Analysis of Security Protocols

Gavin Lowe

Overview

- Brief introduction to security protocols;
- Model checking of security protocols, particularly using the process algebra CSP and the model checker FDR;
- Fault-preserving simplifying transformations for security protocols;
- Verifying protocols via model checking;
- Secure channels and layering of protocols.

Introduction to Security Protocols

Gavin Lowe

Security protocols

A security protocol is an exchange of messages between two or more agents, with security-relevant goals such as:

- establishing a cryptographic key;
- achieving agent authentication;
- authenticating messages;
- financial transactions;
- voting.

Security protocols

- Security protocols are designed to work in a particularly hostile environment, where the network is under the control of some hostile intruder who can:
  - overhear messages,
  - intercept messages,
  - decrypt or encrypt messages with keys he knows,
  - fake messages.
- The intruder might be a normal user of the system, so honest agents might be willing to run the protocol with him.
- Security protocols normally use cryptography to achieve their goals.

A bit of cryptography: symmetric encryption

- A message m can be encrypted with a cryptographic key k, denoted $\{m\}_k$.
- This message can be decrypted only by someone who knows k.
- If k is a key known only to Alice and Bob, then they can exchange secret messages by encrypting them with k.
- This will also provide authentication and integrity.

A bit of cryptography: public key cryptography

- If $PK(Alice)$ is Alice's public key, then Bob can send Alice a secret message by encrypting it with Alice's public key: $\{m\}_{PK(Alice)}$.
- Alice can decrypt this message with her secret key $SK(Alice)$.
- If Bob sends Alice a secret s encrypted with Alice's public key, and subsequently receives s back, then Bob can deduce that Alice decrypted the message: this authenticates Alice (in some sense).
- Alice can send Bob an authenticated message by encrypting it with her secret key: $\{m\}_{SK(Alice)}$.
- Bob can decrypt this message with Alice's public key, and so verify that Alice sent it.

Perfect cryptography

When analysing protocols, we will normally assume perfect cryptography: attackers can encrypt or decrypt messages only if they possess the correct key.

Needham-Schroeder Public Key Protocol

Msg *1*. $a \rightarrow b \ : \ \{a, n_a\}_{PK(b)}$

Msg *2*. $b \rightarrow a \ : \ \{n_a, n_b\}_{PK(a)}$

Msg *3*. $a \rightarrow b \ : \ \{n_b\}_{PK(b)}$.

The protocol aims to authenticate each agent to the other, and to establish a pair of shared secrets n_a and n_b.

Needham-Schroeder Public Key Protocol

Msg *1*. $a \rightarrow b \ : \ \{a, n_a\}_{PK(b)}$
Msg *2*. $b \rightarrow a \ : \ \{n_a, n_b\}_{PK(a)}$
Msg *3*. $a \rightarrow b \ : \ \{n_b\}_{PK(b)}$.

This protocol has the following attack:

Msg $\alpha.1$. $A \rightarrow I \ : \ \{A, N_a\}_{PK(I)}$
Msg $\beta.1$. $I_A \rightarrow B \ : \ \{A, N_a\}_{PK(B)}$
Msg $\beta.2$. $B \rightarrow I_A \ : \ \{N_a, N_b\}_{PK(A)}$
Msg $\alpha.2$. $I \rightarrow A \ : \ \{N_a, N_b\}_{PK(A)}$
Msg $\alpha.3$ $A \rightarrow I \ : \ \{N_b\}_{PK(I)}$
Msg $\beta.3$. $I_A \rightarrow B \ : \ \{N_b\}_{PK(B)}$.

Needham-Schroeder Public Key Protocol (adapted)

We can prevent the attack as follows:

Msg 1. $a \rightarrow b \ : \ \{a, n_a\}_{PK(b)}$
Msg 2. $b \rightarrow a \ : \ \{b, n_a, n_b\}_{PK(a)}$
Msg 3. $a \rightarrow b \ : \ \{n_b\}_{PK(b)}$.

The attack would now become

Msg $\alpha.1$. $A \rightarrow I \ : \ \{A, N_a\}_{PK(I)}$
Msg $\beta.1$. $I_A \rightarrow B \ : \ \{A, N_a\}_{PK(B)}$
Msg $\beta.2$. $B \rightarrow I_A \ : \ \{B, N_a, N_b\}_{PK(A)}$
Msg $\alpha.2$. $I \rightarrow A \ : \ \{B, N_a, N_b\}_{PK(A)}$

and A would reject message $\alpha.2$.

Analysing security protocols

- This protocol was proposed in 1978, and the attack found in 1995;
- Principle: "If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message."[a]
- Many more protocols are subject to attack;
- Clearly, we need good methods for analysing protocols;
- Fortunately, good methods have been developed in the last thirteen years.

[a] Martín Abadi and Roger Needham, Prudent Engineering Practice for Cryptographic Protocols, IEEE Transactions on Software Engineering, 22(1), 1996.

Summary

- Goals of security protocols;
- A bit of cryptography;
- Threat model;
- The Needham-Schroeder Public Key Protocol, and its attack;
- The need for verification techniques.

CSP-Based Model Checking of Security Protocols

Gavin Lowe

Model checking

Model checking involves building a finite state model of a system, and then using a tool to exhaustively explore the state space, in order to verify some property.

Because they deal with finite state systems, it is necessary to consider a particular system running a protocol (e.g. with one initiator and one responder).

Model checkers are easy to use, and very good at finding attacks. However, if no attack is found, there is no guarantee that no attack exists on some larger system.

Model checking security protocols: the CSP approach

This talk will concentrate on analysing protocols using the process algebra CSP and its model checker FDR.

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

Datatypes and channels

Protocol messages are modelled as elements of a CSP datatype.

We will use channels `send` and `receive` to represent honest agents sending and receiving of messages. For example,

$send.a.b.(Msg1, Encrypt.(PK(b), \langle a, na \rangle), \langle\rangle)$

will represent agent a sending a Message 1 intended for b, with contents $\{a, na\}_{PK(b)}$.

