysis we treat all names abstractly: we assume that every agent and server has a unique symbolic identity that all the other agents recognise. Some of the protocol messages allow the SAML authorities to refer to the IP address from which a user is communicating; because IP addresses are easily spoofed, and because a user's IP address may change legitimately during a protocol run we do not include IP addresses in our model.

SOAP binding

The SOAP binding is based on version 1.1 of the Simple Object Access Protocol (SOAP) [W3C00]. SOAP is a lightweight protocol for exchanging structured information in a distributed environment; SOAP uses XML to define a messaging framework that can be used over a variety of underlying protocols. The SAML SOAP binding wraps SAML messages in SOAP messages, and then directly transmits the SOAP messages between SAML authorities.

We model a message exchange over the SOAP binding as a direct transfer of the original message from the sender to the receiver. Kleiner and Roscoe have shown how a SOAP message can be transformed into Casper input [KR04]; we rely on a similar transformation function and on Hui and Lowe's safe simplifying transformations [HL01] to transform the SOAP messages to simple messages that can be used in Casper.

HTTP POST and HTTP redirect bindings

The HTTP POST and redirect bindings are intended for use when SAML authorities need to communicate via an HTTP user agent (i.e. a web browser). The message flow for both bindings is the same:

1. The user agent (u) issues an HTTP request to the first server (s_1);
2. The first server responds to the request; the response contains the message to be delivered to the second server (m) and the second server's identity (s_2);

3. The user agent issues an HTTP request containing the message to the second server.

We model both of these bindings in the following way:

$$\begin{array}{ll} \textit{Message i} & u \rightarrow s_1 : \textit{request}_1 \\ \textit{Message ii} & s_1 \rightarrow u : \textit{response}, s_2, m \\ \textit{Message iii} & u \rightarrow s_2 : \textit{request}_2, s_1, m \end{array}$$

where $\textit{request}_1$, $\textit{request}_2$ and *response* are defined by the scenario in which the binding is used; in particular, they may be empty messages.

When the HTTP redirect binding is used, the message is encoded in a query string and the user agent is redirected (using a `Location` header) to the second server. When the HTTP POST binding is used, the message is encoded in a hidden form field in an XHMTL document; the form is configured to send an HTTP POST request to the second server. The first server may configure the form to submit automatically, or may allow the user to submit the form (by pressing a button). With both bindings, the contents of the message are not protected, and so they can be modified by the user agent, or modified in transit, and they should not be considered confidential.

The SAML specifications state that the HTTP POST and HTTP redirect bindings may also be run over HTTPS (i.e. HTTP with SSL or TLS). When they are run with SSL or TLS then the authentication, confidentiality and integrity properties guaranteed by the transport-layer protocol may be lifted to the SAML messages. When these bindings are run over HTTP (i.e. without SSL or TLS) then the SAML messages may be signed and encrypted in order to provide authentication, confidentiality and integrity properties. In the rest of this section we restrict our attention to the use of the HTTP POST binding, as the HTTP redirect and HTTP POST bindings are equivalent in our model.

HTTP artifact binding

The HTTP artifact binding allows SAML authorities to send messages by reference, rather than by value, via an HTTP user agent. An artifact is essentially a nonce: it is an unforgeable, unpredictable random string, which identifies the message creator, and can only be mapped to the original message by the message creator. We model an artifact as a nonce concatenated to the sender's identity (the mechanism suggested by the SAML documents is a hash of the sender's identity concatenated to a random number).

The message flow is as follows:

1. The message sender (s_1) creates an artifact (a) and sends that to the user (u) using one of the HTTP bindings;

2. When the message recipient (s_2) receives the artifact they use the SOAP binding to resolve the artifact: the message sender sends the message (m) directly to the message recipient.

We model the HTTP artifact binding in the following way:

$$\begin{array}{ll} \textit{Message } i & u \rightarrow s_1 : \textit{request} \\ \textit{Message } ii & s_1 \rightarrow u : \textit{response}, s_2, s_1, a \\ \textit{Message } iii & u \rightarrow s_2 : s_1, a \\ \textit{Message } iv & s_2 \rightarrow s_1 : a \\ \textit{Message } v & s_1 \rightarrow s_2 : m \end{array}$$

where *request* and *response* are defined by the scenario in which the binding is used; in particular, they may be empty messages.

The artifact has a single-use semantic: the SAML authority who issued an artifact for a message should only resolve the artifact, and send out the message, once. The SAML specification documents state that the artifact does not need to be signed, but should always be communicated on a confidential channel. The specification also states that the artifact resolution protocol (messages 4 and 5 above) should be mutually authenticated and integrity protected, either by signing and encrypting the SAML messages, or by using bilateral TLS.

7.3.2 SAML messages

SAML messages are XML documents; the SAML schema documents are published, so while SAML messages contain strong typing information, and many fields, their content is largely predictable or irrelevant to our model. We reduce each of the SAML messages to a message in the context of the CSP model: a sequence of the values that uniquely defines it. We use several of Hui and Lowe's simplifying transformations from [HL01] to reduce messages in this way. In particular:

- We ignore all message formatting information. SAML messages are strongly typed, so we must ensure that if protocol attacks are detected they are not due to typing errors.
- We ignore all message encodings. Some of the SAML bindings specify that messages should be encoded using the deflate and base64 encodings. These encodings are, essentially, isomorphisms between SAML messages and strings of characters, so we do not model them.
- We do not include duplicate fields in any messages. Many of the SAML messages contain identities, nonces or timestamps in several locations; we only include each unique item once. Some messages also include more than one fresh nonce; we only model one fresh nonce in each message.

- We do not model timestamps; if the protocol is free from attacks when the timestamps are not included, then it is certainly free from attacks when the timestamps are included in the messages.

Authentication request message

When a service provider wishes to obtain authentication statements about a user it can send an authentication request message to an identity provider. The authentication request message in the SAML specification contains several fields; the fields that we include in our model are as follows:

ID An identifier for the request; we model this field as a nonce: nsp;[2]

Issuer The identity of the server who issued the request; we model this as the service provider's identity: sp;

Subject The requested subject of the resulting assertion; we model this as the user's identity: u.

We model the authentication request message as: nsp, sp, u; this message is considerably shorter than the message specified by the SAML documents, however our analysis shows that these three fields are sufficient to ensure the security of the protocol.

Authentication response message

When an identity provider receives an authentication request message about a subject that they can authenticate they return a response message containing one or more assertions about the subject. If they cannot authenticate the subject the identity provider can either return an error message, or they can proxy the request to another identity provider. We are only interested in successful flows of the protocol, and we do not model request proxying, so we assume that the identity provider can always authenticate the requested user.

The response message in the SAML specification contains several fields, and at least one authentication assertion; we model this message as follows:

ID An identifier for the response; we model this field as a nonce: $nidp$;

InResponseTo The **ID** of the authentication request message: nsp;

Issuer The identity of the server who issued the response; we model this field as the identity provider's identity: idp;

[2]The specification requires that for pseudo-random IDs the probability of collision should be less than 2^{-128}.

Subject The subject of the assertion; we model this as the user's identity: u. This complex field also contains creation and expiry timestamps (which we do not model) and the identity of the SAML authority to whom the assertion can be delivered (the service provider's identity: sp);

AuthnStatement Describes the statement by the identity provider that the subject was authenticated at a particular time by a particular means. This field contains a creation timestamp and an expiry timestamp (after which the statement should not be relied upon). This field can also contain a DNS domain name or IP address for the user, but we treat all names and identities abstractly, so we model this as the user's identity: u.

If the HTTP POST (or HTTP redirect) binding is used to deliver the response message then the assertion must be protected by a digital signature. We model this as a signature on the entire message; in this case we model the response message as: $\{nidp, idp, nsp, u, sp\}_{SK(idp)}$. If the authentication response message is delivered by the artifact binding then the artifact resolution protocol must be mutually authenticated and integrity protected: we model this exchange as happening over a bilateral TLS channel. In this case we model the response message as: $nidp, idp, nsp, u, sp$.

7.3.3 SAML Web Browser Single-Sign On Protocols

There are two possible scenarios for use of the profile:

Service provider first In the first scenario the user requests a resource from the service provider first. If the service provider requires the user to be authenticated before they provide the resource they send a request to the identity provider to authenticate the user. If the user does not already have a session with the identity provider then one is created (after suitable authentication). The identity provider sends an assertion to the service provider to verify the user's identity.

Identity provider first In the second scenario the user identifies himself to the identity provider first. After this authentication, the identity provider creates a session for the user. When the user requests a resource from a service provider he requests an assertion from the identity provider, and presents that to the service provider to prove that he is who he says he is.

The method that the identity provider uses to authenticate the user is not specified by SAML, so we make it abstract: we model all of the communication from the user to the identity provider, and from the identity provider to the user, on an authenticated and confidential channel (i.e. the

top channel in the hierarchy). This is the same model that we use for bilateral TLS, but in this case it is intended to reflect unilateral TLS with user authentication at the application layer.

In all of the protocols the communication between the user and service provider takes place over a unilateral TLS connection; the communication between the user and identity provider takes place over a unilateral TLS connection in which the user is authenticated at the application layer; and the communication between the service provider and the identity provider takes place over a bilateral TLS connection.

Service provider first

The service provider first protocol has four steps:

1. The user requests a resource from the service provider; we model this request as the user's identity (u);
2. The service provider creates an authentication request message, and sends it (via the user) to the identity provider;
3. The identity provider authenticates the user and sends an assertion back to the service provider (via the user);
4. The service provider checks the assertion and provides the requested resource (m) to the user.

The specification allows POST or artifact binding for the delivery of the authentication request message (in stage 2) and of the authentication response message (in stage 3). There are four possible combinations of bindings, so there are four protocols to study; these are shown below.

POST–POST binding

$$
\begin{array}{lrcl}
\mathit{Message}\ 1 & u & \rightarrow\ sp & : u \\
\mathit{Message}\ 2.1 & sp & \rightarrow\ u & : idp, nsp, sp, u \\
\mathit{Message}\ 2.2 & u & \rightarrow idp & : nsp, sp, u \\
\mathit{Message}\ 3.1 & idp & \rightarrow\ u & : \{nidp, idp, nsp, u, sp\}_{SK(idp)}, sp \\
\mathit{Message}\ 3.2 & u & \rightarrow\ sp & : \{nidp, idp, nsp, u, sp\}_{SK(idp)}, idp \\
\mathit{Message}\ 4 & sp & \rightarrow\ u & : m
\end{array}
$$

```
1   C NR-
1.1 NF NRA-
1.2 C NF NRA NR
3.1 C NF NRA NR
3.2 C NR-
4   NF NRA-
```

```
Session symmetric 1, 2.1, 3.2, 4
Session symmetric 2.2, 3.1
```

POST–artifact binding

$$
\begin{array}{lll}
Message\ 1 & u \rightarrow sp & : u \\
Message\ 2.1 & sp \rightarrow u & : idp, nsp, sp, u \\
Message\ 2.2 & u \rightarrow idp & : nsp, sp, u \\
Message\ 3.1 & idp \rightarrow u & : sp, idp, a \\
Message\ 3.2 & u \rightarrow sp & : idp, a \\
Message\ 3.3 & sp \rightarrow idp & : a \\
Message\ 3.4 & idp \rightarrow sp & : nidp, idp, nsp, u, sp \\
Message\ 4 & sp \rightarrow u & : m
\end{array}
$$

```
1   C NR-
2.1 NF NRA-
2.2 C NF NRA NR
3.1 C NF NRA NR
3.2 C NR-
3.3 C NF NRA NR
3.4 C NF NRA NR
4   NF NRA-

Session symmetric 1, 2.1, 3.2, 4
Session symmetric 2.2, 3.1
Session symmetric 3.3, 3.4
```

Artifact–POST binding

$$
\begin{array}{lll}
Message\ 1 & u \rightarrow sp & : u \\
Message\ 2.1 & sp \rightarrow u & : idp, sp, a \\
Message\ 2.2 & u \rightarrow idp & : sp, a \\
Message\ 2.3 & idp \rightarrow sp & : a \\
Message\ 2.4 & sp \rightarrow idp & : nsp, sp, u \\
Message\ 3.1 & idp \rightarrow u & : \{nidp, idp, nsp, u, sp\}_{SK(idp)}, sp \\
Message\ 3.2 & u \rightarrow sp & : \{nidp, idp, nsp, u, sp\}_{SK(idp)}, idp \\
Message\ 4 & sp \rightarrow u & : m
\end{array}
$$

```
Session symmetric 1, 2.1, 3.2, 4
Session symmetric 2.2, 3.1
Session symmetric 2.3, 3.1

1   C NR-
2.1 NF NRA-
```

```
2.2 C NF NRA NR
2.3 C NF NRA NR
2.4 C NF NRA NR
3.1 C NF NRA NR
3.2 C NR-
4   NF NRA-
```

Artifact–artifact binding

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow sp : u \\
\mathit{Message}\ 2.1 & sp \rightarrow u : idp, sp, a \\
\mathit{Message}\ 2.2 & u \rightarrow idp : sp, a \\
\mathit{Message}\ 2.3 & idp \rightarrow sp : a \\
\mathit{Message}\ 2.4 & sp \rightarrow idp : nsp, sp, u \\
\mathit{Message}\ 3.1 & idp \rightarrow u : sp, idp, a' \\
\mathit{Message}\ 3.2 & u \rightarrow sp : idp, a' \\
\mathit{Message}\ 3.3 & sp \rightarrow idp : a' \\
\mathit{Message}\ 3.4 & idp \rightarrow sp : nidp, idp, nsp, u, sp \\
\mathit{Message}\ 4 & sp \rightarrow u : m
\end{array}
$$

```
1   C NR-
2.1 NF NRA-
2.2 C NF NRA NR
2.3 C NF NRA NR
2.4 C NF NRA NR
3.1 C NF NRA NR
3.2 C NR-
3.3 C NF NRA NR
3.4 C NF NRA NR
4   NF NRA-

Session symmetric 1, 2.1, 3.2, 4
Session symmetric 2.2, 3.1
Session symmetric 2.3, 2.4
Session symmetric 3.3, 3.4
```

Identity provider first

When the identity provider first protocols are used, the **InResponseTo** field in the authentication response message must be empty; we model this message as before, but we remove the service provider's nonce. The identity provider first protocol has three steps:

1. The user requests an assertion to present to the service provider; we model this request as the user's identity (u) and the service provider's identity (sp);

2. The identity provider creates an assertion and sends it (via the user) to the service provider;

3. The service provider checks the assertion and provides the requested resource (m) to the user.

The specification allows POST or artifact binding for the delivery of the authentication response message (in stage 2). This gives us two protocols to study; these are shown below.

POST binding

$$\begin{array}{lll} \textit{Message}\ 1 & u \rightarrow & idp : u, sp \\ \textit{Message}\ 2.1 & idp \rightarrow & u : \{nidp, idp, u, sp\}_{SK(idp)}, sp \\ \textit{Message}\ 2.2 & u \rightarrow & sp : \{nidp, idp, u, sp\}_{SK(idp)} \\ \textit{Message}\ 3 & sp \rightarrow & u : m \end{array}$$

```
1   C NF NRA NR
2.1 C NF NRA NR
2.2 C NR-
3   NF NRA-

Session symmetric 1, 2.1
Session symmetric 2.2, 3
```

Artifact binding

$$\begin{array}{lll} \textit{Message}\ 1 & u \rightarrow & idp : u, sp \\ \textit{Message}\ 2.1 & idp \rightarrow & u : sp, idp, a \\ \textit{Message}\ 2.2 & u \rightarrow & sp : idp, a \\ \textit{Message}\ 2.3 & sp \rightarrow & idp : a \\ \textit{Message}\ 2.4 & idp \rightarrow & sp : nidp, idp, u, sp \\ \textit{Message}\ 3 & sp \rightarrow & u : m \end{array}$$

```
1   C NF NRA NR
2.1 C NF NRA NR
2.2 C NR-
2.3 C NF NRA NR
2.4 C NF NRA NR
3   NF NRA-

Session symmetric 1, 2.1
Session symmetric 2.2, 3
Session symmetric 2.3, 2.4
```

7.3.4 Attacks against the protocols

We used the models of the SAML Single Sign-On protocols described above to test several authentication properties in Casper. Specifically, we tested an authentication property between every pair of roles who communicate directly in the protocol. Of these, the most important property is the authentication of the user to the service provider because achieving this correctly is the goal of the protocols.

When we allow the intruder to play the role of the identity provider FDR detects three classes of attack against the authentication specifications:

- The intruder can pretend to be any user to any service provider;
- The intruder, acting as the identity provider, can fail to authenticate an honest user to an honest service provider (this is a denial of service attack and not an attack against authentication);
- The intruder can authenticate an honest user to an honest service provider when they should not (i.e. the intruder can tell the service provider that they have authenticated the user, even if they have not done so).

All of these attacks are only possible when the intruder can play the identity provider role; this highlights the importance of the trust relationship between the SAML authorities. It is not surprising that there are attacks against the protocol (against the service provider and against the user) when the intruder can play the role of the identity provider: the basic tenet of the protocol is that the identity provider is used to authenticate the user to the service provider, removing the need for the user to authenticate himself directly. We henceforth allow the intruder to play the user and service provider roles, but not the identity provider role.

The SAML specification documents do not require the authentication request message to be signed by the service provider. However, when this message is not signed there is an attack against the authentication check between the service provider and the identity provider. A dishonest user can create fake authentication requests (i.e. create authentication requests and send them to identity providers), and the protocol can be run without the service provider taking part. This attack is only possible against the protocols that use the HTTP POST or redirect bindings to transport the authentication request; i.e. the protocols where the service provider and the identity provider do not communicate directly. Because the authentication request message includes the identity of the user, the supposed request issuer and the asserting party, together with a new ID, this attack does does not allow a dishonest user to be authenticated to a service provider erroneously as the dishonest user does not have a correct session with the identity provider, and cannot establish one.

There are possible replay attacks if the protocol is not implemented correctly by the identity provider. The SAML documents state that the identity provider should keep a list of the requests he has responded to for as long as they are valid[3] so that he can detect replay attacks. If this is not implemented correctly then a dishonest agent, using his own identity, may replay a signed authentication request and receive a new assertion. This attack manifests itself as a run of the protocol that the service provider has not taken part in (i.e. the identity provider to service provider agreement check fails).

If, at any point in a run of the protocol, the intruder gets hold of an assertion he can use it to authenticate himself to the service provider who first requested the assertion. For this reason, the assertions must always be sent on confidential (and, where appropriate, authenticated) channels. When the HTTP POST or HTTP redirect binding is used to transport the authentication assertion from the identity provider to the relying party the assertion may be stored on the user's computer (either in their browsing history in the case of the redirect binding, or in the cached copy of the HTTP POST form submission page). If the user's computer is compromised before the timestamp on the assertion expires the intruder may be able to replay the assertion to assume the user's identity at the service provider. In order to prevent this attack the service provider should keep track of the identifiers in the authentication assertions he has accepted (at least for the duration of the timestamp on the assertions), and he should only accept each assertion once.