The initiator

$$
\begin{aligned}
&INITIATOR(a, na) = \\
&\quad \Box\, b : Agent \bullet env.a.(Env0, b, \langle\rangle) \rightarrow \\
&\quad send.a.b.(Msg1, Encrypt.(PK(b), \langle a, na\rangle), \langle\rangle) \rightarrow \\
&\quad \Box\, nb : Nonce \bullet receive.b.a.(Msg2, Encrypt.(PK(a), \langle na, nb\rangle), \langle\rangle) \rightarrow \\
&\quad send.a.b.(Msg3, Encrypt.(PK(b), \langle nb\rangle), \langle na\rangle) \rightarrow \\
&\quad SKIP
\end{aligned}
$$

Plumbing the network

The intruder

- The main difficulty in creating the *INTRUDER* process is identifying which messages an intruder can build and understand.
- We need to build an inference system: what can the intruder create, given what it knows and has heard?
- This depends on the details of the types of messages and the encryption-related techniques used.

The intruder

We can build a set *Fact* of all subfacts of all possible protocol messages.

We can then model the possible deductions the intruder might make as a relation $\vdash$. Informally, $X \vdash f$ means that from the set of facts X, the intruder can produce f. The relation is defined using rules such as

$$f \in X \Rightarrow X \vdash f,$$

$$X \vdash k \wedge (\forall f \text{ in } fs \bullet X \vdash f) \Rightarrow X \vdash Encrypt.(k, fs),$$

$$X \vdash Encrypt.(k, fs) \wedge X \vdash k^{-1} \wedge f \text{ in } fs \Rightarrow X \vdash f.$$

The intruder process: first attempt

We could parameterize the intruder process by the set of facts that he knows:

$$
\begin{aligned}
&INTRUDER_0(S) = \\
&\quad hear?f \rightarrow INTRUDER_0(S \cup \{f\}) \\
&\quad \Box \\
&\quad say?f : S \rightarrow INTRUDER_0(S) \\
&\quad \Box \\
&\quad leak?f : S \cap ALL\_SECRETS \rightarrow INTRUDER_0(S) \\
&\quad \Box \\
&\quad \Box f : Fact, S \vdash f \bullet infer.f \rightarrow INTRUDER_0(S \cup \{f\})
\end{aligned}
$$

This can be instantiated with some suitable initial knowledge, and the events renamed to synchronise with the honest agents.

The intruder process

This definition ensures

$$\forall\, tr' \frown \langle say.f \rangle \in traces(INTRUDER_0(IIK)) \bullet$$
$$IIK \cup \{f' \mid hear.f' \text{ in } tr'\} \vdash f.$$

So when the events are renamed:

$$\forall\, tr' \frown \langle receive.a.b.f \rangle \in traces(INTRUDER(IIK)) \bullet$$
$$IIK \cup \{f' \mid send.a'.b'.f' \text{ in } tr'\} \vdash f.$$

A highly parallel intruder

- The above design does not work in practice.
- FDR produces explicit state machines for sequential processes, such as *INTRUDER*.
- If there are N facts that the intruder might learn, then the *INTRUDER* process has 2^N states; typically N is of the order of *1000*, so FDR will not be able to build this process.
- Most states of the intruder will not be reachable.
- Instead, we build the intruder out of N component processes, one for each fact, synchronising appropriately on deductions.

Chasing inferences

- When the intruder learns one fact, he can often use it to deduce many more facts; thus deductions often come together.
- The more deductions that are made, the more the intruder can do, so we can force the intruder to make all possible inferences.
- If the intruder can make k such new deductions, then there are (maybe) $k!$ orders in which those deductions can be made.
- But all different orders of the deductions reach the same state, so it doesn't matter what order they are made in, so we can pick an arbitrary order, rather than considering all possible orders.
- The function `chase` will do this for us: it forces τ events to occur, but picking an arbitrary order.

CSP refinement

The model checker FDR tests for refinement in any of the three semantic models.

Most security properties are safety properties, so it is appropriate to use traces refinement:

$SPEC \sqsubseteq_{tr} SYSTEM \Leftrightarrow traces(SPEC) \supseteq traces(SYSTEM)$.

So we need to find a specification process $SPEC$ that allows precisely those traces that do not represent a security breach.

Testing for secrecy

Suppose we want to test the following assertion: whenever an agent completes a run of the protocol, the value of na is secret.

We arrange for the honest agents to perform signal events to indicate that they believe certain values are secret.

An event $signal.Claim\_Secret.a.na.\{b\}$ represents that a believes that na is a secret that only b should be allowed to learn.

We rename each instance of a send of a message 3 in the protocol to an appropriate $signal.Claim\_Secret$ event:

$$[[send.a.b.(Msg3, Encrypt(PK(b), \langle nb \rangle), \langle na \rangle) \leftarrow$$
$$signal.Claim\_Secret.a.na.\{b\} \mid$$
$$a, b \in Agent, na, nb \in Nonce]]$$

Note that na does not occur in the message itself, so we have to include it in an extra field of the CSP event.

Testing for secrecy

We want to test that every trace tr of $SYSTEM_S$ satisfies:

$$\forall\, s : ALL\_SECRETS \bullet \forall\, tr' \bullet tr' \frown \langle leak.s \rangle \leq tr \Rightarrow$$
$$\forall\, a \in Honest\,;\, bs \in \mathrm{P}\, Honest \bullet \neg(signal.Claim\_Secret.a.s.bs \text{ in } tr')$$

The following process allows all traces that satisfy this requirement for the secret s:

$$SECRET\_SPEC_0(s) =$$
$$signal.Claim\_Secret?a!s?bs \rightarrow$$
$$\text{if } I \in bs \text{ then } SECRET\_SPEC_0(s) \text{ else } SECRET\_SPEC_1(s)$$
$$\Box$$
$$leak.s \rightarrow SECRET\_SPEC_0(s)$$
$$SECRET\_SPEC_1(s) =$$
$$signal.Claim\_Secret?a!s?bs \rightarrow SECRET\_SPEC_1(s)$$

Testing for secrecy

The following process allows all traces that satisfy the requirement for all secrets:

$$SECRET\_SPEC = |||_{s:ALL\_SECRETS} SECRET\_SPEC_0(s).$$

The following refinement test checks that all traces of the system satisfy the requirement:

$$assert\ SECRET\_SPEC \sqsubseteq_T SYSTEM\_S$$

Testing for authentication

Suppose we want to check the following authentication condition:

> Whenever the responder b completes the protocol, apparently with a, then a has been running the protocol as initiator, apparently with b, and the two agents agree upon the values of the nonces na and nb.

Testing for authentication

We use two new signal events:

- *signal.Complete.RESPONDER_role.b.a.na.nb*: b has completed the protocol as responder, apparently with a, using the values na and nb for the nonces;
- *signal.Running.INITIATOR_role.a.b.na.nb*: a is running the protocol as initiator, apparently with b, using the values na and nb for the nonces.

The *Complete* is obtained by renaming the responder's receive of message 3; the *Running* is obtained by renaming the initiator's send of message 3.

We then check that whenever the *Complete* event occurs, the *Running* event has occurred previously.

Casper

Writing CSP models of security protocols by hand is tedious, hard-work and error-prone.

Casper is a compiler that will produce CSP models of security protocols, suitable for checking using FDR.

The user prepares a script, describing the protocol, some typing information, some security properties to be considered, and a description of the system to be checked. Casper compiles this into CSP.

A Casper script

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

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

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

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

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

```
#Actual variables
A, B, I : Agent
Na, Nb, Ni : Nonce

#Functions
symbolic PK, SK

#System
INITIATOR(A, Na)
RESPONDER(B, Nb)

#Intruder Information
Intruder = I
IntruderKnowledge = {A, B, I, Ni, PK, SK(I)}
```

Extensions

- Modelling of algebraic equivalences.
- Password guessing attacks.
- Key compromise.
- Application areas: web services protocols, voting protocols, smart cards, pervasive computing, group protocols, ...

Limitations

If we find no attacks on a small system Π running the protocol, then this doesn't necessarily mean that there would be no attacks if we considered some larger system Π'.

There are infinitely many systems that could run the protocol, so how can we analyse them all using a finite amount of model checking? (See later talk.)

Summary

- Modelling honest agents;
- Modelling the intruder;
- Capturing the requirements;
- Casper;
- Extensions;
- Limitations.

Other approaches

- Other model checkers: Interrogator (Jon Millen); Brutus (Ed Clarke et al.); Murϕ (John Mitchell et al.); SATMC (Armando, Carbone, Compagna); ...
- NRL Protocol Analyzer (Cathy Meadows);
- Strand Spaces (Joshua Guttman et al.);
- Rank functions (Steve Schneider et al.);
- Athena (Dawn Song, Adrian Perrig);
- Theorem provers (Larry Paulson; Ernie Cohen; ...);
- ProVerif (Bruno Blanchet).

Fault Preserving Simplifying Transformations for Security Protocols

Gavin Lowe
(joint work with Mei Lin Hui)

Analyzing security protocols

There are a number of techniques for analyzing security protocols, either using a tool or by hand.

These techniques are good at analyzing small protocols such as:

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

But most commercial protocols are somewhat larger ...

Msg1. $M \rightarrow C$: $Accepts$, $MerchantAmount$, $MerchantAmount2Optional$, $MerchantCcId$, $MerchantOrderId$, $MerchantDate$, $MerchantSwVersion$, $Note$, $Payload$, $PayloadNote$, $Type$, $UrlCancel$, $UrlFail$, $UrlPayTo$, $UrlSuccess$, $MD5(Payload)$, $MerchantSignedHashKey\_$, $MerchantSignedHash\_$

Msg2. $C \rightarrow M$: $CyberKey$, $Date$, Id, $MerchantCcId$, $MerchantDate$, $OrderId$, $ServiceCategory$, $Transaction$, $Type$, $MerchantSignedHashKey\_$, $PrHash\_$, $PrSignedHash\_$, $OpaqPrefixCH1\_$, $OpaqueCH1\_$

Msg3. $M \rightarrow CB$: $CyberKey$, $MerchantCcId$, $MerchantCyberKey$, $MerchantDate$, $MerchantTransaction$, $ServiceCategory$, $OpaqPrefixCH1\_$, $OpaqueCH1\_$, $MerchantOpaqPrefixCM1\_$, $MerchantOpaqueCM1\_$

Msg4. $CB \rightarrow M$: $MerchantCcId$, $MerchantTransaction$, $MerchantDate$, $ServiceCategory$, $MerchantOpaqueCM6\_$, $OpaqueCM6\_$

Msg5. $M \rightarrow C$: $Date$, $MerchantCcId$, $MerchantDate$, $MerchantMessage$, $MerchantResponseCode$, $MerchantSwVersion$, Id, $ServiceCategory$, $Transaction$, $Type$, $PrHash\_$, $PrSignedHash\_$, $OpaqueCM6\_$