Similarly, if any of the messages containing an artifact are sent over a non-confidential channel, the intruder can learn the artifact. If the artifact is for an authentication response message, the intruder can use it to assume the user's identity. Because of the single use semantic of the artifact, the intruder would have to know the correct identity to use; he could have at most one guess, though he would probably increase his chances of guessing correctly by analysing the network traffic. For this reason, we believe that the specification documents should *require* the artifact to be sent on a confidential channel (rather than recommending that it should be).

When the artifact binding is used to transfer the authentication response message, which contains the assertion, the intruder, playing the service provider role, can choose not to resolve the assertion, but just to assume that the message it points to is an authentication passed (rather than failed) message. There is nothing the service provider can gain from failing to resolve the artifact, but if a dishonest user knew that a service provider did not resolve artifacts he could pretend to be someone else. This is an unlikely scenario, because at the point of receiving an artifact a service provider would

[3]The validity of the request message is defined by a timestamp value (which we do not model) and the specific set-up in use.

only know the user's claimed identity, and would have no reason to trust this.

In Section 7.2 we asserted that we could model two TLS sessions that are linked by the session resumption mechanism as a single session. This only applies to the SAML Web Browser Single Sign-On Profile when the protocol is run in service provider first mode: in this case, the user and the service provider use two sessions during the protocol run. The first session contains the user's initial request, and the service provider's authentication request message; the second session contains the identity provider's authentication response message, and the final message from the service provider. Our analysis shows that we do not need to treat these two message flows as a single session: when we model the two sessions separately, no new attacks are introduced.

If the authentication request messages are signed by the service provider, the single-use property of artifacts is correctly implemented, artifacts are only ever transmitted on confidential channels, the identity provider only responds to each authentication request once and service providers keep track of which assertions they have previously accepted we can find no further attacks to the protocols.

Recently, Armando et al. analysed the service provider first protocol with HTTP POST and redirect binding [ACC+08], and Google's implementation of this protocol for Google Applications [Goo08]. The model of the protocol in [ACC+08] is the same as the model we presented earler in this section (although it is written using slightly different notation, and some duplicate fields are not removed). Critically though, the implementation of the protocol that Google created deviates from the models in [ACC+08] and in this section in two ways:

- The **InResponseTo** field (i.e. the authentication request identifier) and the service provider's identity are omitted from the authentication assertion;
- The **InResponseTo** field, the service provider's identity and the identity provider's identity are omitted from the authentication response message.

The service provider's identity and the authentication request identifier do not appear anywhere in the authentication response message. This allows assertions provided to one service provider to be presented to a different service provider, in response to a different request, to authenticate the user. This leads to an attack where a dishonest service provider running the protocol with an honest user can replay the assertion he receives to a different service provider to impersonate the user. Armando et al. did not find any attacks against the protocol as specified by [OAS05b].

Other researchers have analysed an earlier version of the Single Sign-On Profile (defined in SAML 1.1 [OAS04]), for example [Gro03] and [HSN05]. However, we have found no complete analysis of all the message flows, and no analysis which examines all of the possible bindings. Our investigation of the new protocols in the latest version of SAML provides a useful confirmation of the earlier results, and serves to highlight the importance of several assumptions and comments in the technical specifications.

7.3.5 Conclusions

Most of the security of the SAML single sign-on protocols is provided by the secure transport layer and the inherent trust that the users and service providers have in the identity provider. There are several attacks against the users and against the service providers when the intruder can take the role of the identity provider, but there are no significant attacks against the authentication of the user to the service provider when the identity provider is honest. Those attacks that do exist are only possible if the protocols are not implemented correctly according to the SAML specifications.

The SAML single sign-on protocols are token authentication protocols: the identity provider gives the user a token to present to the service provider to authenticate himself. Anyone who receives a token asserting the identity of a user can pretend to be that user. A stronger protocol would not rely on the confidentiality of the token to provide authentication. Ideally, a single sign-on system should protect the users and service providers when the identity provider cannot be trusted; in other words, we at least require that a user really was trying to authenticate himself to a service provider for that service provider to accept an assertion from someone claiming to be the user. We may not be able to prevent man-in-the-middle attacks with an untrustworthy identity provider, but we could limit the intruder's capabilities.

7.4 OpenID Authentication

The OpenID Authentication specification [FRH+08] was developed by the OpenID Community, and is based on the original specification for OpenID Authentication written by Fitzpatrick and Recordon [FR06]. OpenID Authentication has recently been adopted by Myspace, Flickr, Yahoo!, and Sourceforge to allow their users to authenticate themselves to other websites. Although these four large community sites only act as identity providers and do not accept authentication statements from other OpenID providers, the adoption of the protocol by such major sites as these is seen as a major step towards wider adoption of the OpenID protocols.

The OpenID Authentication protocols provide a way for a user to prove that he controls a particular identifier without having to provide sensitive in-

formation (such as an email address) or credentials (such as a password). Unlike the SAML Single Sign-On Profile, the OpenID protocols are structured in a decentralised manner: the identity providers (OpenID providers) are not registered with any central authority, so relying parties (agents who accept authentication statements from OpenID providers) must choose whether or not to trust the OpenID providers that their users rely on to prove their identities.

The principal aim of OpenID Authentication is to ease the means by which users authenticate themselves to web sites and web services. In order to ensure that as many users as possible can use OpenID, no special capabilities are required of the user's web browser (the user agent): the protocols use standard HTTP (and HTTPS) requests (as described in Section 7.3). If the user and his identity provider have some means of keeping an authenticated session open the OpenID protocols do not require any visible interaction between the user and the identity provider. If the user's OpenID provider has not yet authenticated the user, or if their most recent authenticated session has expired, the protocol is readily adapted to allow the OpenID provider to create a fresh authenticated session with the user.

In the rest of this section we describe the types of identities and messages that the OpenID protocols use and the types of secure channels that the specification recommends. We present our models of the protocols and we report the attacks that we have found against the protocols. We conclude by discussing the strengths and weaknesses of the OpenID Authentication scheme, and we discuss some of the areas that still need to be worked on to improve the security of the protocols.

7.4.1 OpenID identities and signatures

An OpenID identifier is either an HTTP or HTTPS URI (a URL) or an XRI (Extensible Resource Identifier) [OAS05a]. An XRI is an abstract identifier that identifies a resource so that it can be referenced across multiple domains and transports, and that maintains a persistent link to the resource, even if its network location changes. For simplicity, we treat all OpenID identifiers as URLs, and we model these as abstract identities.

By using URLs as identifiers users can uniquely identify themselves without disclosing any personal information (such as an email address). For example, users may choose to use local identities at their OpenID providers (e.g. `http://username.myopenid.com/`), or they may use their own domain names (e.g. `http://www.user.name/`) and just rely on OpenID providers for authentication. The OpenID scheme allows users to change OpenID providers while maintaining the same identity (either the local identity if it is available at the new provider, or their own domain names).

The signature algorithm supported (and recommended) by the OpenID Authentication specification is HMAC using SHA [KBC97, Fed02] with ei-

ther a 160-bit or 256-bit key and output (in OpenID the key used for HMAC is referred to as an association). We model this signature scheme by a single application of a cryptographic hash function where the hash key (the association) is included in the list of hashed fields, but is not included in plain text in the message. For example, in order to send the signed message m an agent would send $m, h(k, m)$, where k is the association being used as a key for the hash function.

7.4.2 OpenID protocol messages

OpenID protocol messages are either sent directly between relying parties and OpenID providers, or they are sent indirectly via the user agent. In this section we describe our models of direct and indirect communication; we then describe our models of all the possible phases of a complete OpenID Authentication protocol run.

Direct communication

Direct communication is always initiated by the relying party: the request is encoded as an HTTP POST request, and the response is a standard HTTP response with the message in the body. The HTTP encoding (used for the request) encodes the message in a query string: all field names are prefixed with "`openid.`", then fields and their values are linked by an "`=`" symbol, and fields are concatenated with an "`&`". For example:

```
?openid.ns=http://specs.openid.net/auth/2.0&openid.mode=
check_authentication&openid.identity=http://user.name/
```

The response message is encoded using key-value form encoding: each key-value pair appears on a new line, and the key and its value are separated by a "`:`". For example:

```
ns:http://specs.openid.net/auth/2.0
mode:check_authentication
identity:http://user.name/
```

The request and response messages must always include the OpenID version number (`openid.ns`) and the message type (`openid.mode`). However, in order to simplify our model we do not include these fields. By omitting the `mode` of messages we risk introducing attacks where a message of one type could be confused for a different message. However, we believe that the types of the messages, and the protection provided by the transport layer will mean that no attacks of this form are introduced.

We model direct communication by messages sent directly between the relying party and the OpenID provider:

$$\begin{array}{ll} \mathit{Message}\ i & rp \rightarrow op : \mathit{request}? \\ \mathit{Message}\ ii & op \rightarrow rp : \mathit{response}\,. \end{array}$$

Indirect communication

Some of the protocol messages are sent indirectly via the user agent; indirect communication may be initiated by either the relying party or the OpenID provider. Indirect communication may be accomplished in two different ways:

1. The message sender uses a `Location` header to redirect the user agent to the message recipient; the message body is encoded in a query string (as above) and appended to the recipient's URL;

2. The message sender sends an HTML page to the user agent; the HTML page contains a form that is configured to submit a POST request to the message recipient. This causes the user agent to generate a query string encoding of the message, and this is delivered to the message recipient.

As with direct communication the messages sent via the user agent must always include the OpenID version number and the message type, and all keys must be prefixed with `openid`. We model indirect communication in the following way:

$$\begin{array}{ll} \textit{Message } i & i \rightarrow u : r, m \\ \textit{Message } ii & u \rightarrow r : m \end{array}$$

where u is the user agent, i the message sender (the initiator), r the message recipient and m the message. Note that our model of the indirect binding does not include the message sender's identity when the message is delivered to the recipient; in reality this field may be provided if the user agent submits a `Referrer` header, but many user agents (and users) choose to prevent this header from being sent. In the OpenID protocols, the message itself always contains the sender's identity.

The OpenID specifications define seven stages to the authentication protocol, however there is not a unique order in which the stages are run: there are several allowable alternative ways of running the protocol. The seven stages are as follows:

1. **Initiate authentication** The user initiates authentication by presenting an identity (either his own identity, or the identity of an OpenID provider) to the relying party, or by telling his OpenID provider which relying party he wishes to communicate with;

2. **Discovery** The relying party performs discovery on the identity presented by the user to determine whether it is an OpenID provider's identity or a user's identity. If the user has presented his own identity discovery will return with the identity of the user's OpenID provider;

3. **Establish association** The relying party and the OpenID provider establish an association (a shared secret) to be used to sign and verify subsequent messages. This stage is optional: they may use a previously established association as long as it has not expired or been invalidated. Alternatively, the relying party may decide to perform direct verification of the authentication response later in the protocol run;

4. **Authentication request** The relying party redirects the user to the OpenID provider using either HTTP redirection or HTTP form redirection;

5. **Authentication** The OpenID provider authenticates the user (either freshly or using an existing authenticated session);

6. **Authentication response** The OpenID provider redirects the user to the relying party with the authentication response message. If the relying party specified an assertion in the authentication request message the OpenID provider uses that association to sign the response; if not, the OpenID provider creates a fresh (private) association and uses that to sign the response message;

7. **Direct verification** If the response message was signed with a private association the relying party communicates directly with the OpenID provider to verify the signature of the authentication response.

In the rest of this section we describe these protocol stages in more detail.

1. Initiate authentication

The protocol is initiated by the user when he wants to authenticate himself to a relying party. Usually the user requests a resource from the relying party for which the relying party requires proof of his identity. The relying party sends an HTML form to the user with a text field for him to enter his identity; this form is then sent back to the relying party. We assume that the protocol starts at the final stage of this process: the user sends his identity (or the identity of his OpenID provider) to the relying party:

$$\mathit{Message}\ 1 \quad u \rightarrow rp : u\,,$$

or:

$$\mathit{Message}\ 1' \quad u \rightarrow rp : op\,.$$

2. Discovery

During discovery the relying party uses the identity provided by the user to look up the endpoint URL of the user's OpenID provider. The relying party can perform discovery in two different ways: either by using the Yadis protocol [Mil06], or by requesting the URL and looking for information in the response that identifies the OpenID provider.

In order to use the Yadis protocol, the relying party makes an HTTP HEAD or GET request to the URL provided by the user. The result of this request will be an Extensible Resource Descriptor Sequence (XRDS) document (or the location of an XRDS document); this XML file should contain, among other information, the OpenID provider's endpoint URL (their identity) and, if appropriate, the user's local identity at the OpenID provider. If the Yadis protocol fails, the relying party should make an HTTP GET request to the URL provided by the user and search the resulting document for `link` tags identifying the OpenID provider and the user's local identity at the OpenID provider (where appropriate).

If the user provides his own identity then we assume that he is in control of the URL, and so he also controls the result of the HTTP request. We model the discovery protocol in this case as a simple HTTP exchange between the relying party and the user:

$$\begin{array}{ll} \mathit{Message}\ 2.1 & rp \rightarrow u : u \\ \mathit{Message}\ 2.2 & u \rightarrow rp : u, op\,. \end{array}$$

If the user provides an OpenID provider's identity then only the Yadis protocol may be used, and only the OpenID provider's identity will be returned (the user's identity and his local identity will be provided to the relying party later). We omit the discovery protocol in this case because the only result would be the OpenID provider's identity, and the user provided this in the first stage of the protocol. We model the user's identity and his local identity at the OpenID provider as the same abstract identity: u.

3. Establish association

The authentication response message (the sixth stage) is sent indirectly from the OpenID provider to the relying party, so the relying party must have some means of verifying that the message was originally sent by the OpenID provider. In the OpenID protocols the OpenID provider signs the authentication response message using the HMAC keyed-hashing algorithm. In order to use this mechanism the relying party and the OpenID provider need to have a shared secret (an association) to be used as the hash key.

If the relying party and the OpenID provider still have a valid association from a prior protocol run then they can skip this step and use the existing association for this protocol run. Alternatively, if the relying party cannot,

or does not want to, maintain a list of associations it can choose to run the protocol in 'stateless' mode: this allows them to verify the authentication response message directly at the end of the protocol run.

OpenID defines two different ways of establishing associations: either by performing a Diffie-Hellman key exchange [DH76] or by sending the MAC key unencrypted (at the application layer). Because unauthenticated Diffie-Hellman key exchanges are vulnerable to man-in-the-middle attacks they are not suitable for establishing shared secrets (i.e. using a Diffie-Hellman key exchange to create an association could allow an intruder to create a different association with a relying party and an OpenID provider, but lead them both to believe they had created an association with one another).

Rather than model the Diffie-Hellman scheme (which we know will lead to an attack against the OpenID protocol), we only consider the unencrypted method. An alternative would be to model a Diffie-Hellman exchange that takes place over an authenticated connection (e.g. SSL or TLS). However, if the exchange is run over an SSL or TLS connection the Diffie-Hellman exchange is unnecessary: the OpenID provider can just send the HMAC key directly.[4]

With the unencrypted method the relying party sends a direct message to the OpenID provider requesting a new association; the OpenID provider then sends a direct message back to the relying party containing the new association k (the HMAC key), a handle to that key n_k (a nonce that they can both use to refer to the association) and an integer value t (the lifetime of the association in seconds).

For simplicity we do not model the lifetime of the association in our model as we assume that a new association is created for every run of the protocol. The OpenID specifications require that the unencrypted association establishment method is only used in conjunction with transport-layer encryption, but does not specify which form of SSL or TLS to use. For our initial model we assume the weakest possible form: an SSL session between the relying party (the client) and the OpenID provider (the server).

We therefore model the association establishment protocol in the following way:

$$\begin{array}{ll} \mathit{Message}\ 3.1 & rp \rightarrow op : rp, op \\ \mathit{Message}\ 3.2 & op \rightarrow rp : n_k, k\,. \end{array}$$

4. Authentication request

The authentication request message is an indirect message from the relying party to the user's OpenID provider requesting that the OpenID provider

[4]This does not give the relying party a guarantee that the key is freshly generated, but this guarantee is not necessary: we found that modelling the protocol in this way does not cause any attacks.

authenticate the user (if they have not already done so) and return an authentication assertion to the relying party. Our model of this message includes the following fields:

claimed_id The identity the user initially presented to the relying party; if this is an OpenID provider's identity this should have the special value `http://specs.openid.net/auth/2.0/identifier_select`; otherwise, this is the user's identity: u;

identity The user's local identity at his OpenID provider (as found during discovery). If the user presented his OpenID provider's identity in stage 1 this should also have the special value above indicating that the OpenID provider should choose an identifier that belongs to the user; otherwise, this is the user's identity u;

assoc_handle A handle for an association between the relying party and the OpenID provider. This field is optional; if it is omitted the transaction takes place in stateless mode, and the relying party must use direct verification to verify the authentication response message; otherwise, we model this field as the nonce n_k that was established during the association establishment protocol (stage 3);

return_to The URL to which the OpenID provider should direct the user to return the authentication response message; we model this field as the relying party's identity: rp.

The OpenID specification does not make any recommendations about using SSL or TLS for the connection between the user and the relying party at this point in the protocol, so we model the authentication request message being sent on an unauthenticated and non-confidential channel (i.e. $\bot$).

We model the indirect communication of the authentication request message in the following way:

$$\begin{array}{ll} \textit{Message } 4.1 & rp \rightarrow u : op, u, n_k, rp \\ \textit{Message } 4.2 & u \rightarrow op : u, n_k, rp\,. \end{array}$$

5. Authentication

When an OpenID provider receives an authentication request message he should determine who the user is and whether or not he can attest to that user's identity. The means by which the authentication of the user is accomplished is out of the scope of the protocol description, so we model the message that the user sends to the OpenID provider (*Message* 4.2) as being sent on an authenticated channel (i.e. a channel that satisfies at least $NF \wedge NRA^{-}$).

If the relying party asked the OpenID provider to assist the user in choosing an identity then the OpenID provider should communicate with the user and help him to choose an identity. We model this exchange by including the user's identity in the authentication message from the user to the OpenID provider, even if the identity was not included in the relying party's initial request message.

6. Authentication response

Once the OpenID provider has authenticated the user he sends an authentication response message back to the relying party via the user. Our model of this indirect message contains the following fields:

op_endpoint The endpoint URL for the OpenID provider; we model this as the OpenID provider's identity *op*;

claimed_id The user's claimed identity; this is either the value that the user initially presented to the relying party, or if the OpenID provider has helped the user to choose an identity, this is the identity they chose; in both cases, we model this as the user's identity u;

identity The user's local identity at the OpenID provider; as with the **claimed_id** this is either the value the user provided to the relying party or the user's identity at the OpenID provider; we model this field as the user's identity u;

return_to A verbatim copy of the same parameter from the authentication request message (*rp*);

response_nonce A string that is unique to this particular authentication response; the nonce includes a timestamp indicating the current time according to the OpenID provider's clock and it may contain additional characters as necessary to make it unique; we omit the timestamp and model this field as a nonce: n_r;

assoc_handle The handle for the association that was used to sign the authentication response; if the relying party did not specify an association in the request message (or if there was not a request message) then this should be the handle for a new private association; we model this field as the nonce n_k.