$$
\begin{aligned}
&MerchantSignedHashKey\_ = MD5(PK(M))\\
&MerchantSignedHash\_ =\\
&\quad \{MD5\ (Accepts, MerchantDate, MerchantAmount, Note, Type, UrlCancel, UrlFail,\\
&\qquad UrlPayto, UrlSuccess, MerchantSignedHashKey\_)\}_{SK(M)}\\
&PrHash\_ =\\
&\quad MD5\ (Accepts, Date, MerchantAmount, MerchantCcId, MerchantOrderId, MerchantSignedHashKey\_,\\
&\qquad Note, Type, UrlCancel, UrlFail, UrlPayTo, UrlSuccess)\\
&PrSignedHash\_ = MerchantSignedHash\_\\
&OpaqPrefixCH1\_ = \{kcs\}_{PKCyberKey}\\
&OpaqueCH1\_ =\\
&\quad \{Amount, CardCIdOptional, CardCityOptional, CardCountryOptional, CardExpirationDate,\\
&\quad\ CardName, CardNumber, CardOtherFieldsOptional, CardPostalCodeOptional,\\
&\quad\ CardPrefixOptional, CardSalt, CardStateOptional, CardStreetOptional, CardType, SwVersion,\\
&\quad\ MD5(PK(C)), SignatureCH1\_\}_{kcs}\\
&SignatureCH1\_ =\\
&\quad \{MD5\ (Amount, CardCIdOptional, CardCityOptional, CardCountryOptional, CardExpirationDate,\\
&\qquad CardName, CardNumber, CardOtherFieldsOptional, CardPostalCodeOptional,\\
&\qquad CardPrefixOptional, CardSalt, CardStateOptional, CardStreetOptional,\\
&\qquad CardType, CyberKey, Date, Id, MerchantCcId, MerchantSignedHashKey\_, OrderId, PrHash\_,\\
&\qquad PrSignedHash\_, SwVersion, Transaction, Type)\}_{SK(C)}\\
&MerchantOpaqPrefixCM1\_ = \{kms\}_{PKMerchantCyberKey}\\
&MerchantOpaqueCM1\_ =\\
&\quad \{Date, DescriptionListOptional, Id, MerchantAmount, MerchantDba, MerchantLocationOptional,\\
&\quad\ MerchantMessage, MerchantSignedHashKey\_, MerchantSwMessageOptional,\\
&\quad\ MerchantSwServerOptional, MerchantSwVersion, MerchantUrlOptional, OrderId,\\
&\quad\ PrHash\_, PrSignedHash\_, RetrievalReferenceNumberOptional,\\
&\quad\ ServerDateMerchantOptional, TerminalIdFuture, Transaction, TransactionDescriptionOptional,\\
&\quad\ Type, MD5(PKMerchantCyberKey), MerchantSignatureCM1\_\}_{kms}\\
&MerchantSignatureCM1\_ = \ldots\\
&MerchantOpaqueCM6\_ = \ldots\\
&OpaqueCM6\_ = \ldots
\end{aligned}
$$

Fault-preserving simplifying transformations

We define a fault-preserving simplifying transformation upon a protocol to be a transformation such that if the original is insecure, then so is the simplified version.

The idea is to continue applying such transformations until one reaches a protocol that one can analyze. There are three possibilities:

1. the simplified protocol is secure, in which case so is the original;
2. the simplified protocol has an attack, and (essentially) the same attack works on the original protocol;
3. our simplifications have introduced a new attack, which wasn't possible on the original.

We want to simplify the protocol as far as possible, without introducing new attacks.

Overview of talk

I will:

- Present a model of a system running a protocol, based upon CSP;
- Formalize the idea of a transformation, and what it means for a transformation to be fault-preserving;
- Present a result showing that if a transformation satisfies a couple of properties then it is fault-preserving;
- Give examples of transformations satisfying these properties;
- Apply these transformations to the CyberCash protocol.

Messages

Define the space of messages by:

$$\begin{aligned} \mathit{Message} ::= {} & \mathsf{Atom}\ \mathit{Atom} \mid \\ & \mathsf{Pair}\ \mathit{Message}\ \mathit{Message} \mid \\ & \mathsf{Encrypt}\ \mathit{Message}\ \mathit{Atom} \mid \\ & \mathsf{Hash}\ \mathit{HashFn}\ \mathit{Message}. \end{aligned}$$

We will write:

- (M, M') for $\mathsf{Pair}\ M\ M'$;
- $\{M\}_K$ for $\mathsf{Encrypt}\ M\ K$;
- $g(\!|M|\!)$ for $\mathsf{Hash}\ g\ M$.

Honest agents

Every honest agent taking part in the protocol can be modelled as a CSP process, with events representing sending and receiving of messages, and signalling facts about the state of the agent.

We write P_A for the process representing the agent with identity A.

The intruder

We write $B \vdash M$ if the intruder can create message M from the set of messages B:

member $M \in B \Rightarrow B \vdash M$;

pairing $B \vdash M \wedge B \vdash M' \Rightarrow B \vdash (M, M')$;

splitting $B \vdash (M, M') \Rightarrow B \vdash M \wedge B \vdash M'$;

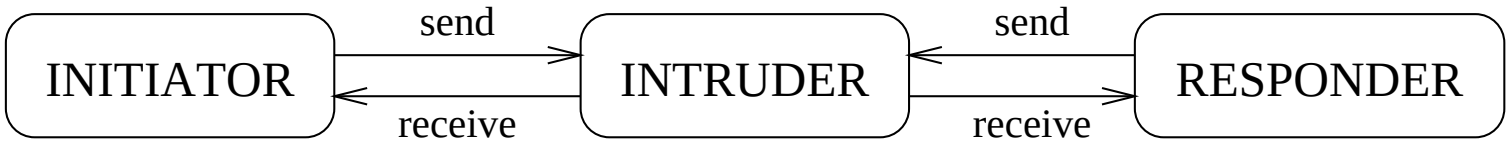
encryption $B \vdash M \wedge B \vdash \mathsf{Atom}\, K \wedge K \in Key \Rightarrow B \vdash \{M\}_K$;

decryption $B \vdash \{M\}_K \wedge B \vdash \mathsf{Atom}\, K^{-1} \Rightarrow B \vdash M$;

hashing $B \vdash M \wedge g \in HashFn \Rightarrow B \vdash g(\!|M|\!)$;

We represent the intruder by a CSP process $INTRUDER(IIK)$, where IIK is the set of messages the intruder knows initially. The intruder can intercept messages, and fake messages that can be derived under the $\vdash$ relation from what it has seen earlier.

A system running the protocol



The system is modelled by the process:

$$SYSTEM \mathrel{\widehat{=}} \left(\left|\left|\left| \right.\right.\right._{A \in Agent} P_A \right) \underset{\{|send, receive|\}}{\|} INTRUDER(IIK),$$

where $Agent$ is the set of all agents in the system, and IIK is the initial knowledge of the intruder.

Formalizing the transformations

We formalize each transformation using a function $\phi : Message \rightarrow Message$, so that if an agent sends or receives a message M in the original protocol, then the same agent sends or receives the message $\phi(M)$ in the transformed protocol.

The honest agent represented by the process P_A in the original protocol is represented by the process $\phi(P_A)$ in the transformed protocol.