We model the OpenID provider's signature on the message as a single application of a cryptographic hash function to the hash key concatenated to the values to be signed; we include all of the necessary fields in the signature. The indirect communication of the authentication response message

is therefore modelled in the following way:

$$\begin{array}{ll} \textit{Message } 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\ \textit{Message } 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k)\,, \end{array}$$

where k is the association that n_k refers to.

The specification states that relying parties should accept and verify unrequested associations, so the protocol can be run in two ways: either in OpenID provider first mode or relying party first mode, depending on whom the user first communicates with.

When the protocol is run in OpenID provider first mode, or when the user provides his OpenID provider's identity in the first stage of the protocol (i.e. when the relying party has not yet performed discovery on the claimed identifier) the relying party must perform discovery on the claimed identifier. This is to ensure that the OpenID provider who signed the authentication response is authorised (by the user) to make assertions about the claimed identifier. This discovery phase is modelled in the same way as the earlier discovery protocol (messages 2.1 and 2.2).

The OpenID Authentication specification document recommends that the OpenID provider's endpoint URL should always be protected by an SSL certificate; this is to ensure that the user can correctly identify the OpenID provider. It also serves another very important purpose which is not directly referred to by the specification: the SSL channel protects the confidentiality of the authentication response message. If anyone other than the user learns the content of the authentication response they can present it to the relying party and impersonate the user. We model the communication channel between the user and the OpenID provider as an SSL connection (the user plays the client role, the OpenID provider plays the server role). The authentication of the user to the server at the application layer elevates the properties of the secure channel; we model a strong symmetric channel that satisfies $C \wedge NF \wedge NRA \wedge NR$ in both directions.

The specification does not make any recommendations about the channel between the user and the relying party. However, it is vital that the authentication response is delivered to the relying party on a confidential channel. We model the channel between the user and the relying party as an SSL connection (the user plays the client role, the relying party plays the server role).

7. Direct verification

If the authentication response message is signed with a private association the relying party must use the direct verification protocol to verify the signature. The protocol is initiated by the relying party sending a direct message to the OpenID provider including an exact copy of all the fields in the authentication response message.

When an OpenID provider receives a direct verification request message he should first check that the association handle refers to a private association, and that he has not already responded to a verification check for the **response_nonce** in the request. If both of these conditions are met, and if the signature is valid the OpenID provider should send a direct message to the relying party including an **is_valid** field set to true. Rather than introducing a new variable, we just model this message as a confirmation of the user's identity. The direct verification protocol is modelled in the following way:

$$\begin{array}{ll} \textit{Message } 7.1 & rp \rightarrow op : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\ \textit{Message } 7.2 & op \rightarrow rp : u\,. \end{array}$$

The OpenID specification recommends that the OpenID provider's endpoint URL should be protected by an SSL certificate, so we model the communication channel between the relying party and the OpenID provider as an SSL connection (the relying party plays the client role, the OpenID provider plays the server role).

7.4.3 OpenID protocols

The messages of the OpenID Authentication protocol can be exchanged in six different ways. When the protocol is run in OpenID provider first mode there are two possibilities: the user either identifies himself by using a domain name that he owns, or he uses a local identifier at his OpenID provider. The first message in both variations is a message from the user to his OpenID provider; this message is used to authenticate the user's identity, and to tell the OpenID provider which relying party he wishes to communicate with:

$$\textit{Message } 1 \quad u \rightarrow op : u, rp\,.$$

The OpenID provider then sends an authentication response message, via the user, to the relying party (messages 6.1 and 6.2). When the relying party receives this authentication message he performs the discovery protocol (messages 3.1 and 3.2) to make sure that the OpenID provider who signed the message is authorised to make assertions about the claimed identifier.

The OpenID provider uses a private association to sign the authentication message, so the relying party must use the direct verification protocol (messages 7.1 and 7.2) to verify the signature. Once the relying party has verified the message signature he can be sure that the user has correctly been authenticated to him, and so he provides the requested resource to the user.

When the protocol is run in relying party first mode there are four possibilities: the user either provides his own identity or he provides an OpenID

provider's identity, and the relying party either establishes an association before sending the request message or the relying party and the OpenID provider use the direct verification protocol to verify the signature.

In the two variations when the user provides his own identity the first message is message 1; the relying party then performs the discovery protocol (messages 2.1 and 2.2) to discover the user's OpenID provider. If the user provides his OpenID provider's identity (message $1'$) then the relying party performs discovery on it, but we do not model this stage.

The discovery protocol uses an HTTP connection if the identity is an HTTP URL, or an HTTPS connection if the identity is a secure URL. We assume that all user-owned URLs are not secure (i.e. they are HTTP URLs) because we assume that most users do not have an SSL certificate installed on their own domain; we assume that all identities provided by OpenID providers are secure URLs.

After the relying party has discovered the user's OpenID provider's identity he can either run the association establishment protocol (messages 3.1 and 3.2) or choose not to (either to use an existing association, or to use direct verification later). The relying party sends an authentication request message to the OpenID provider via the user (messages 4.1 and 4.2). The OpenID provider authenticates the user, and then sends an authentication response back to the relying party (messages 6.1 and 6.2). Finally, if necessary, the relying party runs the direct verification protocol (messages 7.1 and 7.2) before providing the user with the resource he requested.

Full listings of the six variations of the protocol and the Casper channel models we use for our analysis are shown in Appendix C.

7.4.4 Attacks against the protocols

We used the models of the OpenID Authentication protocols described above to test several authentication properties in Casper. Specifically, we tested an authentication property between every pair of roles who communicate directly in the protocol. Of these, the most important property is the authentication of the user to the relying party because achieving this correctly is the goal of the protocols.

In contrast to the trusted Identity Providers in the SAML Single Sign-On protocols, we cannot assume that the OpenID providers are trustworthy: we must allow the intruder to play the role of an OpenID provider, as well as allowing him to play the roles of users and relying parties. Naturally, we expect to find attacks against users who rely on dishonest OpenID providers to prove their identity. However, a very important property that a single sign-on system should satisfy is that it protects users against dishonest identity providers that they do not trust; in other words, an OpenID provider can only authenticate a user to a relying party identity if the user has said (either publicly, or within the protocol run) that they trust that OpenID

provider. We found several attacks against the OpenID protocols that show that they do not necessarily have this property.

In the rest of this section we describe the attacks that we found against the OpenID protocols. For each attack we describe a scenario in which it is possible, we describe the consequences of the attack and the steps that can be taken to prevent it.

If a user has an insecure identity (i.e. an HTTP URL rather than an HTTPS URL) the intruder can spoof the discovery phase of the protocol and change the identity of the user's OpenID provider; this attack can be performed against any of the protocols where the user presents an identity that he controls. Below, we give a trace of the relying party first protocol with verification by association in which the intruder has impersonated the user A. RP is the relying party; OP is A's (honest) OpenID provider, and the intruder (I) can act as an OpenID provider using his own identity.

$$
\begin{array}{ll}
\mathit{Message}\ 1 & I_A \rightarrow RP : A \\
\mathit{Message}\ 2.1 & RP \rightarrow I_A : A \\
\mathit{Message}\ 2.2 & I_A \rightarrow RP : A, I \\
\mathit{Message}\ 3.1 & RP \rightarrow I : RP, I \\
\mathit{Message}\ 3.2 & I \rightarrow RP : N_K, K \\
\mathit{Message}\ 4.1 & RP \rightarrow I_A : A, N_K, RP \\
\mathit{Message}\ 6.2 & I_A \rightarrow RP : I, A, RP, N_R, N_K, h(K, I, U, RP, N_R, N_k) \\
\mathit{Message}\ 8 & RP \rightarrow I_A : M\ .
\end{array}
$$

Message 1 is the initial phase of the protocol where the intruder impersonates the user and requests a resource from the relying party. Messages 2.1 and 2.2 are the discovery protocol; in this attack the intruder intercepts the relying party's request to the user's identity URL and sends back his own response, identifying himself as the user's OpenID provider. Messages 3.1 and 3.2 are the association establishment protocol. Message 4.1 is the first stage of the indirect communication of the authentication request message from the relying party to the OpenID provider (in this case, the intruder). The intruder, acting as the user, sends back the authentication response to the relying party in message 6.2. In message 8 the relying party sends the resource (M) to the intruder. The intruder has successfully authenticated himself as the user A to the relying party even though A did not take part in the protocol run.

This attack is possible because the discovery response message (message 2.2 above) is not sent on an authenticated channel. In order to prevent this attack, the message from the user that identifies his OpenID provider must be authenticated by some means.

When we model security protocols, we usually assume that users are played by a single agent process, that they do not have any specialised security software on their computer and that they do not possess any credentials

other than shared secrets with other agents (e.g. passwords). The OpenID discovery protocol is performed by the relying party, and the point of a single sign-on system is that the user and the relying party do not share any secrets, so there does not appear to be any way to authenticate the discovery protocol.

However, in the case of the OpenID Authentication scheme, two of the assumptions above are no longer valid:

1. The role of a user who controls his own identity is no longer played by a single agent process: the role is played by the web server that responds to HTTP requests at the user's identity URL and by the user himself at his own personal computer;

2. A user who controls his own identity can establish some credentials by installing an SSL certificate on the server that responds to HTTP requests at his identity.

With an SSL certificate installed on the user's identity URL, the discovery process can take place over an SSL connection, and the message from the user's web server to the relying party can be sent over the authenticated half of an SSL connection. This simple fix prevents the attack shown above. We recommend that insecure URLs should not be allowed to be used as identities within the OpenID protocols.

A dishonest OpenID provider can assume the identity of any user for whom he controls a local identifier. This is not surprising because it is the OpenID provider who proves the user is who he says he is. However, it is worth bearing in mind that users should only establish identities with providers that they trust, and relying parties should only allow OpenID providers whom they trust to authenticate their users to them.

The relying party always plays the client role when establishing SSL connections to the OpenID provider, so the relying party's identity is never authenticated to the OpenID provider. This means that there are some possible protocol runs in which an OpenID provider believes that a relying party has been running the protocol when they have not. These could easily be prevented by running the direct verification protocol and the association establishment protocol over a bilateral TLS connection, rather than an SSL connection. However, the added complexity of the stronger channel is unnecessary because without it there are no attacks against the authentication of the user to the relying party.

A relying party may choose not to perform discovery or not to verify the authentication message (either directly or by checking the signature). Although the user always communicates with his OpenID provider during the protocol run, he does not receive any guarantee that the relying party has communicated with the OpenID provider. This only leads to attacks against the protocol if an intruder knows that a certain relying party does

not perform discovery or does not check message signatures. In this case the intruder can assume any identity with the relying party. This attack highlights the trust that users must have in relying parties before they provide any personal information to them.

When the protocol is run in relying party first mode, the first time that the user communicates with his OpenID provider may be when he is redirected to the provider by the relying party. We assume that the user is capable of correctly identifying his OpenID provider, and would notice if he was sent to another website. Unfortunately, because of the way the OpenID protocols are designed to be implemented, this may not be the case: the OpenID provider's website may appear in a frame or a popup on the relying party's website, and so the user may not necessarily be able to tell whether or not they are really communicating with the correct provider. The only way to ensure that the user can check the identity of the OpenID provider is for the user to be redirected to the provider's website in a standard web page view so that he can easily check the details of the SSL certificate. An alternative is for the user to programme his browser with the identities (i.e. the URLs) of the OpenID providers that he trusts; however, support for this is currently limited to Mozilla Firefox, and this mechanism cannot be used when the user uses a different computer.

The security of the OpenID protocols relies heavily on the secrecy of the authentication response message. The OpenID provider should only give this message to the user once he has correctly authenticated him. However, anyone who gets hold of an authentication response message can identify himself as the user to whom it refers by presenting it to the relying party it was originally intended for. The message is protected by a timestamp and a nonce; these are designed to protect against replay attacks. However, if an intruder gets hold of an authentication response message and presents it to the relying party before the user does he can authenticate himself as the user. In order to prevent this sort of attack, the authentication response message must only be communicated on secret channels.

For the OpenID protocols this means that all communication between the user and the two servers must take place over SSL connections. The SSL connection from the user to the OpenID provider only provides $NF \wedge NRA^{-}$ from the OpenID provider to the user; however, the user authenticates himself to the OpenID provider in the application layer before the authentication response message is sent, so this elevates the channel to provide authentication in both directions, thus establishing a confidential channel from the OpenID provider to the user. The SSL connection from the user to the relying party is confidential because it satisfies $C \wedge NR^{-}$.

In our model of OpenID Authentication we assume that whenever the protocol is run, relying parties either use direct verification or they establish a new association. In fact, relying parties and OpenID providers are allowed to establish associations that they use in more than one run of the proto-

col. Establishing long-term secrets in this manner makes individual protocol runs faster: signatures can be checked by the relying party without any communication between the relying party and the OpenID provider. However, this has the disadvantage that authentication messages can be replayed to the relying party they were originally intended for, and in order to detect replays, relying parties must store all the response nonces they have seen within the lifetime of the association. The protocol is much more secure when a new association is established with every protocol run, and much less effort is required from the relying party to ensure that authentication messages are not replayed.

Even though we did not model the timestamp or the nonce in the authentication response, we did not find any replay attacks because every authentication response message was either verified directly, or by a freshly established association. The security of the protocols in this case, and in particular the guarantee of freshness of the authentication response is provided by the strong symmetric authenticated channel that the relying party and the OpenID provider establish.

7.4.5 Conclusions

All of the attacks that we found against the OpenID Authentication protocols can be prevented by following the strongest recommendations in the specification: users should only use secure URLs as identifiers (either URLs that they control, or local identifiers at OpenID providers); users should always communicate with relying parties and OpenID providers on SSL connections; and relying parties and OpenID providers should perform the association establishment protocol and the direct verification protocol over SSL connections.

Almost all of the security of the OpenID Authentication scheme is due to the protection provided by the secure transport layer protocols that it uses. The only security mechanism in the application layer is the OpenID provider's signature on the authentication response message, which is based entirely on the secrecy of the association. The timestamp and nonce in the authentication response message help to prevent replay attacks, but the protection of a confidential secure transport layer is sufficient to replace these. Replay attacks are not possible if the relying party and the OpenID provider communicate directly in every protocol run. In fact, if a new association is used for every protocol run (and if the user's identity is tied to the association by being included in the association establishment or direct verification protocol), the signature can be removed completely; the only token that needs to be given to the user is the association itself. Clearly the protocol is more efficient if associations can be reused, and in order for this to be possible without introducing attacks, the signature mechanism is necessary.

The OpenID specification [FRH+08] is well written: it is unambiguous,

and easy to follow. The only problem with the specification is that the strongest recommendations mentioned above are not compulsory, they are just suggested. The protocol itself is also simple; rather than using complicated XML documents to carry the protocol messages, the messages are encoded in plain text. However, all field names and types are recorded unambiguously, so the protocols are not susceptible to typing attacks; furthermore, the messages themselves are each given a unique tag (the `mode` field) so messages cannot be confused for one another.

7.5 Developing a new single sign-on protocol

In this section we describe the development process for a new single sign-on protocol. We start the development process by formalising the goals of the protocol, and we decide on a basic message exchange framework (i.e. who sends what to whom in order to achieve the goals). We then determine which channel properties we can use to simplify the protocol; finally, we test our protocol in Casper using the new channel models.

The single sign-on protocol is designed to authenticate a user u to a service provider *sp* by means of a trusted third party: the identity provider *idp*. The goal of the protocol is therefore to achieve this authentication correctly; we recall the definition of *agreement* from [Low97]:

Definition 7.5.1 (Agreement)**.** A protocol guarantees to an initiator A *agreement* with a responder B on a set of data items ds if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in ds, and each such run of A corresponds to a *unique* run of B.

The goal of our single sign-on protocol is therefore to achieve agreement between the user (acting as the protocol initiator) and the service provider (acting as the responder) on the user's request and the user's identity; we test this goal directly in Casper.

We assume that identity providers are trustworthy, and that there is a reliable way for a service provider to determine, or confirm, who a user's identity provider is. For example:

- There may only be one identity provider in the system;
- The user's identity may be a URL on the identity provider's domain (e.g. `http://www.facebook.com/profile.php?id=36801608`);
- The user's identity may be a URL on their own domain, and could identify the identity provider when requested over an HTTPS connection.

Our single sign-on protocol is designed for service-provider first interaction; the user provides his identity to the service provider in order to request a resource, or an action, but before the resource can be delivered, or the action performed, the service provider must ensure that the user is who he claims to be. The phases of the protocol are as follows:

1. The user first visits a service provider and requests a resource, or an action, that is restricted;

2. The service provider generates a new secret (a nonce) and sends it, and the user's identity, to the identity provider. The understanding is that the identity provider should only tell the nonce to the user once he has correctly authenticated him;

3. The service provider redirects the user's browser to his identity provider's website, with a message containing the service provider's identity (so that the identity provider knows which nonce to give the user). The user authenticates himself to his identity provider by some means out of scope of the protocol description (for example, a username and password);

4. When the user has proved who he is, the identity provider gives him the nonce, and redirects him back to the service provider;

5. The user's knowledge of the nonce proves to the service provider that he is who he claimed to be, and so the service provider performs the action, or delivers the resource to the user.

The minimal messages that are required to achieve the message exchange described above are as follows:

$$
\begin{array}{lrcl}
\mathit{Message}\ 1 & u & \rightarrow\ sp & : u, idp, req? \\
\mathit{Message}\ 2.1 & sp & \rightarrow idp & : u, n_{sp} \\
\mathit{Message}\ 2.2 & idp & \rightarrow\ sp & : ok \\
\mathit{Message}\ 3.1 & sp & \rightarrow\ u & : idp, sp \\
\mathit{Message}\ 3.2 & u & \rightarrow idp & : sp \\
\mathit{Message}\ 4.1 & idp & \rightarrow\ u & : sp, u, n_{sp} \\
\mathit{Message}\ 4.2 & u & \rightarrow\ sp & : u, n_{sp} \\
\mathit{Message}\ 5 & sp & \rightarrow\ u & : resp\,.
\end{array}
$$

We examine each message in turn, and consider which channel properties are necessary for the transmission of each message in order to protect the protocol. Messages 1, 2.2 and 3.1 can be sent over the bottom channel. The secrecy of the nonce is essential for the correctness of the protocol, so every time the nonce is transmitted between agents, it must be sent over a confidential channel; messages 2.1, 4.1 and 4.2 must all be sent over channels that satisfy at least $C \wedge NR^{-}$.

If the resource that the user requested is secret, then message 5 must be sent over a confidential channel. On the other hand, if the protocol is being used for credit then message 1 should be sent over a confidential channel (or the request should not be sent until message 4.2). If the protocol is used for responsibility (i.e. if the user has asked the service provider to perform an action), then either the request should be sent in message 4.2, or messages 1 and 4.2 should be sent in an injective session.

We model the authentication of the user to the identity provider by an authenticated channel for message 3.2; however, if the authentication is performed at the application layer, this message (which will contain the user's credentials) should be sent on a confidential channel (to protect the secrecy of the credentials).