The intruder for the new protocol is essentially the same, but we take the intruder's initial knowledge to be IIK': $INTRUDER(IIK')$.

The system running the transformed protocol is

$$SYSTEM' \mathrel{\widehat{=}} \left(\Big|\Big|\Big|_{A \in Agent} \phi(P_A) \right) \underset{\{|send, receive|\}}{\|} INTRUDER(IIK').$$

Relating the traces of the systems

Theorem 1 If

$$\phi(IIK) \subseteq IIK', \tag{1}$$

and

$$\forall B \in \mathrm{P}(Message) ; M \in Message \bullet \\ B \cup IIK \vdash M \Rightarrow \phi(B) \cup IIK' \vdash \phi(M), \tag{2}$$

then

$$\forall tr \in traces(SYSTEM) \bullet \phi(tr) \in traces(SYSTEM').$$

The proof proceeds by proving:

$$\forall tr \in traces(INTRUDER(S)) \bullet \phi(tr) \in traces(INTRUDER(\phi(S))).$$

Preservation of attacks against secrecy

Suppose tr is a secrecy attack in the original protocol, concerning the secrecy of s.

Then tr is of the form

$\langle \ldots, signal.Claim\_Secret.A.s.B, \ldots, leak.s, \ldots \rangle$.

Then the simplified protocol has the trace $\phi(tr)$, which is of the form

$\langle \ldots, signal.Claim\_Secret.A.\phi(s).B, \ldots, leak.\phi(s), \ldots \rangle$,

which is an attack against the secrecy of $\phi(s)$.

Hence attacks on secrecy are preserved.

Preservation of attacks against authentication

In order to show that a transformation preserves attacks on authentication, we require an additional condition. If we want to test for agreement on the sequence of values Ms, then

$\forall\, Ms' \in Message^* \bullet \phi(Ms) \neq \phi(Ms')$.

Suppose tr is an authentication attack in the original protocol, concerning agreement on Ms.

Then it is of the form $tr'^\frown\langle signal.Complete.A.B.Ms\rangle$ with no event of the form $signal.Running.B.A.Ms$ in tr'.

Then the simplified protocol has the trace $\phi(tr)$, which is of the form $\phi(tr')^\frown\langle signal.Complete.A.B.\phi(Ms)\rangle$ with no event of the form $signal.Running.B.A.\phi(Ms)$ in $\phi(tr')$.

So $\phi(tr)$ is an attack on authentication.

Hence attacks on authentication are preserved.

Remove encryptions

We can define a simplifying transformation that replaces encrypted messages of the form $\{M\}_K$ by the body M, for every $\{M\}_K$ in some set $Encs$:

$$\begin{aligned}
\phi(\mathsf{Atom}\ A) &\mathrel{\widehat{=}} \mathsf{Atom}\ A,\\
\phi(M, M') &\mathrel{\widehat{=}} (\phi(M), \phi(M')),\\
\phi(\{M\}_K) &\mathrel{\widehat{=}} \begin{cases} \phi(M) & \text{if } \{M\}_K \in Encs,\\ \{\phi(M)\}_K & \text{otherwise,}\end{cases}\\
\phi(g(\!|M|\!)) &\mathrel{\widehat{=}} g(\!|\phi(M)|\!).
\end{aligned}$$

We define the intruder's knowledge in the new system by:

$$IIK' \mathrel{\widehat{=}} \phi(IIK).$$

We can show that these definitions satisfy equations (1) and (2). Hence this is a fault-preserving simplifying transformation.

Other transformations

Removing fields completely Removing all occurrences of particular values.

Strip of hash functions Replace hash function applications of the form $h(\lfloor M \rfloor)$ by M (for $h(\lfloor M \rfloor)$ taken from some suitable set).

Remove hashed message Completely remove hash function applications of the form $h(\lfloor M \rfloor)$ (for $h(\lfloor M \rfloor)$ taken from some suitable set).

Renaming atoms Apply some function $f : Atom \to Atom$ to all atoms.

Coalescing pairs Replace adjacent pairs of atoms $(\mathsf{Atom}\ A, \mathsf{Atom}\ A')$ by $\mathsf{Atom}\ A$ (for (A, A') taken from some suitable set).

Swapping pairs Replace adjacent pairs of atoms $(\mathsf{Atom}\ A, \mathsf{Atom}\ A')$ by $(\mathsf{Atom}\ A', \mathsf{Atom}\ A)$ (for (A, A') taken from some suitable set).

The Cybercash protocol

The CyberCash protocol can be simplified using these transformations to:

$$
\begin{array}{lll}
\text{Msg } 1. & M \rightarrow C & : MerchantCcId, MerchantAmount \\
\text{Msg } 2. & C \rightarrow M & : Id, MerchantAmount, MerchantCcId, Transaction, \{kcs\}_{PKCyberKey}, \\
 & & \{Amount, CardNumber, MerchantCcId, \\
 & & \quad Id, MerchantAmount, Transaction\}_{kcs} \\
\text{Msg } 3. & M \rightarrow CB & : MerchantCcId, MerchantTransaction, \{kcs\}_{PKCyberKey}, \\
 & & \{Amount, CardNumber, MerchantCcId, \\
 & & \quad Id, MerchantAmount, Transaction\}_{kcs}, \\
 & & \{kms\}_{PKMerchantCyberKey}, \\
 & & \{Id, MerchantAmount, MerchantCcId, \\
 & & \quad MerchantTransaction, Transaction\}_{kms} \\
\text{Msg } 4. & CB \rightarrow M & : MerchantCcId, MerchantTransaction, \\
 & & \{CardNumber, MerchantAmount, Id, MerchantId, Transaction\}_{kms}, \\
 & & \{CardNumber, Amount\}_{kcs} \\
\text{Msg } 5. & M \rightarrow C & : MerchantCcId, MerchantAmount, Transaction, Id, \\
 & & \{CardNumber, Amount\}_{kcs} \ .
\end{array}
$$

which appears to be secure, as far as secrecy is concerned.

Summary

Many real-world protocols are too complex to be feasible to analyse.