Even when the channel properties listed above are used, we have discovered an attack against the authentication of the user to the service provider. During a run of the protocol in which the intruder impersonates an honest user, the intruder can re-ascribe message 2.1 to himself so that, although he cannot learn the nonce directly from overhearing this message, he can cause the identity provider to associate that nonce with a session between the user and the intruder:[5]

$$\begin{array}{lrl}
\alpha.1 & I_u & \rightarrow sp : u, idp, req? \\
\alpha.2.1 & (sp \rightarrow I) & \rightarrow idp : u, n_{sp} \\
\alpha.2.2 & I_{idp} & \rightarrow sp : ok \\
\alpha.3.1 & sp & \rightarrow I_u : idp, sp\,.
\end{array}$$

If that honest user then begins a protocol run with the intruder playing the role of the service provider, the identity provider will tell the user to give the original service provider's nonce to the intruder to prove his identity:

$$\begin{array}{lrl}
\beta.1 & u & \rightarrow I : u, idp, req?' \\
\beta.3.1 & I & \rightarrow u : idp, I \\
\beta.3.2 & u & \rightarrow idp : I \\
\beta.4.1 & idp & \rightarrow u : I, u, n_{sp}\,.
\end{array}$$

Once the intruder learns the nonce, he can present it to the real service provider to impersonate the user:

$$\begin{array}{ll}
 & \beta.4.2 \quad u \rightarrow I : u, n_{sp} \\
\alpha.4.2 \quad I_u \rightarrow sp : u, n_{sp} & \\
\alpha.5 \quad sp \rightarrow I : resp\,. &
\end{array}$$

We discovered this attack by checking an agreement property between the user and the service provider in Casper and FDR using the new channel models described in Chapter 6.

[5]The extra notation for the sender's identity of message $\alpha.2.1$ is intended to show that the intruder re-ascribes this message to himself.

This attack is possible because the intruder can re-ascribe message 2.1 in the protocol. The attack could be prevented by sending message 2.1 over an authenticated channel (increasing the requirement for this channel to $C \wedge NF \wedge NRA \wedge NR$), however it is simpler for the service provider to include his identity in the application-layer message:

$$\textit{Message } 2.1 \quad sp \rightarrow idp : u, sp, n_{sp} \,.$$

With this change made to the protocol, the authentication check in FDR does not detect any more attacks against the authentication of the user to the service provider.

Rather than use an injective session channel to link the request (message 1) with the authenticated channel from the user to the service provider (message 4.2), we delay (or repeat) the request to message 4.2:

$$\textit{Message } 4.2 \quad u \rightarrow sp : u, n_{sp}, req? \,.$$

With the following channel setup, there are no attacks against the authentication of the user to the service provider:

```
2.1 C NR-
3.2 C NR-
4.1 C NR-
4.2 C NR-
```

There are, however, possible attack traces against this protocol in which the intruder can cause a different authentication specification to fail; for example:

- The intruder can impersonate the service provider and the identity provider so that the user believes he has completed a run of the protocol with the service provider. This attack could not lead an honest service provider to authenticate a user incorrectly, but it might lead a user to believe that he will be given credit for his request, or that the action he requested will be performed. This attack is prevented by sending message 5 on an authenticated channel (i.e. $NF \wedge NRA^{-}$), and sending messages 4.2 and 5 on a symmetric session channel.

- As with the OpenID and SAML protocols, a dishonest service provider can choose not to authenticate the user. In this case the attack manifests itself as a trace where the intruder fakes message 4.1 from the identity provider to the user, and does not send message 2.1. This is a protocol run that the identity provider has not taken part in; however, this attack only disadvantages the service provider. This attack can easily be prevented by sending message 4.1 over an authenticated injective session channel.

- The service provider is not authenticated to the identity provider, so the intruder can send fake nonces to the identity provider. The intruder cannot complete a protocol run in which he authenticates himself as an honest user, however he might be able to mount a denial of service attack by filling the identity provider's memory with fake nonces. If message 2.1 is sent on an authenticated injective session channel then the identity provider only stores valid nonces, and this attack can be prevented.

- The intruder can replay a valid message 3.2 to make the identity provider believe the user wants to run the protocol again. The intruder cannot use this attack to authenticate himself as the user, but the attack can easily be prevented by sending message 3.2 on an injective session channel.

We note that the channel setup listed above simulates unilateral TLS (or SSL) connections between the user and the service provider, the service provider and the identity provider, and the user and the identity provider, with application layer authentication on the channel from the user to the identity provider.[6] Further, using SSL channels for every point-to-point connection prevents all of the other attack traces discussed above (though we emphasise that the channel setup above is sufficient to guarantee the main goal of the protocol: authentication of the user to the service provider).

In developing this new single sign-on protocol we found the abstract channel properties very helpful. By explicitly assuming the secure transport layer could provide certain properties to the application layer, we specified a protocol without any encryption, message signing, or hash functions in the application layer. As a result of this, our protocol is simple, and would be straightforward to implement. There are of course several implementation decisions that we have made implicit in our presentation of the protocol (e.g. how long should the identity provider store nonces before forgetting them, and how many nonces should the identity provider remember for each user-service provider pair); however, these decisions can be made at the time of implementation without affecting the security of the protocol's main goal: the authentication of the user to the service provider.

This protocol was developed and tested in less than a day. This incredibly short development and testing process is partly due to the highly abstract formalism of the protocol description, but also largely due to the abstraction of the secure transport layer properties. Because the authentication, confidentiality and session properties are easily understood, we discovered most of the requirements of the secure transport layer without testing the protocol in Casper. The tool support (provided by Casper and FDR) meant

[6]The channel for message 4.1 is confidential because a unilateral SSL channel with application-layer authentication is confidential in both directions.

that we could test different versions of the protocol and different channel setups easily and quickly, and so it was straightforward to prevent the few attacks that were possible.

7.6 Conclusions

In this chapter we have studied two different solutions to the identity federation problem. In the first, the SAML Single Sign-On Web Browser Profile, centralised Identity Providers authenticate their users to Service Providers by giving them signed authentication assertions. In the second solution, the OpenID Authentication scheme, each user chooses an identifier (or a set of identifiers) that uniquely identify him. In order to prove to a relying party that he controls an identifier, the user communicates with an OpenID provider with whom he has some means of proving his identity. The OpenID provider creates a signed authentication message that the user can present to the relying party.

Both of these schemes allow a user to register his details with a single identity provider, and then to prove his identity to several service providers, without having to create new authentication details (such as username and passwords) with each one. Both of these solutions are token authentication protocols: the user is given a token to present to the relying party, and this token has the following properties:

- The token is signed by a particular identity provider; this property ensures that the relying party who receives the token knows the identity of the server who is authenticating the user's identity;
- The token can only be delivered to a particular relying party; this property ensures that the identity provider knows to whom the user wants to authenticate himself;
- The token can only be used once by the specified user at the specified relying party; this property ensures that old tokens cannot be reused;
- The identity provider would only give the token to the user once he had authenticated him, and convinced himself that he really was communicating with the specified user.

Both of these protocols rely on the value of the token being kept secret from everyone but the user, the identity provider and the relying party. If an intruder obtains a token that refers to another user's identity, he can authenticate himself as that user at the relying party for whom the token was originally intended. In order to keep the value of the token secret both of the protocols rely on a secure transport layer; in particular, they both specify that authentication assertions should only be communicated on channels protected by an SSL or TLS connection.

Both sets of specification documents clearly point out the need to keep the authentication tokens secret, and the role of SSL in achieving this confidentiality. However, they do not clearly state the other properties of the transport layer that the protocols rely on. In both cases the symmetric session property of SSL and TLS prevents replay attacks, and the authentication properties prevent attacks in which an intruder can impersonate a relying party or identity provider.

Both sets of specifications fail to specify exactly which channels should be used for communication between every pair of agents. We feel that the specifications would be much clearer, and the protocols more secure, if the specification documents adopted a more formal approach to specifying the properties that the secure transport layer must satisfy in order to make the protocol work correctly. For example, the OpenID protocol requires a confidential channel from the user to the relying party so that the authentication token cannot be overheard; however, this is not specified in [FRH+08].

We also found that both protocols are overly complicated, and, once the properties of the secure transport layer are understood, it is easy to see how they can be simplified. In particular, the signatures on messages can be removed completely if the relying party and identity provider communicate directly and establish a shared secret over a strongly authenticated, confidential channel such as a unilateral (or bilateral) TLS connection.

In Section 7.5 we described a new single sign-on protocol that we designed using the channel specifications from Chapter 3. We designed the protocol to be as concise as possible: all messages only contain the necessary fields, and we specify the minimum channel properties necessary in order for the protocol to achieve its goal securely. There are several design decisions that would need to be made to implement and run this protocol in a web-based environment, but each decision can be supported by reference to the underlying requirements for the channels. In particular, we claim that if all communication in the protocol is run over unilateral TLS (or SSL) connections then the protocol is free from attacks.

The short development process, and the compact description of the protocol are due in part to the highly abstract formalism of the protocol description, but they are also due to the abstraction of the secure transport layer properties. By using the properties from Chapter 3 we simplify the application-layer protocol, and so we simplify the design methodology.

We used the updated tool support to test the first version of the protocol against the goals it was designed to achieve, and we found an attack against the authentication of the user to the service provider, and several attacks against other authentication specifications. It was straightforward to analyse the output from FDR to understand how the attacks were possible, and to modify the channel properties the protocol needs. We updated the formal model, ran the Casper checks again, and confirmed that the changes we made prevented the intruder from performing the attacks.

Chapter 8

Conclusions and future work

In this thesis we have presented a hierarchy of secure channel properties, and we have described a formal framework for reasoning with these properties.

We described a system comprising a set of agents communicating over an insecure network which is controlled by a Dolev-Yao style active intruder. The honest agents can send and receive messages, while the intruder can send, receive, fake and hijack messages. The intruder is constrained only by rules that prevent him from performing impossible events: he can only hijack messages that have already been sent, and he can only send and fake messages that he knows. We defined the set of *ValidSystemTraces*: the set of all possible traces that the honest agents and the intruder can perform.

We specified confidential channels that prevent the intruder from overhearing messages, and we described a necessary condition for a secure transport protocol to establish confidential channels: the intruder must only be able to learn messages that were sent to him, or that were sent on non-confidential channels.

We described several dishonest events that the intruder can perform (e.g. honest re-ascribing), and we presented security specifications that prevent the intruder from performing these events. We investigated the combinations of these specifications and we found that several of them collapse: they allow behaviours that simulate events that they block. We presented five collapsing rules, and, having taken these rules into account, we described a hierarchy of eleven confidential and authenticated channel specifications. We described several of these specifications in more detail, and in every case we presented a simple protocol that satisfies the specification. We have not proved that these protocols satisfy the specifications, however, for these simple examples the proofs appear to be straightforward.

We also investigated channel specifications that group several messages together into a session. We specified injective and symmetric forms of the session property, and then we described even stronger guarantees that secure channels can provide.

We formalised our notion of simulation in terms of the honest agents' views of the valid system traces. By hiding the intruder's events, and only examining the events that he can cause the honest agents to perform, we make the intruder's activity abstract; this allows us to compare channel specifications, even when the model of the intruder is different. We used our simulation relation to define an equivalence relation (mutual simulation), and we used this relation to prove the equivalence of alternative forms of our channel specifications. Rather than blocking the intruder's events, these alternative forms state the possible events that could precede a receive event, and require that one of them does. The alternative specifications are more conducive to proving properties about the secure channel properties.

We showed that every possible combination of the channel primitives (the specifications that block the individual intruder events) is either a point in the channel hierarchy, or collapses to a unique point in the hierarchy. We also showed that we could safely block some combinations of the intruder's events because they simulate other events. We specified a sufficient condition for substituting session channels for stream channels without introducing attacks.

We investigated chaining our secure channel properties through a trusted third party (a proxy). We showed, in two different cases, that our channel properties are invariant under chaining, and that the overall channel property through a proxy is at least as strong as the greatest lower bound of the channels to and from the proxy. In some cases, due to the trust relations between agents and some extra application-layer data, the overall channel is stronger than both channels.

We described abstract CSP models of our channel properties; these models capture the properties of the channels (such as no-redirecting) rather than modelling the concrete transport layer protocols. We built these models into Casper, and we proved that the models are equivalent to the formal properties, even though the model of the intruder is slightly different.

We used the Casper models to study two single sign-on protocols for the internet. These protocols both rely on SSL and TLS connections to function correctly; we modelled the protocols in Casper, and we described several attacks that we found against the protocols. In almost every case the attacks are only possible if the protocol is not implemented exactly as the specification describes it; however, we found a serious attack against the OpenID authentication protocol when users use insecure identities.

Finally, we showed how the channel properties and the channel hierarchy can be used to simplify the protocol development process. We described a new single sign-on protocol which we developed using this technique; by explicitly relying on precisely specified properties of the secure transport layer we justify several simplifications in the application-layer protocol. In particular, our protocol does not use encryption or digital signatures in the application layer.

We hope that the work presented in this thesis will make it easier to specify secure channel properties and to compare these properties with one another. We also hope that this work will make it easier for secure transport layer protocol designers to specify exactly what properties their protocol can provide, and for application-layer security protocol designers to specify exactly what properties their protocol requires of the secure transport layer.

8.1 Future work

In this section we describe several areas for further research, and where we believe there are interesting results to be discovered.

8.1.1 Further channel properties

Our hierarchy of channel properties is far from complete; in this section we describe three directions in which our properties could be extended

Recentness There are several properties that TLS provides that are not captured by our channel specifications; one of these properties is recentness. Because a new TLS session must be established via the full handshake protocol or by resuming an old session, the agents in a TLS session have already verified that each other is 'alive' before they send their first application-layer messages. This has the effect of guaranteeing that each message an agent receives in a TLS connection must have been created recently (i.e. it cannot have been sent before the handshake protocol was performed). Hence recentness is a property provided by TLS.

A guarantee of recentness can also be afforded by adding timestamps to the transport-layer protocol:

$$Message \quad A \rightarrow B : \{m, t\}_{SK(A)} .$$

When B receives this message he knows that A created it at time t, so he knows how fresh that message is (assuming that his clock is synchronised with A's).

We could specify a recentness property by introducing timestamps of a globally synchronised clock to the send, receive, fake and hijack events:

send$.A.c_A.t.B.m$**:** agent A sent message m in connection c_A at time t to agent B;

receive$.B.c_B.t.m$**:** agent B received message m in connection c_B at time t, apparently from agent A;

fake$.A.B.c_B.t.m$**:** the intruder faked message m so that agent B receives it in connection c_B at time t, apparently from A;

hijack$.A \to A'.B \to B'.c_B.t.m$**:** the intruder hijacked a message originally sent by A to B, so that agent B' receives it in connection $c_{B'}$ at time t, apparently from A'.

By introducing these timestamps we derive some additional limitations on the behaviour of the honest agents and the intruder: agents cannot receive messages earlier than they were sent, and the intruder cannot hijack messages so that they are received earlier than they were sent. However, these conditions are implied by our existing network rules and the condition that timestamp values are non-decreasing in all valid system traces. We can now specify a recentness property by requiring that the difference in the timestamps on send and receive events is less than a certain bound.

It is clear that this recentness property is independent of the single-message properties, but it is less clear how it interacts with the stronger stream properties. For example, when an agent receives a message on a mutual stream channel then he knows that the message is at least as recent as the last message he sent.

Weak confidentiality We have seen that a unilateral TLS connection uses the same transport-layer message encoding to carry application-layer messages, but only provides confidentiality in one direction, because only one agent is authenticated to the other. Channels such as this seem to satisfy a weak form of confidentiality: when they are established correctly (i.e. when two honest agents are communicating in a symmetric session) they protect the confidentiality of the application-layer messages. When they are not established correctly (i.e. when the intruder is faking the part of the unauthenticated agent) they do not provide confidentiality, because the messages are not delivered to the agent that the sender intended them for. There is scope for a weaker confidentiality specification to capture this property: if $C^-(R_i \to R_j)$ then only the agent to whom the messages are being delivered can decrypt them.

While this property can be specified independently of the single-message properties, it makes most sense in the context of symmetric session channels (or stronger channels). On these channels an authenticated message sender knows that all his messages are being received by a single agent, and that they are kept secret from all other agents, but he does not necessarily know who the receiving agent is.

Innominate channels In every property in our single-message hierarchy, at least one agent is authenticated to the other. That is, none of our channels are capable of providing properties (such as the session or stream properties, or confidentiality) unless one of the parties is authenticated to the other. It would be interesting to investigate properties that allow two agents to

establish a weakly-confidential channel, say, without either agent having to authenticate himself.

We believe that there is scope for specifying innominate channels such as this. These channels are not strictly anonymous in the sense that an anonymous channel would prevent the recipient of a message from being able to prove who sent the message to him, but they remove the requirement for authentication. It is generally accepted that two agents who do not possess (shared) secrets cannot establish secure communications links, and this is borne out in practice: when TLS is run in 'anonymous authentication' mode using Diffie-Hellman key establishment, there is a man-in-the-middle attack [Res01]. In general, it seems likely that if A and B wish to establish a channel without checking each other's identity, then they do not have any means of detecting whether the intruder has interposed himself between them (and is simply listening to, and relaying messages in each direction).

8.1.2 Classifying secure transport protocols

In Chapter 3 we gave single-message examples of protocols that we believe satisfy the properties in the hierarchy. For example, we claim that encrypting a message with the intended recipient's public key, and including the sender's identity in the encrypted message ($\{A, m\}_{PK(B)}$) achieves $C \wedge NRA \wedge NR^{-}$. We have not proved that these example protocols really do satisfy the properties that we claim they do; it should be straightforward to prove that these protocols are correct.

In Chapter 2 we described several 'real-world' secure transport protocols (such as IPsec and TLS). We can describe which of our channel properties we believe these protocols satisfy, but it would be good to be able to prove that they do indeed satisfy these properties. Kamil and Lowe have made some progress in this direction; in [KL08] they use strand spaces to prove that TLS provides a property that is equivalent to a stream channel that satisfies $C \wedge NF \wedge NRA \wedge NR$.

8.1.3 Proxies and chaining

In Chapter 5 we presented two chaining theorems for secure channels. The theorems are useful because they describe ways in which secure channels might be used, and they allow users of our secure channel specifications to calculate the properties of the overall channel through a proxy very simply. One can easily tell whether or not the chained form of two channels still provides a particular property. In particular, we have shown that the single message channels defined in Chapter 3 are invariant under chaining through a proxy, provided that the proxy is trustworthy.

In Chapter 3 we presented several session properties. A session channel guarantees that all the messages received in a connection were sent in a single

connection. We also specified stream properties which guarantee the session property, and also that messages are received in the same order as that in which they were sent. It would be interesting to investigate whether the session and stream properties are invariant under chaining. It seems likely that this is the case (assuming that whenever the proxy receives several messages in a single session he forwards them in a single session, in the same order).

It would also be interesting to investigate more general chaining; for example, the effect of multiple chains of secure channels. When agents playing role R_i send messages to agents playing role R_j through a multiplexing proxy, the elevation function is different on the channels to and from the proxy. If the chaining is set up as

$$R_i \rightarrow \mathit{Proxy} \rightarrow \mathit{Proxy}' \rightarrow R_j \,,$$

it is not clear what properties the channel through the two different proxies satisfies.

Using the theorems in this paper we could calculate the properties of the overall channel in two different ways: by calculating the resultant channel over the first two connections, then using this result to calculate the result of the overall chain, or by calculating the result of the last two connections first.[1]

$$\begin{array}{l} \downarrow (\nwarrow_m \mathit{Chain}(R_i \rightarrow \mathit{Proxy} \rightarrow \mathit{Proxy}') \sqcap \nearrow_m (\mathit{Proxy}' \rightarrow R_j)) \,, \text{ or} \\ \downarrow (\nwarrow_m (R_i \rightarrow \mathit{Proxy}) \sqcap \nearrow_m \mathit{Chain}(\mathit{Proxy} \rightarrow \mathit{Proxy}' \rightarrow R_j)) \,, \end{array}$$

where:

$$\begin{array}{l} \mathit{Chain}(R_i \rightarrow \mathit{Proxy} \rightarrow \mathit{Proxy}') = \\ \quad \downarrow (\nwarrow_m (R_i \rightarrow \mathit{Proxy}) \sqcap \nearrow_m (\mathit{Proxy} \rightarrow \mathit{Proxy}')) \,, \text{ and} \\ \mathit{Chain}(\mathit{Proxy} \rightarrow \mathit{Proxy}' \rightarrow R_j) = \\ \quad \downarrow (\nwarrow_m (\mathit{Proxy} \rightarrow \mathit{Proxy}') \sqcap \nearrow_m (\mathit{Proxy}' \rightarrow R_j)) \,. \end{array}$$

Because the elevation functions ($\nwarrow_m$ and $\nearrow_m$) are not the same, in most cases these calculations will give different results. For this reason we believe that the overall channel is likely to satisfy the following specification:

$$\downarrow (\nwarrow_m (R_i \rightarrow \mathit{Proxy}) \sqcap (\mathit{Proxy} \rightarrow \mathit{Proxy}') \sqcap \nearrow_m (\mathit{Proxy}' \rightarrow R_j)) \,.$$

The specifications of the channels to the first proxy and from the last proxy are elevated in the usual way, but there is no elevation on the intermediate channel. It is straightforward to generalise this result to longer chains.

[1] In these examples, when we write a channel such as $R_i \rightarrow R_j$, we are referring to the channel properties satisfied by this channel, rather than the channel itself.

8.1.4 Layering

The concept of layering is natural with architectures such as the one described in this thesis. Of course, running an application-layer protocol on top of a transport-layer channel is one instance of layering; however, it is also possible to layer one secure transport protocol on top of another. For example, consider the following four ways of sending message m from agent A to agent B:

$$\begin{array}{ll} \mathit{Message}\,1 & A \rightarrow B : \{m\}_{PK(B)} \\ \mathit{Message}\,2 & A \rightarrow B : \{m\}_{SK(A)} \\ \mathit{Message}\,3 & A \rightarrow B : \{\{m\}_{PK(B)}\}_{SK(A)} \\ \mathit{Message}\,4 & A \rightarrow B : \{\{m\}_{SK(A)}\}_{PK(B)} \end{array}$$

Message 1 satisfies $C \wedge NR^{-}$, and Message 2 satisfies $NF \wedge NRA^{-}$. These channels could be layered in two different ways; firstly, by layering the first channel on top of the second, and secondly, by layering the second channel on top of the first. The order in which the channels are layered is important in determining the properties of the resultant channel:

- The first order (shown as Message 3) results in a channel that satisfies $C \wedge NF \wedge NRA^{-} \wedge NR$;
- The second order (shown as Message 4) results in a channel that satisfies $C \wedge NF \wedge NRA \wedge NR^{-}$.

We saw in Chapter 5 (when we considered the multiplexing proxies) that adding extra information in the application layer can have an effect on the properties of the secure transport layer. For example, if an agent includes his own identity in the application-layer message, then a confidential channel can be elevated to a confidential channel that satisfies $C \wedge NRA \wedge NR^{-}$. On session or stream channels, the agent sending messages only needs to include his identity in one message for the transformation to be applied to every message sent on the channel. There are many other ways that application-layer messages can be used to strengthen the properties of the secure transport layer. For example:

- A could prove knowledge of a long-term secret that is shared with B (such as a password) in order to authenticate himself;
- On a confidential channel, an agent B could choose a nonce and send it to A in the first message he sends; if A includes that nonce in every message he sends to B this could convert a message channel into a session channel.

It would be interesting to use our framework to study formally the problem of layering channels, and to prove that these equivalences are sound.

8.1.5 Group protocols

In this thesis we assumed that the only available cryptographic algorithms are cryptographic hash functions, symmetric key and individual public key encryption, and secret key signing. It would be useful to extend the formalism to consider groups of agents who share secret keys. For example, suppose that a group of agents $\{A_1, \ldots, A_n\}$ know k, a symmetric key, and each agent A_i sends messages to the other agents in the following way: $\{A_i, m\}_k$.

This channel is non-fakeable and non-re-ascribable to agents outside the group; it is also confidential in the sense that it protects the confidentiality of messages from agents outside the group.

This is a very simple example of a protocol based on a group key: there are many more complicated protocols that might be used to establish channels between groups, or within groups. These channels might provide properties that we have not yet specified; for example, a protocol in which agents sign messages with a group secret key will be non-fakeable by anyone outside the group, but not within the group.

8.1.6 Data independence

In Chapter 6 we described how abstract models of the channel properties can be built into Casper in order to test security protocols for secrecy and authentication properties when they are run in a finite system (i.e. a system in which a fixed number of honest agents run the protocol a fixed number of times). The numbers involved in this sort of analysis are generally small (of the order of one or two agents each running the protocol at most twice), because otherwise the space requirements for conducting the state space exploration in FDR become intractable.

Roscoe, Broadfoot and Kleiner have used data independence techniques to simulate the behaviour of an unbounded number of agents using an unbounded supply of fresh variables [RB99, RK06]. These systems allow a finite check in FDR to verify protocols in which the agents can perform an unbounded number of parallel runs.

We would be interested in applying these techniques to the models we described in Chapter 6 in order to obtain much stronger guarantees when performing Casper analyses.

8.1.7 Case studies

In Chapter 7 we presented two case studies of protocols that use SSL and TLS connections. We would be interested to conduct further case studies using our channel models. We are particularly interested in finding more security protocols that use SSL and TLS (e.g. [Vis06]), and in finding security protocols that use different secure transport protocols (such as IPsec). We also believe that there is scope to use our channel models to analyse

some of the protocols introduced by Creese et al. that use empirical channels [CGH+05].

Delegation protocols also provide an interesting area for further extensions to the model. In many delegation protocols security credentials are established in the application layer, and then used in the transport layer. This crossing of layers is not something our current model can represent, as we assume that application-layer messages and transport-layer messages are disjoint. There may also be other classes of security protocol in which data values established in one layer are used in the other, so it would be useful if our model could be extended to enable us to study these.

Bibliography

[AB05] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW 18)*, pages 140–154, 2005.

[Aba98] M. Abadi. Two facets of authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 25–32, 1998.

[ABV+04] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. RFC3748: Extensible authentication protocol (eap). Technical report, Network Working Group, 2004.

[AC08] A. Armando and L. Compagna. SAT-based model checking for security protocols analysis. *International Journal of Information Security*, 7(1), January 2008.

[ACC07] A. Armando, R. Carbone, and L. Compagna. LTL model checking for security protocols. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, 2007.

[ACC+08] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*, 2008.

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[AR02] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[BCK98] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols (extended abstract). *Proceedings of the thirtieth annual ACM symposium on theory of computing*, pages 419–428, 1998.

[BF08] M. Bugliesi and R. Focardi. Language based secure communication. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, 2008.

[BGH+91] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of two-party authentication protocols. *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 44–61, 1991.

[BGW01] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, 2001.

[Bir88] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall International (UK) Ltd. Hertfordshire, UK, 1988.

[BKN04] M. Bellare, T. Kohno, and C. Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the Encode-then-Encrypt-and-MAC paradigm. *ACM Transactions on Information Systems Security*, 7(2):206–241, 2004.

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

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, 2001.

[Bla02] B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes on Computer Science*, pages 342–359, 2002.

[BLP03] G. Bella, C. Longo, and L. Paulson. Verifying second-level security protocols. *Theorem Proving in Higher Order Logics: 16th International Conference, (TPHOLs 2003)*, 2003.

[BLR00] P. Broadfoot, G. Lowe, and A.W. Roscoe. Automating data independence. In *Computer Security - ESORICS 2000, 6th European*

Symposium on Research in Computer Security, volume 1895, pages 175–190, 2000.

[BMP06] G. Bella, F. Massacci, and L. Paulson. Verifying the SET Purchase Protocols. *Journal of Automated Reasoning*, 36(1):5–37, 2006.

[Boy93] C. Boyd. Security architectures using formal methods. *IEEE Journal on Selected Areas in Communications*, 11(5):694–701, 1993.

[BP97] G. Bella and L. Paulson. Using Isabelle to prove properties of the Kerberos authentication system. *Workshop on Design and Formal Verification of Security Protocols (DIMACS)*, 1997.

[BPW06] M. Backes, B. Pfitzmann, and M. Waidner. Formal methods and cryptography. In *Proceedings of 14th International Symposium on Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 612–616, 2006.

[BR93] M. Bellare and P. Rogaway. Entity authentication and key distribution. *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, 773:232–249, 1993.

[BR95] M. Bellare and P. Rogaway. Provably secure session key distribution: the three party case. *Proceedings of the twenty-seventh annual ACM symposium on Theory of Computing*, pages 57–66, 1995.

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[CDL+99] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 55–69, 1999.

[CGH+05] S. Creese, M. Goldsmith, R. Harrison, A.W. Roscoe, P. Whittaker, and I. Zakiuddin. Exploiting empirical engagement in authentication protocol design. *International Conference on Security in Pervasive Computing, Lecture Notes in Computer Science*, 3450:119–133, 2005.

[CJM00] E.M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, 2000.

[CK01] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. *Lecture Notes in Computer Science*, 2045:453–474, 2001.

[CK02] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. *Theory and Application of Cryptographic Techniques*, pages 337–351, 2002.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. *Workshop on Industrial-Strength Formal Specification Techniques*, 95, 1995.

[DA99] T. Dierks and C. Allen. RFC2246: The TLS protocol version 1.0. Technical report, The Internet Engineering Task Force, 1999.

[DDHY92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[Dil08] C. Dilloway. Chaining secure channels. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

[DL07] C. Dilloway and G. Lowe. On the specification of secure channels. In *Proceedings of the 7th International Workshop on Issues in the Theory of Security (WITS'07)*, 2007.

[DL08] C. Dilloway and G. Lowe. Specifying secure channels. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, 2008.

[DOW92] W. Diffie, P. Oorschot, and M. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[DS97] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. *Lecture Notes in Computer Science*, pages 121–136, 1997.

[DS04] R. Delicata and S. Schneider. Towards the rank function verification of protocols that use temporary secrets. In *Proceedings of the Workshop on Issues in the Theory of Security*, 2004.

[DY83] D. Dolev and A.C. Yao. On security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[Fed02] Federal Information Processing Standards. *Announcing the Secure Hash Standard*, 2002.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext transfer protocol — HTTP/1.1. Technical report, The Internet Engineering Task Force, 1999.

[FH07] D. Florencio and C. Herley. A large-scale study of web password habits. *Proceedings of the 16th international conference on World Wide Web*, pages 657–666, 2007.

[FHG99a] F. Fábrega, J. Herzog, and J. Guttman. Mixed strand spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, 1999.

[FHG99b] F.J.T. Fábrega, J.C. Herzog, and J.D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2–3):191–230, 1999.

[FK97] I. Foster and C. Kesselman. Globus: a metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115, 1997.

[FKK96] A. Freier, P. Karlton, and P. Kocher. *The SSL Protocol Version 3.0*, 1996.

[FR06] B. Fitzpatrick and D. Recordon. *OpenID Authentication 1.1*, 2006.

[FRH+08] B. Fitzpatrick, D. Recordon, D. Hardt, J. Bufu, and J. Hoyt. *OpenID Authentication 2.0 - Final*. OpenID Community, 2008. Available from http://openid.net/specs/openid-authentication-2_0.html.

[FSE05] Formal Systems (Europe) Ltd. *FDR2 User Manual*, 2005. Available from http://www.fsel.com/.

[GF00a] J. Guttman and F. Fábrega. Authentication tests. *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[GF00b] J. Guttman and F Fábrega. Protocol independence through disjoint encryption. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 24–34, 2000.

[GF05] J. Guttman and F. Fábrega. The sizes of skeletons: security goals are decidable. Technical Report Technical Report 05B09, MITRE, 2005. Available from http://www.mitre.org/tech/strands/.

[Goo08] Google. *Web-based reference implementation of SAML Single Sign-On for Google Apps*, 2008. Available from http://code.google.com/apis/apps/sso/saml_reference_implementation_web.html.

[Gro03] T. Groß. Security analysis of the SAML Single Sign-On browser/artifact profile. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 298–307, 2003.

[Hea05] J. Heather. 'Oh!... Is it really you?' — Using rank functions to verify authentication protocols. PhD thesis, Department of Computer Science, Royal Holloway, University of London, 2005.

[HL01] M.L. Hui and G. Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1–2):3–46, 2001.

[HLS03] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[Hoa78] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. Hoare. *CSP–Communicating Sequential Processes*. Prenctice Hall, 1985.

[HS05] J. Heather and S. Schneider. A decision procedure for the existence of a rank function. *Journal of Computer Security*, 13(2):317–344, 2005.

[HSN05] S.M. Hansen, J. Skriver, and H.R. Nielson. Using static analysis to validate the SAML Single Sign-On Protocol. In *Proceedings of the Workshop on Issues in the Theory of Security*, 2005.

[Ins81a] Information Science Institute. RFC791: Internet protocol. Technical report, The Internet Engineering Task Force, 1981.

[Ins81b] Information Science Institute. RFC793: Transmission control protocol. Technical report, The Internet Engineering Task Force, 1981.

[Jon02] J. Jonsson. On the security of CTR+ CBC-MAC. In *Proceedings of the 9th Annual Workshop on Selected Areas in Cryptography (SAC 2002)*, volume 2595, pages 76–93, 2002.

[Kau05] C. Kaufman. RFC4306: Internet key exchange (IKEv2) protocol. Technical report, Network Working Group, 2005.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. RFC2104: HMAC: Keyed-hashing for message authentication. Technical report, The Internet Engineering Task Force, 1997.

[KCC01] E. Kwon, Y. Cho, and K. Chae. Integrated transport layer security: End-to-end security model between WTLS and TLS. *Proceedings of the The 15th International Conference on Information Networking*, 2001.

[Ken05a] S. Kent. RFC4302: IP authentication header. Technical report, Network Working Group, 2005.

[Ken05b] S. Kent. RFC4303: IP encapsulating security payload ESP. Technical report, Network Working Group, 2005.

[KL08] A. Kamil and G. Lowe. Analysing TLS in the strand spaces model. Technical report, 2008. Available via http://web.comlab.ox.ac.uk/people/Allaa.Kamil/.

[KR04] E. Kleiner and A.W. Roscoe. Web services security: a preliminary study using casper and FDR. *Proceedings of Automated Reasoning for Security Protocol Analysis (ARSPA 04)*, 2004.

[Kra01] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). *Lecture Notes in Computer Science*, 2139, 2001.

[KS05] S. Kent and K. Seo. RFC4301: Security architecture for the internet protocol. Technical report, Network Working Group, 2005.

[LAN01] LAN/MAN Standards Committee of the IEEE Computer Society. *Port Based Network Access Control*, 2001. Available from http://standards.ieee.org/getieee802/download/802.1X-2001.pdf.

[LAN07] LAN/MAN Standards Committee of the IEEE Computer Society. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*, 2007. Available from http://standards.ieee.org/getieee802/download/802.11-2007.pdf.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, 1055:147–166, 1996.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 31–44, 1997.

[Low98] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1–2):53–84, 1998.

[Low99] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(2–3):89–146, 1999.

[Low04] G. Lowe. Analysing protocols subject to guessing attacks. *Journal of Computer Security*, 12(1):83–97, 2004.

[MAFH02] S. Malladi, J. Alves-Foss, and R. Heckendorn. On preventing replay attacks on security protocols. In *Proceedings of the International Conference on Security and Management*, pages 77–83, 2002.

[Mea92] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1:5–53, 1992.

[Mea94] C. Meadows. The NRL Protocol Analyzer: An overview. In *Proceedings of the 2nd Conference on the Practical Applications of Prolog*, 1994.

[Mea96a] C. Meadows. Analyzing the Needham-Schroeder Public-Key protocol: A comparison of two approaches. *Proceedings of the 4th European Symposium on Research in Computer Security: Computer Security*, pages 351–364, 1996.

[Mea96b] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[Mea99] C. Meadows. Analysis of the Internet Key Exchange protocol using the NRL Protocol Analyzer. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 216–231, 1999.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus.* Cambridge University Press, 1999.

[Mil06] J. Miller. *Yadis Specification Version 1.0.* Yadis, 2006. Available from http://www.yadis.org/.

[MMS97] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[MS94] U. Maurer and P. Schmid. A calculus for secure channel establishment in open networks. In *Computer Security — ESORICS 94*, volume 875 of *Lecture Notes in Computer Science*, pages 175–192. Springer-Verlag, November 1994.

[MSS98] J.C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. *Proceedings of the 7th conference on USENIX Security Symposium*, pages 16–16, 1998.

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

[OAS04] OASIS Security Services Technical Committee. *Technical Overview of the OASIS Security Assertion Markup Language (SAML) V1.1*, 2004. Available from http://www.oasis-open.org/committees/security/.

[OAS05a] OASIS Extensible Resource Identifier (XRI) Technical Committee. *Extensible Resource Identifier (XRI) Syntax V2.0*, 2005. Available from http://www.oasis-open.org/committees/xri/.

[OAS05b] OASIS Security Services Technical Committee. *Assertions and Protocols for the Security Assertion Markup Language (SAML) V2.0*, 2005. Available from http://www.oasis-open.org/committees/security/.

[OAS05c] OASIS Security Services Technical Committee. *Security Assertion Markup Language (SAML) V2.0 Technical Overview*, 2005. Available from http://www.oasis-open.org/committees/security/.

[OAS07] OASIS Security Services Technical Committee. *Assertions and Protocols for the Security Assertion Markup Language (SAML) V2.0 - Errata Composite*, 2007. Available from http://www.oasis-open.org/committees/security/.

[Pau94] L. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, (828), 1994.

[Pau98] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[Pau99] L.C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.

[PB61] W. Peterson and D. Brown. Cyclic codes for error detection. In *Proceedings of the Institute of Radio Engineers*, volume 49, pages 228–235, 1961.

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

[Res01] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[RG97] A.W. Roscoe and M. Goldsmith. The perfect spy for model-checking crypto-protocols. In *Proceedings of DIMACS Workshop on the Design and Formal Verification of Crypto-Protocols*, 1997.

[RK06] A.W. Roscoe and E. Kleiner. Modelling unbounded parallel sessions of security protocols in CSP. Technical report, 2006. Available from http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/.

[RN06] A.W. Roscoe and L. H. Nguyen. Efficient group authentication protocols based on human interaction. In *Proceedings of ARSPA 2006*, 2006.