We can often see how to simplify the protocols to make analysis feasible.

This work gives formal justification to that approach.

Verifying Protocols via Model Checking

Gavin Lowe
based on work with Bill Roscoe,
Eldar Kleiner, Philippa Broadfoot (Hopcroft)

Verifying protocols via model checking

Recall that when we do a standard analysis of a protocol using model checking, we consider a particular system, with some fixed choice of initiators and responders.

If we find no attack on this system, then that doesn't guarantee that there is no attack on other systems.

In practice, most attacks are found by model checking a few small systems. But we would like some stronger guarantees. We would like to be able to perform model checking in a way that allows us to deduce that the protocol is secure for all systems.

Verifying protocols via model checking

There are infinitely many different settings where the protocol could be run for two reasons:

- There are infinitely many choices for the system, i.e., the number of agents taking the different roles (e.g. initiator and responder), and the values with which the parameters are instantiated;
- There are infinitely many choices for the atomic types *Agent*, *Nonce*, etc.

Verifying protocols via model checking

Idea:

- Build an abstract model Abs such that, for every system Π running the protocol, if there is an attack on Π, then there is an attack on Abs.
- If we analyse Abs and find no attacks, then we can deduce that the protocol is secure for all systems Π.
- The analysis might find false attacks on Abs (since analysis of security protocols is undecidable in general); we try to make such false attacks rare.

Verifying protocols via model checking

The approach proceeds in two stages:

- In the first stage we reduce the infinitely many different systems to a single system, by capturing the behaviours of all but one honest agent within the intruder process. (However, there are still infinitely many different choices for the atomic types.)
- In the second stage, we show that it is enough to consider a single choice for the types.

We will illustrate the technique on the corrected version of the Needham Schroeder Public Key Protocol:

Msg *1*. $a \rightarrow b \ : \ \{a, n_a\}_{PK(b)}$
Msg *2*. $b \rightarrow a \ : \ \{b, n_a, n_b\}_{PK(a)}$
Msg *3*. $a \rightarrow b \ : \ \{n_b\}_{PK(b)}$.

Internalising agents

Suppose we are trying to show that there are no secrecy or authentication attacks against any initiator. By symmetry, it is enough to consider a particular initiator instance, say $Initiator(Alice, Na)$, which we call the principal initiator instance.

For convenience, we partition the type of all nonces into:

- Na, the principal initiator's nonce;
- $Nonce^{Init}$, the nonces that other initiators might use;
- $Nonce^{Resp}$, the nonces that responders might use.

Internalising agents

Consider some system Π running the protocol, e.g. with honest agents:

$Initiator(Alice, Na), Initiator(A_1, Na_1), \ldots, Initiator(A_m, Na_m),$
$Responder(B_1, Nb_1), \ldots, Responder(B_n, Nb_n)$

(with the nonces distinct, $Na_1, \ldots, Na_m \in Nonce^{Init}$, and $Nb_1, \ldots, Nb_n \in Nonce^{Resp}$).

What we do is build a model Abs_0 where the behaviours of $Initiator(A_1, Na_1)$, $\ldots$, $Initiator(A_m, Na_m)$, $Responder(B_1, Nb_1)$, $\ldots$, $Responder(B_n, Nb_n)$ are captured within the intruder; the model then is built from just the principal initiator and the extended intruder.

Internalising agents: message 1

At any time, the initiator $Initiator(A_i, Na_i)$ could send its message 1 to some agent b: $\{A_i, Na_i\}_{PK(b)}$.

We can capture the fact that the intruder could learn this message, by including all such messages in the intruder's initial knowledge.

We can overapproximate this initial knowledge by including all messages of the form $\{a, na\}_{PK(b)}$ for $a, b \in Agent,\ na \in Nonce^{Init}$.

Note that this new initial knowledge is independent of the choice of system.

Internalising agents: message 2

Suppose the intruder knows a message of the form of message 1 of the protocol: $\{a, n\}_{PK(B_i)}$. Then he can send this to the responder $Responder(B_i, Nb_i)$ and so learn the corresponding message 2: $\{B_i, n, Nb_i\}_{PK(a)}$.

We can capture this behaviour by giving the intruder a deduction:

$$\{a, n\}_{PK(B_i)} \vdash \{B_i, n, Nb_i\}_{PK(a)}$$

We can overapproximate all such behaviours with all responders by giving the intruder all deductions of the form:

$$\{a, n\}_{PK(b)} \vdash \{b, n, nb\}_{PK(a)}, \qquad \text{for all } a,\ b,\ n \text{ and for all } nb \in Nonce^{Resp}.$$

Note that these deductions are independent of the choice of the system.

Internalising agents: message 3

We can do similarly for the interaction the intruder can have with an initiator (other than the principal initiator) by sending a message 2 and receiving the corresponding message 3. We give the intruder all deductions of the form

$\{b, na, n\}_{PK(a)} \vdash \{n\}_{PK(b)}$, for all a, b, n and for all $na \in Nonce^{Init}$.

Relating the traces

Observation: If tr is any trace that a system Π can perform, then Abs_0 can perform the same trace, except where all events of the non-principal agents are replaced with corresponding deductions of the intruder.

In this sense, Abs_0 captures the behaviours of all systems Π (for a given choice of $Agent$, $Nonce^{Init}$ and $Nonce^{Resp}$).

Preservation of attacks against secrecy

Suppose tr is a secrecy attack in Π against the principal initiator, concerning the secrecy of s.

Then tr is of the form

$\langle \ldots, signal.Claim\_Secret.Alice.s.B, \ldots, leak.s, \ldots \rangle$.

Then Abs_0 has a trace of the same form, which is also an attack against the secrecy of s.

Hence this transformation preserves attacks on secrecy.

Preservation of attacks against authentication

Let's consider authentication of the responder to the principal initiator with agreement on the initiator's nonce.

To do this we need to tweak the model Abs_0 slightly. In Π, the responder performs a $signal.Running.B.Alice.na$ event on sending message 2. In Abs_0 we therefore rename the deduction that corresponds to message 2 to the corresponding event $signal.Running.B.Alice.na$.