[RN08] A.W. Roscoe and L. H. Nguyen. Authenticating ad hoc networks by comparison of short digests. *Information and Computation*, 206:250–271, 2008.

[Ros96] A.W. Roscoe. Intensional specifications of security protocols. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 28–38, 1996.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[RSG+01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A.W. Roscoe. *The Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[SBL06] Y. Song, K. Beznosov, and V. Leung. Multiple-channel security architecture and its implementation over SSL. *EURASIP Journal on Wireless Communications and Networking*, 2006(2):78–78, 2006.

[Sch98] S. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.

[Sch02] S. Schneider. Verifying authentication protocol implementations. In *Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 5–24, 2002.

[SD05] S. Schneider and R. Delicata. Verifying security protocols: An application of CSP. *Communicating Sequential Processes: The First 25 Years: Symposium on the Occasion of 25 Years of CSP*, 2005.

[Sim85] G. Simmons. How to (selectively) broadcast a secret. In *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, 1985.

[SIR02] A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin, and Shamir attack to break WEP. *Proceedings of the 2002 Network and Distributed Systems Security Symposium*, 1722, 2002.

[Vis06] Visa International Service Association. *Verified by Visa System Overview External Version 1.0.2*, 2006. Available from https://partnernetwork.visa.com/vpn/global/category.do.

[W3C00] W3C: The World Wide Web Consortium. *Simple Object Access Protocol (SOAP) 1.1*, 2000. Available from http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[Wal00] J. Walker. Unsafe at any key size; an analysis of the WEP encapsulation. *IEEE Document 802.11-00/362*, 2000.

[War05] B. Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. *Journal of Computer Security*, 13(3):565–591, 2005.

[Wi-03] Wi-Fi Alliance. *Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks*, 2003. Available from http://54g.org/pdf/Whitepaper_Wi-Fi_Security4-29-03.pdf.

[WS96] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 4–4, 1996.

[WSF+03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pages 48–57, 2003.

[YL06a] T. Ylonen and C. Lonvick. RFC4251: The Secure Shell (SSH) protocol architecture. Technical report, Network Working Group, 2006.

[YL06b] T. Ylonen and C. Lonvick. RFC4253: The Secure Shell (SSH) transport layer protocol. Technical report, Network Working Group, 2006.

Appendix A

Alternative channel specifications

A.1 Authenticated channels

Definition A.1.1 ($Alt(\bot)$)**.**

$$
\begin{aligned}
&Alt(\bot)(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
&\qquad receive.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad fake.A.B.c_B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \to A.B' \to B.c_B.m \ \mathbf{in}\ tr\,.
\end{aligned}
$$

Definition A.1.2 ($Alt(NF \wedge NRA^-)$)**.**

$$
\begin{aligned}
&Alt(NF \wedge NRA^-)(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
&\qquad receive.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \to A.B' \to B.c_B.m \ \mathbf{in}\ tr \wedge \\
&\qquad\qquad ((A' = A) \vee Dishonest(A))\,.
\end{aligned}
$$

Definition A.1.3 ($Alt(NF \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{aligned}
&Alt(NF \wedge NRA^- \wedge NR^-)(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
&\qquad receive.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \to A.B' \to B.c_B.m \ \mathbf{in}\ tr \wedge \\
&\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B'))\,.
\end{aligned}
$$

Definition A.1.4 ($Alt(NF \wedge NRA^- \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^- \wedge NR)(R_i \rightarrow R_j) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
\qquad receive.B.c_B.A.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i \cdot hijack.A' \rightarrow A.B.c_B.m \textbf{ in } tr \wedge \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \,.
\end{array}
$$

Definition A.1.5 ($Alt(C \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NR^-)(R_i \rightarrow R_j) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
\qquad receive.B.c_B.A.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr \vee \\
\qquad\quad fake.A.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \rightarrow A.B' \rightarrow B.c_B.m \textbf{ in } tr \wedge \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \,.
\end{array}
$$

Definition A.1.6 ($Alt(C \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NRA^- \wedge NR^-)(R_i \rightarrow R_j) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
\qquad receive.B.c_B.A.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr \vee \\
\qquad\quad fake.A.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot hijack.A' \rightarrow A.B' \rightarrow B.c_B.m \textbf{ in } tr \wedge \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \,.
\end{array}
$$

Definition A.1.7 ($Alt(C \wedge NRA \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NRA \wedge NR^-)(R_i \rightarrow R_j) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; m : Message_{App} \cdot \\
\qquad receive.B.c_B.A.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.B.m \textbf{ in } tr \vee \\
\qquad\quad fake.A.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists B' : \hat{R}_j \cdot hijack.A.B' \rightarrow B.c_B.m \textbf{ in } tr \wedge \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \,.
\end{array}
$$

Definition A.1.8 ($Alt(C \wedge NF \wedge NRA^{-} \wedge NR^{-})$)**.**

$$
\begin{aligned}
&Alt(C \wedge NF \wedge NRA^{-} \wedge NR^{-})(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad C(R_i \to R_j) \wedge \\
&\quad \forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{App} \cdot \\
&\qquad \mathit{receive}.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j \cdot \mathit{hijack}.A' \to A.B' \to B.c_B.m \ \mathbf{in}\ tr \wedge \\
&\qquad\qquad ((A' = A) \vee \mathit{Dishonest}(A)) \wedge ((B' = B) \vee \mathit{Dishonest}(B'))\,.
\end{aligned}
$$

Definition A.1.9 ($Alt(C \wedge NF \wedge NRA^{-} \wedge NR)$)**.**

$$
\begin{aligned}
&Alt(C \wedge NF \wedge NRA^{-} \wedge NR)(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad C(R_i \to R_j) \wedge \\
&\quad \forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{App} \cdot \\
&\qquad \mathit{receive}.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad \exists A' : \hat{R}_i \cdot \mathit{hijack}.A' \to A.B.c_B.m \ \mathbf{in}\ tr \wedge \\
&\qquad\qquad ((A' = A) \vee \mathit{Dishonest}(A))\,.
\end{aligned}
$$

Definition A.1.10 ($Alt(C \wedge NF \wedge NRA \wedge NR^{-})$)**.**

$$
\begin{aligned}
&Alt(C \wedge NF \wedge NRA \wedge NR^{-})(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad C(R_i \to R_j) \wedge \\
&\quad \forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{App} \cdot \\
&\qquad \mathit{receive}.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \ \mathbf{in}\ tr \vee \\
&\qquad\quad \exists B' : \hat{R}_j \cdot \mathit{hijack}.A.B' \to B.c_B.m \ \mathbf{in}\ tr \wedge \\
&\qquad\qquad ((B' = B) \vee \mathit{Dishonest}(B'))\,.
\end{aligned}
$$

Definition A.1.11 ($Alt(C \wedge NF \wedge NRA \wedge NR)$)**.**

$$
\begin{aligned}
&Alt(C \wedge NF \wedge NRA \wedge NR)(R_i \to R_j) \mathrel{\widehat{=}} \\
&\quad C(R_i \to R_j) \wedge \\
&\quad \forall B : \hat{R}_j; c_B : \mathit{Connection}; A : \hat{R}_i; m : \mathit{Message}_{App} \cdot \\
&\qquad \mathit{receive}.B.c_B.A.m \ \mathbf{in}\ tr \Rightarrow \\
&\qquad\quad \exists c_A : \mathit{Connection} \cdot \mathit{send}.A.c_A.B.m \ \mathbf{in}\ tr\,.
\end{aligned}
$$

A.2 Simple proxy channels

Definition A.2.1 ($Alt(\bot)$)**.**

$$
\begin{array}{l}
Alt(\bot)(Proxy_{(R_i,R_j)}^{\longrightarrow})(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\quad fake.P_{(A,B)}.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad hijack.A' \rightarrow A.P_{(A',B')} \rightarrow P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad hijack.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.2 ($Alt(NF \wedge NRA^-)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^-)(Proxy_{(R_i,R_j)}^{\longrightarrow})(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P_{(A',B')} \rightarrow P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.3 ($Alt(NF \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^- \wedge NR^-)(Proxy_{(R_i,R_j)}^{\longrightarrow})(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P_{(A',B')} \rightarrow P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B')} \rightarrow P_{(A,B)}.B' \rightarrow B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.4 ($Alt(NF \wedge NRA^- \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^- \wedge NR)(Proxy_{(R_i,R_j)}^{\longrightarrow})(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; P_{(A',B)} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P_{(A',B)} \rightarrow P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B)} \rightarrow P_{(A,B)}.B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.5 ($Alt(C \wedge NR^{-})$)**.**

$$
\begin{array}{l}
Alt(C \wedge NR^{-})(\overrightarrow{Proxy_{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\quad fake.P_{(A,B)}.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \to A.P_{(A',B')} \to P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.6 ($Alt(C \wedge NRA^{-} \wedge NR^{-})$)**.**

$$
\begin{array}{l}
Alt(C \wedge NRA^{-} \wedge NR^{-})(\overrightarrow{Proxy_{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\quad fake.P_{(A,B)}.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \to A.P_{(A',B')} \to P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.7 ($Alt(C \wedge NRA \wedge NR^{-})$)**.**

$$
\begin{array}{l}
Alt(C \wedge NRA \wedge NR^{-})(\overrightarrow{Proxy_{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\quad fake.P_{(A,B)}.B.c_B.m \textbf{ in } tr \vee \\
\qquad\quad \exists B' : \hat{R}_j; P_{(A,B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A.P_{(A,B')} \to P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A,B')} \to P_{(A,B)}.B' \to B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.8 ($Alt(C \wedge NF \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA^- \wedge NR^-)(Proxy_{\overrightarrow{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; P_{(A',B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \to A.P_{(A',B')} \to P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B')} \to P_{(A,B)}.B' \to B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.9 ($Alt(C \wedge NF \wedge NRA^- \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA^- \wedge NR)(Proxy_{\overrightarrow{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i P_{(A',B)} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge \\
\qquad\qquad\quad hijack.A' \to A.P_{(A',B)} \to P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A',B)} \to P_{(A,B)}.B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.10 ($Alt(C \wedge NF \wedge NRA \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA \wedge NR^-)(Proxy_{\overrightarrow{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr \vee \\
\qquad\quad \exists B' : \hat{R}_j; P_{(A,B')} : \widehat{Proxy}_{(R_i,R_j)}; c_P : Connection \cdot \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A.P_{(A,B')} \to P_{(A,B)}.c_P.m \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P_{(A,B')} \to P_{(A,B)}.B' \to B.c_B.m \textbf{ in } tr\,.
\end{array}
$$

Definition A.2.11 ($Alt(C \wedge NF \wedge NRA \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA \wedge NR)(Proxy_{\overrightarrow{(R_i,R_j)}})(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy_{(R_i,R_j)}) \wedge C(Proxy_{(R_i,R_j)} \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P_{(A,B)} : \widehat{Proxy}_{(R_i,R_j)}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P_{(A,B)}.m \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P_{(A,B)}.m \textbf{ in } tr\,.
\end{array}
$$

A.3 Multiplexing proxy channels

Definition A.3.1 ($Alt(\bot)$)**.**

$$
\begin{array}{l}
Alt(\bot)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad fake.P.B.c_B.\langle A, m\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad hijack.A' \rightarrow A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\qquad hijack.P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.2 ($Alt(NF \wedge NRA^-)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.3 ($Alt(NF \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^- \wedge NR^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.4 ($Alt(NF \wedge NRA^- \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(NF \wedge NRA^- \wedge NR)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P.c_P.\langle m, B\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.5 ($Alt(C \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NR^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow Proxy) \wedge C(Proxy \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad fake.P.B.c_B.\langle A, m\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.6 ($Alt(C \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NRA^- \wedge NR^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow Proxy) \wedge C(Proxy \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad fake.P.B.c_B.\langle A, m\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \rightarrow A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.7 ($Alt(C \wedge NRA \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NRA \wedge NR^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \rightarrow Proxy) \wedge C(Proxy \rightarrow R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists c_P : Connection \cdot fake.A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad fake.P.B.c_B.\langle A, m\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.P.B' \rightarrow B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.8 ($Alt(C \wedge NF \wedge NRA^- \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA^- \wedge NR^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy) \wedge C(Proxy \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.A' \to A.P.c_P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\qquad\quad hijack.P.B' \to B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.9 ($Alt(C \wedge NF \wedge NRA^- \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA^- \wedge NR)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy) \wedge C(Proxy \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists A' : \hat{R}_i; c_P : Connection \cdot \\
\qquad\qquad ((A' = A) \vee Dishonest(A)) \wedge \\
\qquad\qquad\quad hijack.A' \to A.P.c_P.\langle m, B\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.10 ($Alt(C \wedge NF \wedge NRA \wedge NR^-)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA \wedge NR^-)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy) \wedge C(Proxy \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr \vee \\
\qquad\quad \exists B' : \hat{R}_j; c_P : Connection \cdot \\
\qquad\qquad ((B' = B) \vee Dishonest(B')) \wedge \\
\qquad\qquad\quad hijack.P.B' \to B.c_B.\langle A, m\rangle \textbf{ in } tr\,.
\end{array}
$$

Definition A.3.11 ($Alt(C \wedge NF \wedge NRA \wedge NR)$)**.**

$$
\begin{array}{l}
Alt(C \wedge NF \wedge NRA \wedge NR)(Proxy(R_i, R_j))(tr) \mathrel{\widehat{=}} \\
\quad C(R_i \to Proxy) \wedge C(Proxy \to R_j) \wedge \\
\quad \forall B : \hat{R}_j; c_B : Connection; A : \hat{R}_i; P : \widehat{Proxy}; m : Message_{App} \cdot \\
\qquad receive.B.c_B.P.\langle A, m\rangle \textbf{ in } tr \Rightarrow \\
\qquad\quad \exists c_A : Connection \cdot send.A.c_A.P.\langle m, B\rangle \textbf{ in } tr\,.
\end{array}
$$

Appendix B

Additional material for chaining theorems

B.1 Simple Proxy results

See Figure B.1.

B.2 Multiplexing Proxy results

See Figure B.2.

Channel to proxy				Channel from proxy				Resultant channel			
C	NF	NRA	NR	C	NF	NRA	NR	C	NF	NRA	NR
C	NF	NRA	NR	C	NF	NRA	NR^-	C	NF	NRA	NR
C	NF	NRA	NR	C	NF	NRA^-	NR	C	NF	NRA^-	NR
C	NF	NRA	NR	C		NRA	NR^-	C		NRA	NR^-
C	NF	NRA	NR	C	NF	NRA^-	NR^-	C	NF	NRA^-	NR
C	NF	NRA	NR		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA	NR	C		NRA^-	NR^-	C		NRA^-	NR^-
C	NF	NRA	NR		NF	NRA^-	NR^-		NF	NRA^-	NR
C	NF	NRA	NR	C			NR^-	C			NR^-
C	NF	NRA	NR		NF	NRA^-			NF	NRA^-	NR
C	NF	NRA	NR			$\perp$				$\perp$	
C	NF	NRA	NR^-	C	NF	NRA	NR	C	NF	NRA	NR
C	NF	NRA	NR^-	C	NF	NRA	NR^-	C	NF	NRA	NR
C	NF	NRA	NR^-	C	NF	NRA^-	NR	C	NF	NRA^-	NR
C	NF	NRA	NR^-	C		NRA	NR^-	C		NRA	NR^-
C	NF	NRA	NR^-	C	NF	NRA^-	NR^-	C	NF	NRA^-	NR
C	NF	NRA	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA	NR^-	C		NRA^-	NR^-	C		NRA^-	NR^-
C	NF	NRA	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR
C	NF	NRA	NR^-	C			NR^-	C			NR^-
C	NF	NRA	NR^-		NF	NRA^-			NF	NRA^-	NR
C	NF	NRA	NR^-			$\perp$				$\perp$	
C	NF	NRA^-	NR	C	NF	NRA	NR	C	NF	NRA	NR
C	NF	NRA^-	NR	C	NF	NRA	NR^-	C	NF	NRA	NR
C	NF	NRA^-	NR	C	NF	NRA^-	NR	C	NF	NRA^-	NR
C	NF	NRA^-	NR	C		NRA	NR^-	C		NRA	NR^-
C	NF	NRA^-	NR	C	NF	NRA^-	NR^-	C	NF	NRA^-	NR
C	NF	NRA^-	NR		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA^-	NR	C		NRA^-	NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA^-	NR			$\perp$				$\perp$	
C		NRA	NR^-	C	NF	NRA	NR	C		NRA	NR^-
C		NRA	NR^-	C	NF	NRA	NR^-	C		NRA	NR^-
C		NRA	NR^-	C	NF	NRA^-	NR	C		NRA^-	NR^-
C		NRA	NR^-	C		NRA	NR^-	C		NRA	NR^-
C		NRA	NR^-	C	NF	NRA^-	NR^-	C		NRA^-	NR^-
C		NRA	NR^-		NF	NRA^-	NR			$\perp$	
C		NRA	NR^-	C		NRA^-	NR^-	C		NRA^-	NR^-
C		NRA	NR^-		NF	NRA^-	NR^-			$\perp$	

Channel to proxy				Channel from proxy				Resultant channel			
C		NRA	NR^-	C			NR^-	C			NR^-
C		NRA	NR^-		NF	NRA^-				$\perp$	
C		NRA	NR^-			$\perp$				$\perp$	
C	NF	NRA^-	NR^-	C	NF	NRA	NR	C	NF	NRA	NR
C	NF	NRA^-	NR^-	C	NF	NRA	NR^-	C	NF	NRA	NR
C	NF	NRA^-	NR^-	C	NF	NRA^-	NR	C	NF	NRA^-	NR
C	NF	NRA^-	NR^-	C		NRA	NR^-	C		NRA	NR^-
C	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-	C	NF	NRA^-	NR
C	NF	NRA^-	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA^-	NR^-	C		NRA^-	NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR
C	NF	NRA^-	NR^-	C			NR^-	C			NR^-
C	NF	NRA^-	NR^-		NF	NRA^-			NF	NRA^-	NR
C	NF	NRA^-	NR^-			$\perp$				$\perp$	
	NF	NRA^-	NR^-	C	NF	NRA	NR		NF	NRA^-	NR
	NF	NRA^-	NR	C	NF	NRA	NR^-		NF	NRA^-	NR
	NF	NRA^-	NR	C	NF	NRA	NR		NF	NRA^-	NR
	NF	NRA^-	NR	C	NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR	C	NF	NRA^-	NR^-		NF	NRA^-	NR
	NF	NRA^-	NR		NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR	C		NRA^-	NR^-		NF	NRA^-	NR
	NF	NRA^-	NR		NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR	C			NR^-				
	NF	NRA^-	NR		NF	NRA^-			NF	NRA^-	NR
	NF	NRA^-	NR			$\perp$				$\perp$	
	NF	NRA^-	NR	C		NRA^-	NR^-		NF	NRA^-	NR
	NF	NRA^-	NR		NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR	C		NRA	NR^-			$\perp$	

Channel to proxy				Channel from proxy				Resultant channel			
	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-		NF	NRA^-	NR
	NF	NRA^-	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR^-	C		NRA^-	NR^-			$\perp$	
	NF	NRA^-	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR
	NF	NRA^-	NR^-	C			NR^-			$\perp$	
	NF	NRA^-	NR^-			$\perp$				$\perp$	
C			NR^-	C	NF	NRA	NR	C	NRA	NR^-	
C			NR^-	C	NF	NRA	NR^-	C	NRA	NR^-	
C			NR^-	C	NF	NRA^-	NR	C	NRA^-	NR^-	
C			NR^-	C		NRA	NR^-	C	NRA	NR^-	
C			NR^-	C	NF	NRA^-	NR^-	C	NRA^-	NR^-	
C			NR^-		NF	NRA^-	NR		$\perp$		
C			NR^-	C		NRA^-	NR^-	C	NRA^-	NR^-	
C			NR^-		NF	NRA^-	NR^-		$\perp$		
C			NR^-	C			NR^-	C			NR^-
C			NR^-		NF	NRA^-			$\perp$		
C			NR^-			$\perp$			$\perp$		
	NF	NRA^-		C	NF	NRA	NR		NF	NRA^-	
	NF	NRA^-		C	NF	NRA	NR^-		NF	NRA^-	
	NF	NRA^-		C	NF	NRA^-	NR		NF	NRA^-	
	NF	NRA^-		C		NRA	NR^-		NF	NRA^-	
	NF	NRA^-		C	NF	NRA^-	NR^-		NF	NRA^-	
	NF	NRA^-			NF	NRA^-	NR		NF	NRA^-	
	NF	NRA^-		C			NR^-		NF	NRA^-	
	NF	NRA^-			NF	NRA^-			NF	NRA^-	
	NF	NRA^-				$\perp$				$\perp$	
		$\perp$		C	NF	NRA	NR			$\perp$	
		$\perp$		C	NF	NRA	NR^-			$\perp$	
		$\perp$		C	NF	NRA^-	NR			$\perp$	
		$\perp$		C		NRA^-	NR^-			$\perp$	
		$\perp$		C	NF	NRA^-	NR^-			$\perp$	
		$\perp$			NF	NRA^-	NR			$\perp$	
		$\perp$		C		NRA^-	NR^-			$\perp$	
		$\perp$			NF	NRA^-	NR^-			$\perp$	
		$\perp$		C			NR^-			$\perp$	
		$\perp$			NF	NRA^-				$\perp$	
		$\perp$				$\perp$				$\perp$	

Figure B.1: The table of resultant channels through a simple proxy.

Channel to proxy				Channel from proxy				Resultant channel			
C	NF	NRA	NR	C	NF	NRA	NR	C	NF	NRA	NR
C	NF	NRA	NR	C	NF	NRA	NR^-	C	NF	NRA	NR^-
C	NF	NRA	NR	C	NF	NRA^-	NR	C	NF	NRA	NR
C	NF	NRA	NR	C		NRA	NR^-	C		NRA	NR^-
C	NF	NRA	NR	C	NF	NRA^-	NR^-	C	NF	NRA	NR^-
C	NF	NRA	NR		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA	NR	C		NRA^-	NR^-	C		NRA	NR^-
C	NF	NRA	NR		NF	NRA^-	NR^-		NF	NRA^-	NR^-
C	NF	NRA	NR	C			NR^-	C		NRA	NR^-
C	NF	NRA	NR		NF	NRA^-			NF	NRA^-	
C	NF	NRA	NR			$\perp$				$\perp$	
C	NF	NRA	NR^-	C	NF	NRA	NR	C	NF	NRA	NR
C	NF	NRA	NR^-	C	NF	NRA	NR^-	C	NF	NRA	NR^-
C	NF	NRA	NR^-	C	NF	NRA^-	NR	C	NF	NRA	NR
C	NF	NRA	NR^-	C		NRA	NR^-	C		NRA	NR^-
C	NF	NRA	NR^-	C	NF	NRA^-	NR^-	C	NF	NRA	NR^-
C	NF	NRA	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA	NR^-	C		NRA^-	NR^-	C		NRA	NR^-
C	NF	NRA	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR^-
C	NF	NRA	NR^-	C			NR^-	C		NRA	NR^-
C	NF	NRA	NR^-		NF	NRA^-			NF	NRA^-	
C	NF	NRA	NR^-			$\perp$				$\perp$	
C	NF	NRA^-	NR	C	NF	NRA	NR	C	NF	NRA^-	NR
C	NF	NRA^-	NR	C	NF	NRA	NR^-	C	NF	NRA^-	NR^-
C	NF	NRA^-	NR	C	NF	NRA^-	NR	C	NF	NRA^-	NR
C	NF	NRA^-	NR	C		NRA	NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR	C	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-
C	NF	NRA^-	NR		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA^-	NR	C		NRA^-	NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR		NF	NRA^-	NR^-		NF	NRA^-	NR^-
C	NF	NRA^-	NR	C			NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR		NF	NRA^-			NF	NRA^-	
C	NF	NRA^-	NR			$\perp$				$\perp$	
C		NRA	NR^-	C	NF	NRA	NR	C		NRA	NR^-
C		NRA	NR^-	C	NF	NRA	NR^-	C		NRA	NR^-
C		NRA	NR^-	C	NF	NRA^-	NR	C		NRA	NR^-
C		NRA	NR^-	C		NRA	NR^-	C		NRA	NR^-
C		NRA	NR^-	C	NF	NRA^-	NR^-	C		NRA	NR^-
C		NRA	NR^-		NF	NRA^-	NR			$\perp$	
C		NRA	NR^-	C		NRA^-	NR^-	C		NRA	NR^-
C		NRA	NR^-		NF	NRA^-	NR^-			$\perp$	

Channel to proxy				Channel from proxy				Resultant channel			
C		NRA	NR^-	C			NR^-	C		NRA	NR^-
C		NRA	NR^-		NF	NRA^-				$\perp$	
C		NRA	NR^-			$\perp$				$\perp$	
C	NF	NRA^-	NR^-	C	NF	NRA	NR	C	NF	NRA^-	NR
C	NF	NRA^-	NR^-	C	NF	NRA	NR^-	C	NF	NRA^-	NR^-
C	NF	NRA^-	NR^-	C	NF	NRA^-	NR	C	NF	NRA^-	NR
C	NF	NRA^-	NR^-	C		NRA	NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-
C	NF	NRA^-	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
C	NF	NRA^-	NR^-	C		NRA^-	NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR^-
C	NF	NRA^-	NR^-	C			NR^-	C		NRA^-	NR^-
C	NF	NRA^-	NR^-		NF	NRA^-			NF	NRA^-	
C	NF	NRA^-	NR^-			$\perp$				$\perp$	
C		NRA^-	NR^-	C	NF	NRA	NR	C		NRA^-	NR^-
C		NRA^-	NR^-	C	NF	NRA	NR^-	C		NRA^-	NR^-
C		NRA^-	NR^-	C	NF	NRA^-	NR	C		NRA^-	NR^-
C		NRA^-	NR^-	C		NRA^-	NR^-	C		NRA^-	NR^-
C		NRA^-	NR^-	C	NF	NRA^-	NR^-	C		NRA^-	NR^-
C		NRA^-	NR^-		NF	NRA^-	NR			$\perp$	
C		NRA^-	NR^-	C		NRA^-	NR^-	C		NRA^-	NR^-
C		NRA^-	NR^-		NF	NRA^-				$\perp$	
C		NRA^-	NR^-			$\perp$				$\perp$	
	NF	NRA^-	NR^-	C	NF	NRA	NR		NF	NRA^-	NR
	NF	NRA^-	NR^-	C	NF	NRA	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-	C	NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR^-	C		NRA	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR^-	C		NRA^-	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-	C			NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-		NF	NRA^-			NF	NRA^-	
	NF	NRA^-	NR^-			$\perp$				$\perp$	
		$\perp$		C	NF	NRA	NR			$\perp$	
		$\perp$		C	NF	NRA	NR^-			$\perp$	
		$\perp$		C	NF	NRA^-	NR			$\perp$	
		$\perp$		C		NRA	NR^-			$\perp$	
		$\perp$		C	NF	NRA^-	NR^-			$\perp$	
		$\perp$			NF	NRA^-	NR			$\perp$	
		$\perp$		C		NRA^-	NR^-			$\perp$	
		$\perp$			NF	NRA^-	NR^-			$\perp$	
		$\perp$		C			NR^-			$\perp$	
		$\perp$			NF	NRA^-				$\perp$	
		$\perp$				$\perp$				$\perp$	

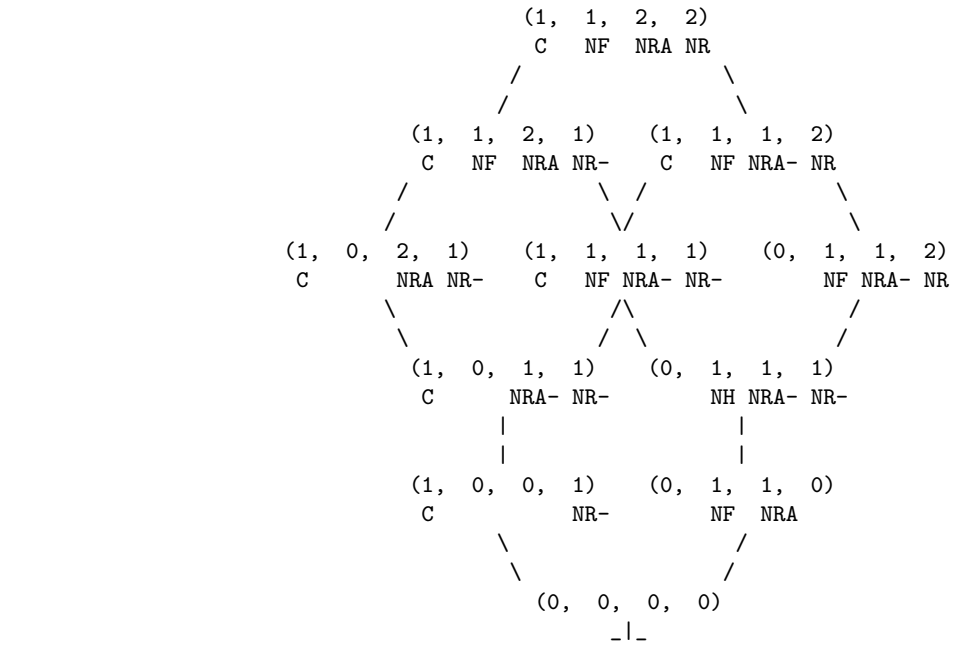
Channel to proxy				Channel from proxy				Resultant channel			
	NF	NRA^-	NR^-	C	NF	NRA^-	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-		NF	NRA^-	NR		NF	NRA^-	NR
	NF	NRA^-	NR^-	C		NRA^-	NR^-			$\perp$	
	NF	NRA^-	NR^-		NF	NRA^-	NR^-		NF	NRA^-	NR^-
	NF	NRA^-	NR^-	C		NRA^-	NR^-			$\perp$	
	NF	NRA^-	NR^-		NF	NRA^-			NF	NRA^-	
	NF	NRA^-	NR^-			$\perp$				$\perp$	
		$\perp$		C	NF	NRA	NR			$\perp$	
		$\perp$		C	NF	NRA	NR^-			$\perp$	
		$\perp$		C	NF	NRA^-	NR			$\perp$	
		$\perp$		C		NRA	NR^-			$\perp$	
		$\perp$		C	NF	NRA^-	NR^-			$\perp$	
		$\perp$			NF	NRA^-	NR			$\perp$	
		$\perp$		C		NRA^-	NR^-			$\perp$	
		$\perp$			NF	NRA^-	NR^-			$\perp$	
		$\perp$		C			NR^-			$\perp$	
		$\perp$			NF	NRA^-				$\perp$	
		$\perp$				$\perp$				$\perp$	

Figure B.2: The table of resultant channels through a multiplexing proxy.

B.3 Subsidiary (shared) channel functions

channels.lhs

```
>module Channels (
>  Channel, ChannelComp (Less, Greater, Equal, Incomparable),
>  channelCompare, hierarchy, collapse, channelShow, glb)
>where

A channel is represented by its co-ordinates in the full lattice:
(C, NF, NRA, NR)

>type Channel = (Integer, Integer, Integer, Integer)

>data ChannelComp = Less | Equal | Greater | Incomparable
>                  deriving (Eq, Show)

Channels are compared pointwise:

>channelCompare :: Channel -> Channel -> ChannelComp
>channelCompare (c1, nf1, nra1, nr1) (c2, nf2, nra2, nr2)
> | c1 == c2 && nf1 == nf2 && nra1 == nra2 && nr1 == nr2 = Equal
> | c1 <= c2 && nf1 <= nf2 && nra1 <= nra2 && nr1 <= nr2 = Less
> | c1 >= c2 && nf1 >= nf2 && nra1 >= nra2 && nr1 >= nr2 = Greater
> | otherwise = Incomparable

The hierarchy is represented as follows:
```




```
>hierarchy :: [Channel]
>hierarchy = [(1, 1, 2, 2),
>(1, 1, 2, 1), (1, 1, 1, 2),
```

```
>(1, 0, 2, 1), (1, 1, 1, 1), (0, 1, 1, 2),
>(1, 0, 1, 1), (0, 1, 1, 1),
>(1, 0, 0, 1), (0, 1, 1, 0),
>(0, 0, 0, 0)]

Collapse a point in the lattice by applying the collapsing rules until we
reach a fixpoint

>collapse :: Channel -> Channel
>collapse c = if (c == c') then c else collapse c'
>  where c' = collapse' c

>collapse' :: Channel  -> Channel
>collapse' (0,0,x,y) = (0,0,0,0)
>collapse' (x,1,0,y) = (x,0,0,y)
>collapse' (0,1,2,x) = (0,1,1,x)
>collapse' (1,x,y,0) = (0,x,y,0)
>collapse' (1,0,x,2) = (1,0,x,1)
>collapse' c = c

Show a channel as a string

>channelShow :: Channel -> String
>channelShow (0,0,0,0) = "_|_"
>channelShow (c, nf, nra, nr) = cStr ++ nfStr ++ nraStr ++ nrStr
>  where cStr    = if c > 0 then "C" ++
>                  (if nf + nra + nr > 0 then " ^ " else "") else ""
>        nfStr   = if nf > 0 then "NF" ++
>  (if nra + nr > 0 then " ^ " else "") else ""
>        nraStr  = if nra > 0 then dashNRA ++
>  (if nr > 0 then " ^ " else "") else ""
>        nrStr   = if nr > 0 then dashNR else ""
>        dashNRA = if nra == 1 then "NRA-" else "NRA"
>        dashNR  = if nr == 1 then "NR-" else "NR"

The greatest lower bound (in the hierarchy) of two channels

>glb :: Channel -> Channel -> Channel
>glb (c1, nf1, nra1, nr1) (c2, nf2, nra2, nr2) =
>  (min c1 c2, min nf1 nf2, min nra1 nra2, min nr1 nr2)
```

B.4 Simple proxy proof script

```
>import Channels

We calculate the channel combinations where the expected result (calculated
using the elevation rules) and the actual result (calculated using the trace
patterns) differ.

>difference :: IO()
>difference = (putStr . concat) [
>  "Actual: " ++ (actual c1 c2) ++
>  "Expected: " ++ (expected c1 c2) ++ "\n" |
```

```
>   c1 <- hierarchy, c2 <- hierarchy,
>   (actual c1 c2) /= (expected c1 c2)]
```

The expected result (calculated using the elevation rules).

```
>expected :: Channel -> Channel -> String
>expected c1 c2 = "A --[" ++
>  (channelShow c1) ++ "]--> Proxy --[" ++
>  (channelShow c2) ++ "]--> B = " ++
>  (channelShow resultant) ++ "\n"
>  where resultant = collapse (glb (raiseToProxy c1) (raiseFromProxy c2))

>raiseToProxy :: Channel -> Channel
>raiseToProxy (c, nf, nra, nr) = (c, nf, nra', nr')
>  where nra' = if (nr > 0) then 2 else nra
>        nr'  = if (nr > 0) then 2 else 0

>raiseFromProxy :: Channel -> Channel
>raiseFromProxy (c, nf, nra, nr) = (c, nf, nra, nr')
>  where nr' = if (nra > 0) then 2 else nr
```

The actual result (calculated using trace patterns).

```
>actual :: Channel -> Channel -> String
>actual c1 c2 = "A --[" ++
>  (channelShow c1) ++ "]--> Proxy --[" ++
>  (channelShow c2) ++ "]--> B = " ++
>  (channelShow (match (resultant A) (resultant I) c1 c2)) ++ "\n"
>  where resultant a = (map head . toSender a . toProxy) (finalEvent a)
>        toSender a = applyChannel c1 . concat . map
>                     (initial (Sender a) (Receiver B))
>        toProxy = map preProxy . applyChannel c2 . pre
>        finalEvent a = (Receive (Proxy (Sender a, Receiver B), Receiver B))
```

We use representative identities: A, A', B, B' are honest; I is dishonest.

```
>data Identity = A | A' | B | B' | I
>  deriving (Eq, Show)
```

An agent is either a proxy between two agents, a sender, or a receiver.

```
>data Agent = Proxy (Agent, Agent) | Sender Identity | Receiver Identity
>  deriving (Eq, Show)
```

Proxies are honest if the agent they send on behalf of is honest.

```
>honest :: Agent -> Bool
>honest (Proxy (a,b)) = honest a
>honest (Sender a) = a /= I
>honest (Receiver a) = a /= I
```

It's convenient to list the senders, receivers and proxies.

```
>senders :: [Agent]
```

```
>senders = [Sender A, Sender A', Sender I]
>receivers :: [Agent]
>receivers = [Receiver B, Receiver B', Receiver I]
>proxies :: [Agent]
>proxies = [Proxy (a, b) | a <- senders, b <- receivers]
```

And to pick out the sender or receiver from a proxy.

```
>sender :: Agent -> Agent
>sender (Proxy (a, b)) = a
>receiver :: Agent -> Agent
>receiver (Proxy (a, b)) = b
```

Each event is either a send, receive, fake or hijack. We list the sender's identity first, the recipient's second.

```
>data Event = Send (Agent, Agent) |
>            Receive (Agent, Agent) |
>            Fake (Agent, Agent) |
>            Hijack (Agent, Agent, Agent, Agent)
>  deriving (Eq,Show)
```

The function pre calculates which events could have occurred immediately before the final receive event. However, we don't let the intruder fake with his own (e.g. a dishonest) identity.

```
>pre :: Event -> [[Event]]
>pre (Receive (p,b)) = sends:fakes:hijacks
>  where sends = [Send (p,b), Receive (p,b)]
>        fakes = if (honest (p)) then [Fake (p,b), Receive (p,b)]
>                      else []
>        hijacks = [[Send (p', receiver (p')),
>                          Hijack (p', p, receiver (p'), b),
>                          Receive (p,b)] | p' <- proxies]
```

If the proxy p sent a message to b, then who sent the message to p?

```
>preProxy :: [Event] -> [Event]
>preProxy (Send (p,b):xs) = Receive (sender p, p):Send (p,b):xs
>preProxy xs = xs
```