Suppose tr is an attack against authentication within Π.

Then it is of the form $tr' \frown \langle signal.Complete.Alice.B.Na \rangle$ with no event of the form $signal.Running.B.Alice.Na$ in tr'.

Then in Abs_0, there is a trace of the same form, which is an attack against authentication.

Hence this transformation preserves attacks on authentication.

Internalising: taking stock

The abstract model Abs_0 can be used to identify attacks upon all systems running the protocol.

However, it is parameterised by the (potentially infinite) types $Agent$, $Nonce^{Init}$ and $Nonce^{Resp}$.

Renaming

Let ϕ be an arbitrary (type-preserving) function over the atomic types *Agent* and *Nonce*.

We lift ϕ to messages, events, traces, etc., in the obvious way.

Later we will choose ϕ to collapse the atomic types to some fixed finite types.

Renaming

Consider the system Abs obtained by renaming Abs_0 under ϕ, using types $\phi(Agent)$ and $\phi(Nonce)$. More precisely:

- The principle initiator $Initiator(Alice, Na)$ is replaced by $Initiator(\phi(Alice), \phi(Na))$;
- Each value x in the intruder's initial knowledge is replaced by $\phi(x)$;
- All agents are restricted to use values from $\phi(Agent)$ and $\phi(Nonce)$.

We will show that every attack upon Abs_0 is reflected by an attack upon Abs.

Renaming

Observation: If tr is any trace that Abs_0 can perform, then Abs can perform $\phi(tr)$.

This observation depends crucially on the following property of the intruder's deduction relation:

$$X \vdash f \;\Rightarrow\; \phi(X) \vdash \phi(f)$$

(Roscoe calls such deductive systems positive.)

(Formally, the observation is justified by results from data independence.)

Preservation of attacks against secrecy

Let's consider secrecy attacks against the principal initiator $Alice$. By symmetry, it's enough to consider attacks in runs with a particular responder Bob.

Suppose $\phi(Alice) \neq \phi(i) \neq \phi(Bob)$ for all intruder identities i.

Suppose tr is an attack against the secrecy of s.

Then tr is of the form

$\langle \ldots, signal.Claim\_Secret.Alice.s.Bob, \ldots, leak.s, \ldots \rangle$.

Hence the trace $\phi(tr)$ of Abs is of the form

$\langle \ldots, signal.Claim\_Secret.\phi(Alice).\phi(s).\phi(Bob), \ldots, leak.\phi(s), \ldots \rangle$.

So $\phi(tr)$ is an attack against the secrecy of $\phi(s)$.

Hence this transformation preserves attacks on secrecy.

Preservation of attacks against authentication

Let's consider authentication of the responder to the principal initiator $Alice$ with agreement on the initiator's nonce Na. By symmetry, it's enough to consider just authentication of a particular responder Bob.

Suppose $\phi(Alice) \neq \phi(c)$ for all $c \neq Alice$, $\phi(Bob) \neq \phi(c)$ for all $c \neq Bob$, and $\phi(n) \neq \phi(Na)$ for all $n \neq Na$.

Suppose tr is an attack against authentication.

Then it is of the form $tr' \frown \langle signal.Complete.Alice.B.Na \rangle$ with no event of the form $signal.Running.B.Alice.Na$ in tr'.

Then the trace $\phi(tr)$ of Abs is of the form $\phi(tr') \frown \langle signal.Complete.\phi(Alice).\phi(Bob).\phi(Na) \rangle$ with no event of the form $signal.Running.\phi(Bob).\phi(Alice).\phi(Na)$ in $\phi(tr')$.

So $\phi(tr)$ is an attack on authentication. Hence this transformation preserves attacks on authentication.

Choosing ϕ

We have shown that attacks against an arbitrary system Π running the protocol are reflected in attacks on Abs.

Suppose we define ϕ by

$$\begin{aligned}
\phi(Alice) &= Alice \\
\phi(Bob) &= Bob \\
\phi(c) &= Mallory, \qquad \text{for all } c \in Agent,\ c \neq Alice, Bob \\
\phi(Na) &= Na \\
\phi(n) &= N1 \qquad \text{for } n \in Nonce^{Init} \\
\phi(n') &= N2 \qquad \text{for } n \in Nonce^{Resp}
\end{aligned}$$

Then Abs is finite: it uses 3 agent identities and 3 nonces.

We can model check Abs; FDR finds no attacks.

Hence we can deduce that there are no attacks upon Π: the protocol is secure.

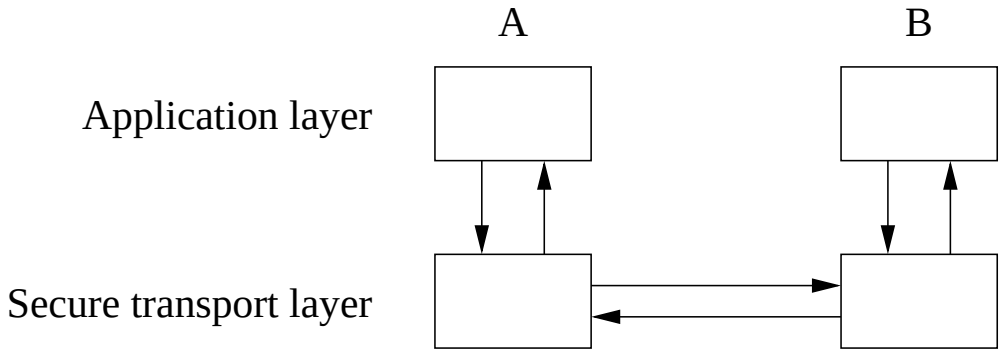
Secure Channels and Layering of Protocols

Gavin Lowe
based on work with Christopher Dilloway,
Allaa Kamil, Philippa Broadfoot (Hopcroft)

Secure transport protocols

Many distributed security architectures make use of a secure transport protocol, such as TLS, to protect communications on the network.

An application protocol, layered on top of the secure transport protocol, provides extra security guarantees (and functionality).



Different application protocols might require different security properties from the secure transport protocol.