Once we know who the proxy received the message from we can work out all possible traces that would result in our original event. However:
We don't let the intruder send messages to the wrong recipient;
We don't let the intruder fake with his own identity, with the wrong sender's identity, or to the wrong recipient.

```
>initial :: Agent -> Agent -> [Event] -> [[Event]]
>initial a1 b1 (Receive (a,p):xs) = sends:fakes:hijacks
>  where sends = if (honest (a) || receiver (p) == b1) then
>                Send (a,p):Receive (a,p):xs else []
>        fakes = if (honest (a) && a == a1 && receiver(p) == b1) then
>                Fake (a,p):Receive (a,p):xs else []
>        hijacks = [[Send ((sender (p')), p'),
```

```
>                   Hijack (sender (p'),a,p',p),
>                   Receive (a,p)] ++ xs | p' <- proxies]
>initial _ _ xs = [xs]
```

We can apply a channel to the events each side of the proxy (apply only looks at the first two events in each trace).

```
>applyChannel :: Channel -> [[Event]] -> [[Event]]
>applyChannel _ [] = []
>applyChannel cs ([]:xss) = applyChannel cs xss
>applyChannel (c,nf,nra,nr) ((x:y:xs):xss) =
>  if (nfs nf (x,y) && nras nra (x,y) && nrs nr (x,y)) then
>  (x:y:xs):(applyChannel (c,nf,nra,nr) xss) else
>  (applyChannel (c,nf,nra,nr) xss)
>  where nfs 0 _ = True
>        nfs 1 ((Fake (a,b)),y) = False
>        nfs 1 _ = True
>        nras 0 _ = True
>        nras 1 (x,(Hijack (a,a',b,b'))) = (a == a') || (not $ honest(a'))
>        nras 1 _ = True
>        nras 2 (x,(Hijack (a,a',b,b'))) = (a == a')
>        nras 2 _ = True
>        nrs 0 _ = True
>        nrs 1 (x,(Hijack (a,a',b,b'))) = (b == b') || (not $ honest(b))
>        nrs 1 _ = True
>        nrs 2 (x,(Hijack (a,a',b,b'))) = (b == b')
>        nrs 2 _ = True
```

The function match takes a list of initial honest and initial dishonest events and discovers which channel they correspond to.

```
>match :: [Event] -> [Event] -> Channel -> Channel -> Channel
>match hs ds (c1,nf1,nra1,nr1) (c2,nf2,nra2,nr2) = collapse (c,nf,nra,nr)
>  where c   = if (c1 == 1 && c2 == 1) then 1 else 0
>        nf  = if (nf1 == 1 && nf2 == 1) then 1 else 0
>        nra = minimum (map trd (map hEvents hs ++ map dEvents ds))
>        nr  = minimum (map fth (map hEvents hs ++ map dEvents ds))
```

The functions hEvents and dEvents tell which channel properties a certain event implies.

```
>hEvents :: Event -> Channel
>hEvents (Send (Sender A,Proxy (Sender A,Receiver B)))    = (1,1,2,2)
>hEvents (Send (Sender A,Proxy (Sender A,Receiver I)))    = (1,1,2,1)
>hEvents (Send (Sender A',Proxy (Sender A',Receiver B)))  = (1,1,0,2)
>hEvents (Send (Sender A',Proxy (Sender A',Receiver I)))  = (1,1,0,1)
>hEvents (Send (Sender I,Proxy (Sender I,Receiver B)))    = (1,0,2,2)
>hEvents (Send (Sender I,Proxy (Sender I,Receiver I)))    = (1,0,2,2)
>hEvents (Send (Sender A,Proxy (Sender A,Receiver B')))   = (1,1,2,0)
>hEvents (Send (Sender A',Proxy (Sender A',Receiver B'))) = (1,1,0,0)
>hEvents (Send (Sender I,Proxy (Sender I,Receiver B')))   = (1,0,2,2)
>hEvents (Fake (Sender A,Proxy (Sender A,Receiver B)))    = (1,0,2,2)
>hEvents (Fake (Proxy (Sender A,Receiver B),Receiver B))  = (1,0,2,2)
```

```
>dEvents :: Event -> Channel
>dEvents (Send (Sender I,Proxy (Sender I,Receiver B)))    = (1,1,2,2)
>dEvents (Send (Sender A,Proxy (Sender A,Receiver B)))    = (1,1,1,2)
>dEvents (Send (Sender A',Proxy (Sender A',Receiver B)))  = (1,1,1,2)
>dEvents (Send (Sender A,Proxy (Sender A,Receiver I)))    = (1,1,2,2)
>dEvents (Send (Sender A',Proxy (Sender A',Receiver I)))  = (1,1,2,2)
>dEvents (Send (Sender I,Proxy (Sender I,Receiver I)))    = (1,1,2,2)
>dEvents (Send (Sender A,Proxy (Sender A,Receiver B')))   = (0,1,2,2)
>dEvents (Send (Sender A',Proxy (Sender A',Receiver B'))) = (0,1,2,2)
>dEvents (Send (Sender I,Proxy (Sender I,Receiver B')))   = (1,1,2,2)

>trd (_,_,x,_) = x
>fth (_,_,_,x) = x
```

B.5 Multiplexing proxy proof script

```
>import Channels
```

We calculate the channel combinations where the expected result (calculated using the elevation rules) and the actual result (calculated using the trace patterns) differ.

```
>difference :: IO()
>difference = (putStr . concat) [
>  "Actual: " ++ (actual c1 c2) ++
>  "Expected: " ++ (expected c1 c2) ++ "\n" |
>   c1 <- hierarchy, c2 <- hierarchy,
>   (actual c1 c2) /= (expected c1 c2)]
```

The expected result (calculated using the elevation rules).

```
>expected :: Channel -> Channel -> String
>expected c1 c2 = "A --[" ++
>  (channelShow c1) ++ "]--> Proxy --[" ++
>  (channelShow c2) ++ "]--> B = " ++
>  (channelShow resultant) ++ "\n"
>  where resultant = collapse (glb (raiseToProxy c1) (raiseFromProxy c2))

>raiseToProxy :: Channel -> Channel
>raiseToProxy (c, nf, nra, nr) = (c, nf, nra, 2)

>raiseFromProxy :: Channel -> Channel
>raiseFromProxy (c, nf, nra, nr) = (c, nf, 2, nr)
```

The actual result (calculated using trace patterns).

```
>actual :: Channel -> Channel -> String
>actual c1 c2 = "A --[" ++
>  (channelShow c1) ++ "]--> Proxy --[" ++
>  (channelShow c2) ++ "]--> B = " ++
>  (channelShow (match (resultant A) (resultant I) c1 c2)) ++ "\n"
>  where resultant a = (map head . toSender a . toProxy) (finalEvent a)
>        toSender a = applyChannel c1 . concat . map
```

```
>                     (initial (Sender a) (Receiver B))
>        toProxy = map preProxy . applyChannel c2 . pre
>        finalEvent a = (Receive (Proxy, Receiver B, Sender a))
```

We use representative identities: A, A', B, B' are honest; I is dishonest.

```
>data Identity = A | A' | B | B' | I
>  deriving (Eq, Show)
```

An agent is either a proxy, a sender, or a receiver.

```
>data Agent = Proxy | Sender Identity | Receiver Identity
>  deriving (Eq, Show)
```

All proxies are honest (so we never question it).

```
>honest :: Agent -> Bool
>honest (Sender a) = a /= I
>honest (Receiver a) = a /= I
```

It's convenient to list the senders and receivers.

```
>senders :: [Agent]
>senders = [Sender A, Sender A', Sender I]

>receivers :: [Agent]
>receivers = [Receiver B, Receiver B', Receiver I]
```

Each event is either a send, receive, fake or hijack. We list the sender's idenitity first, the receiver's second and the third party's (the original sender or the final recipient) third.

```
>data Event = Send (Agent, Agent, Agent) |
>             Receive (Agent, Agent, Agent) |
>             Fake (Agent, Agent, Agent) |
>             Hijack (Agent, Agent, Agent, Agent)
>  deriving (Eq,Show)
```

The function pre calculates which events could have occurred immediately before the final receive event. However, we don't let the intruder fake with his own (e.g. a dishonest) identity.

```
>pre :: Event -> [[Event]]
>pre (Receive (Proxy,b,a)) = sends:fakes:hijacks
>  where sends = [Send (Proxy,b,a), Receive (Proxy,b,a)]
>        fakes = if (honest (a)) then
>                [Fake (Proxy,b,a), Receive (Proxy,b,a)] else []
>        hijacks = [[Send (Proxy,b',a),
>                    Hijack (Proxy,b',b,a),
>                    Receive (Proxy,b,a)] | b' <- receivers]
```

If the proxy p sent a message to b, then who sent the message to p?

```
>preProxy :: [Event] -> [Event]
```

```
>preProxy (Send (Proxy,b,a):xs) = Receive (a,Proxy,b):Send (Proxy,b,a):xs
>preProxy xs = xs
```

Once we know who the proxy received the message from we can work out all possible traces that would result in our original event. However:
We don't let the intruder send messages to the wrong recipient;
We don't let the intruder fake with his own identity, with the wrong sender's identity, or to the wrong recipient.

```
>initial :: Agent -> Agent -> [Event] -> [[Event]]
>initial a1 b1 (Receive (a,Proxy,b):xs) = sends:fakes:hijacks
>  where sends = if (honest(a) || b == b1) then
>                (Send (a,Proxy,b)):(Receive (a,Proxy,b)):xs else []
>        fakes = if (honest(a) && a == a1 && b == b1) then
>                (Fake (a,Proxy,b)):(Receive (a,Proxy,b)):xs else []
>        hijacks = [[Send (a',Proxy,b),
>                    Hijack (a',a,Proxy,b),
>                    Receive (a,Proxy,b)] ++ xs | a' <- senders]
>initial _ _ xs = [xs]
```

We can apply a channel to the events each side of the proxy (apply only looks at the first two events in each trace).

```
>applyChannel :: Channel -> [[Event]] -> [[Event]]
>applyChannel _ [] = []
>applyChannel cs ([]:xss) = applyChannel cs xss
>applyChannel (c,nf,nra,nr) ((x:y:xs):xss) =
>  if (nfs nf (x,y) && nras nra (x,y) && nrs nr (x,y)) then
>  (x:y:xs):(applyChannel (c,nf,nra,nr) xss) else
>  (applyChannel (c,nf,nra,nr) xss)
>  where nfs 0 _ = True
>        nfs 1 ((Fake (a,Proxy,b)),y) = False
>        nfs 1 ((Fake (Proxy,b,a)),y) = False
>        nfs 1 _ = True
>        nras 0 _ = True
>        nras 1 (x,(Hijack (a,a',Proxy,b))) = (a == a') || (not $ honest(a'))
>        nras 1 (x,(Hijack (Proxy,b,b',a))) = True
>        nras 1 _ = True
>        nras 2 (x,(Hijack (a,a',Proxy,b))) = (a == a')
>        nras 2 (x,(Hijack (Proxy,b,b',a))) = True
>        nras 2 _ = True
>        nrs 0 _ = True
>        nrs 1 (x,(Hijack (a,a',Proxy,b))) = True
>        nrs 1 (x,(Hijack (Proxy,b,b',a))) = (b == b') || (not $ honest(b))
>        nrs 1 _ = True
>        nrs 2 (x,(Hijack (a,a',Proxy,b))) = True
>        nrs 2 (x,(Hijack (Proxy,b,b',a))) = (b == b')
>        nrs 2 _ = True
```

The function match takes a list of initial honest and initial dishonest events and discovers which channel they correspond to.

```
>match :: [Event] -> [Event] -> Channel -> Channel -> Channel
>match hs ds (c1,nf1,nra1,nr1) (c2,nf2,nra2,nr2) = collapse (c,nf,nra,nr)
```

```
>  where c   = if (c1 == 1 && c2 == 1) then 1 else 0
>        nf  = if (nf1 == 1 && nf2 == 1) then 1 else 0
>        nra = minimum (map trd (map hEvents hs ++ map dEvents ds))
>        nr  = minimum (map fth (map hEvents hs ++ map dEvents ds))
```

The functions hEvents and dEvents tell which channel properties a certain event implies.

```
>hEvents :: Event -> Channel
>hEvents (Send (Sender A,Proxy,Receiver B))   = (1,1,2,2)
>hEvents (Send (Sender A,Proxy,Receiver I))   = (1,1,2,1)
>hEvents (Send (Sender A,Proxy,Receiver B'))  = (1,1,2,0)
>hEvents (Fake (Sender A,Proxy,Receiver B))   = (1,0,2,2)
>hEvents (Fake (Proxy,Receiver B,Sender A))   = (1,0,2,2)
>hEvents (Send (Sender A',Proxy,Receiver B))  = (1,1,0,2)
>hEvents (Send (Sender I,Proxy,Receiver B))   = (1,0,2,2)
>hEvents (Send (Sender A',Proxy,Receiver I))  = (1,1,0,1)
>hEvents (Send (Sender I,Proxy,Receiver I))   = (1,1,2,2)
>hEvents (Send (Sender A',Proxy,Receiver B')) = (1,1,0,0)
>hEvents (Send (Sender I,Proxy,Receiver B'))  = (1,1,2,2)

>dEvents :: Event -> Channel
>dEvents (Send (Sender I,Proxy,Receiver B))  = (1,1,2,2)
>dEvents (Send (Sender I,Proxy,Receiver I))  = (1,1,2,2)
>dEvents (Send (Sender I,Proxy,Receiver B')) = (1,1,2,2)
>dEvents (Send (Sender A,Proxy,Receiver B))  = (1,1,1,2)
>dEvents (Send (Sender A',Proxy,Receiver B)) = (1,1,1,2)
>dEvents (Send (Sender A,Proxy,Receiver I))  = (1,1,2,2)
>dEvents (Send (Sender A',Proxy,Receiver I)) = (1,1,2,2)
>dEvents (Send (Sender A,Proxy,Receiver B')) = (1,1,2,2)
>dEvents (Send (Sender A',Proxy,Receiver B'))= (1,1,2,2)

>trd (_,_,x,_) = x
>fth (_,_,_,x) = x
```

Appendix C

OpenID Authentication protocols

C.1 OpenID provider first

User-controlled identity

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow op : u, rp \\
\mathit{Message}\ 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 2.1 & rp \rightarrow u : rp, u \\
\mathit{Message}\ 2.2 & u \rightarrow rp : u, op \\
\mathit{Message}\ 7.1 & rp \rightarrow op : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 7.2 & op \rightarrow rp : u \\
\mathit{Message}\ 8 & rp \rightarrow u : m
\end{array}
$$

```
#Channels
1   C NF NRA NR
6.1 C NF NRA NR
6.2 C NR-
7.1 C NR-
7.2 NF NRA-
8   NF NRA-

Session symmetric 1, 6.1
Session symmetric 6.2, 8
Session symmetric 7.1, 7.2
```

OpenID provider's identity

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow op : u, rp \\
\mathit{Message}\ 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 2.1 & rp \rightarrow op : rp, u \\
\mathit{Message}\ 2.2 & op \rightarrow rp : u, op \\
\mathit{Message}\ 7.1 & rp \rightarrow op : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 7.2 & op \rightarrow rp : u \\
\mathit{Message}\ 8 & rp \rightarrow u : m
\end{array}
$$

```
#Channels
1   C NF NRA NR
6.1 C NF NRA NR
6.2 C NR-
2.1 C NR-
2.2 NF NRA-
7.1 C NR-
7.2 NF NRA-
8   NF NRA-

Session symmetric 1, 6.1
Session symmetric 6.2, 8
Session symmetric 2.1, 2.2
Session symmetric 7.1, 7.2
```

C.2 Relying party first

User-controlled identity and association established

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow rp : u \\
\mathit{Message}\ 2.1 & rp \rightarrow u : u \\
\mathit{Message}\ 2.2 & u \rightarrow rp : u, op \\
\mathit{Message}\ 3.1 & rp \rightarrow op : rp, op \\
\mathit{Message}\ 3.2 & op \rightarrow rp : n_k, k \\
\mathit{Message}\ 4.1 & rp \rightarrow u : op, u, n_k, rp \\
\mathit{Message}\ 4.2 & u \rightarrow op : u, n_k, rp \\
\mathit{Message}\ 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 8 & rp \rightarrow u : m
\end{array}
$$

```
#Channels
3.1 C NR-
4.1 NF NRA-
4.2 C NF NRA NR
6.1 C NF NRA NR
6.2 C NR-
8 NF NRA-

Session symmetric 3.1, 3.2
Session symmetric 4.2, 6.1
Session symmetric 6.2, 8
```

OpenID provider's identity and association established

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow rp : op \\
\mathit{Message}\ 3.1 & rp \rightarrow op : rp, op \\
\mathit{Message}\ 3.2 & op \rightarrow rp : n_k, k \\
\mathit{Message}\ 4.1 & rp \rightarrow u : op, n_k, rp \\
\mathit{Message}\ 4.2 & u \rightarrow op : u, n_k, rp \\
\mathit{Message}\ 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 8 & rp \rightarrow u : m
\end{array}
$$

```
#Channels
```

```
3.1 C NR-
3.2 NF NRA-
4.2 C NF NRA NR
6.1 C NF NRA NR
6.2 C NR-
8 NF NRA-

Session symmetric 3.1, 3.2
Session symmetric 4.2, 6.1
Session symmetric 6.2, 8
```

User-controlled identity and direct verification

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow rp : u, op \\
\mathit{Message}\ 2.1 & rp \rightarrow u : u \\
\mathit{Message}\ 2.2 & u \rightarrow rp : u, op \\
\mathit{Message}\ 4.1 & rp \rightarrow u : op, u, rp \\
\mathit{Message}\ 4.2 & u \rightarrow op : u, rp \\
\mathit{Message}\ 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 7.1 & rp \rightarrow op : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 7.2 & op \rightarrow rp : u \\
\mathit{Message}\ 8 & rp \rightarrow u : m
\end{array}
$$

```
#Channels
4.2 C NF NRA NR
6.1 C NF NRA NR
6.2 C NR-
7.1 C NR-
7.2 NF NRA-
8 NF NRA-

Session symmetric 4.2, 6.1
Session symmetric 6.2, 8
Session symmetric 7.1, 7.2
```

OpenID provider's identity and direct verification

$$
\begin{array}{ll}
\mathit{Message}\ 1 & u \rightarrow rp : op \\
\mathit{Message}\ 4.1 & rp \rightarrow u : op, rp \\
\mathit{Message}\ 4.2 & u \rightarrow op : u, rp \\
\mathit{Message}\ 6.1 & op \rightarrow u : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 6.2 & u \rightarrow rp : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 7.1 & rp \rightarrow op : op, u, rp, n_r, n_k, h(k, op, u, rp, n_r, n_k) \\
\mathit{Message}\ 7.2 & op \rightarrow rp : u \\
\mathit{Message}\ 8 & rp \rightarrow u : m
\end{array}
$$

```
#Channels
4.2 C NF NRA NR
6.1 C NF NRA NR
6.2 C NR-
7.1 C NR-
7.2 NF NRA-
8 NF NRA-
```

```
Session symmetric 4.2, 6.1
Session symmetric 6.2, 8
Session symmetric 7.1, 7.2
```