The secure transport protocol provides a secure channel to the application protocol.

Examples

- An e-commerce protocol;
- A mechanism for deciding whether a file access should be authorised in a computational grid;
- A mechanism for delegating file access rights;
- A single sign on mechanism.

Know the enemy

There are two types of intruder or attacker that one needs to guard against:

Dishonest outsiders: dishonest agents who are not openly participating in this instance of the transaction, but who might instead imitate a participant; the secure transport protocol aims to protect against these.

Dishonest insiders: dishonest agents who are openly participating in this instance of the transaction, using their own identities, but who attempt to subvert the transaction in some way; the application protocol aims to protect against these.

Overview

- Example: a single sign on protocol;
- Formalising the transport layer services;
- Analysing such layered designs.

SAML Single Sign On Protocol

Single sign on protocols allow a user A to authenticate herself once to some trusted identity provider IdP, and for that identity provider to provide credentials with which the user can authenticate herself to service providers SP.

Such protocols are normally layered on top of a secure transport protocol, e.g. to provide mutual authentication between the user and the identity provider, to authenticate the service provider to the user, and to keep various data confidential.

SAML Single Sign On Protocol

One version of the SAML Single Sign On Protocol is

Msg *1*. $A \rightarrow SP \quad : \quad A, SP, request$

Msg *2*. $SP \rightarrow A \quad : \quad A, IdP, ID, SP, request$

Msg *3*. $A \rightarrow IdP \quad : \quad A, IdP, ID, SP, request$

Msg *4*. $IdP \rightarrow A \quad : \quad ID, SP, IdP, \{ID, A, IdP, SP\}_{SK(IdP)}, request$

Msg *5*. $A \rightarrow SP \quad : \quad ID, SP, IdP, \{ID, A, IdP, SP\}_{SK(IdP)}, request$

where *request* is some request from A to SP, and ID is an identifier for the request generated by SP.

The communications between A and SP are over a channel where SP is authenticated, and where messages are confidential (e.g. unilateral TLS); the communications between A and IdP are over a channel where both parties are authenticated, and where messages are confidential (e.g. bilateral TLS or unilateral TLS augmented with A's password).

SAML-based SSO for Google Applications

Google adapted the SAML SSO Protocol by removing the grey fields below:

Msg *1*. $A \rightarrow SP \;:\; A, SP, request$

Msg *2*. $SP \rightarrow A \;:\; A, IdP, ID, SP, request$

Msg *3*. $A \rightarrow IdP \;:\; A, IdP, ID, SP, request$

Msg *4*. $IdP \rightarrow A \;:\; ID, SP, IdP, \{ID, A, IdP, SP\}_{SK(IdP)}, request$

Msg *5*. $A \rightarrow SP \;:\; ID, SP, IdP, \{ID, A, IdP, SP\}_{SK(IdP)}, request$

Attacking SAML-based SSO for Google Applications

This change allows a dishonest service provider to obtain the authentication assertion from a run with Alice:

$$\begin{array}{lrcl}
\text{Msg } \alpha.1. & Alice \rightarrow BadSP & : & Alice, BadSP, request \\
\text{Msg } \alpha.2. & BadSP \rightarrow Alice & : & Alice, IdP, ID, BadSP, request \\
\text{Msg } \alpha.3. & Alice \rightarrow IdP & : & Alice, IdP, ID, BadSP, request \\
\text{Msg } \alpha.4. & IdP \rightarrow Alice & : & BadSP, \{Alice, IdP\}_{SK(IdP)}, request \\
\text{Msg } \alpha.5. & Alice \rightarrow BadSP & : & BadSP, \{Alice, IdP\}_{SK(IdP)}, request
\end{array}$$

and then to replay it while imitating Alice in a run with an honest service provider:[a]

$$\begin{array}{lrcl}
\text{Msg } \beta.1. & BadSP(Alice) \rightarrow SP & : & Alice, SP, request' \\
\text{Msg } \beta.2. & SP \rightarrow BadSP(Alice) & : & Alice, IdP, ID', SP, request' \\
\text{Msg } \beta.5. & BadSP(Alice) \rightarrow SP & : & SP, \{Alice, IdP\}_{SK(IdP)}, request'
\end{array}$$

[a] Attack found by Armando, Carbone, Compagna, Cuellar and Tobarro

Modelling and analysing layered architectures

The wrong way to model and analyse such layered architectures is to explicitly model the details of both the secure transport protocol and the application protocol.

The right way is to abstract away from the implementation of the secure transport protocol, and simply to model the services it provides to the application protocol.

We will formalise services that secure transport protocols might provide. We provide a number of building blocks, from which channels can be specified. We start with single messages, and then extend to sessions containing multiple messages.

Confidentiality

Many secure transport protocols aim to keep the application layer data confidential: the intruder will observe transport layer messages passing on the network, but will not be able to calculate application layer data from them.

More precisely, the set of application layer messages that the intruder can deduce from some trace are those that can be deduced (using the normal $\vdash$ relation) from:

- the messages he knew initially;
- the application layer messages that have (deliberately) been sent to him;
- the application layer messages that have been sent on non-confidential channels.

Faking

Many transport layer protocols prevent the intruder from faking messages (i.e., choosing an arbitrary application layer message m and sending it over the transport protocol). This will be the case when the application layer message m from A to B is encoded as, for example:

- $\{m\}_{shared(A,B)}$;
- $B, \{m\}_{SK(A)}$.

We say that the channel satisfies no faking (NF).

Redirecting

If application layer message m from A to B is encoded as the transport layer message $B, \{m\}_{SK(A)}$, then the intruder can change the transport layer message to $C, \{m\}_{SK(A)}$, thereby redirecting the application message.

Many transport layer protocols will prevent redirecting, for example if the encoding is $\{B, m\}_{SK(A)}$. We say that such channels satisfy no redirecting (NR).

However, if m is encoded as $B, \{\{m\}_{SK(A)}\}_{PK(B)}$, then the intruder can redirect it only if it is sent to him (when it is encrypted with his public key). We say that such channels satisfy no redirecting when sent to honest agents or no honest redirecting (NR^-).

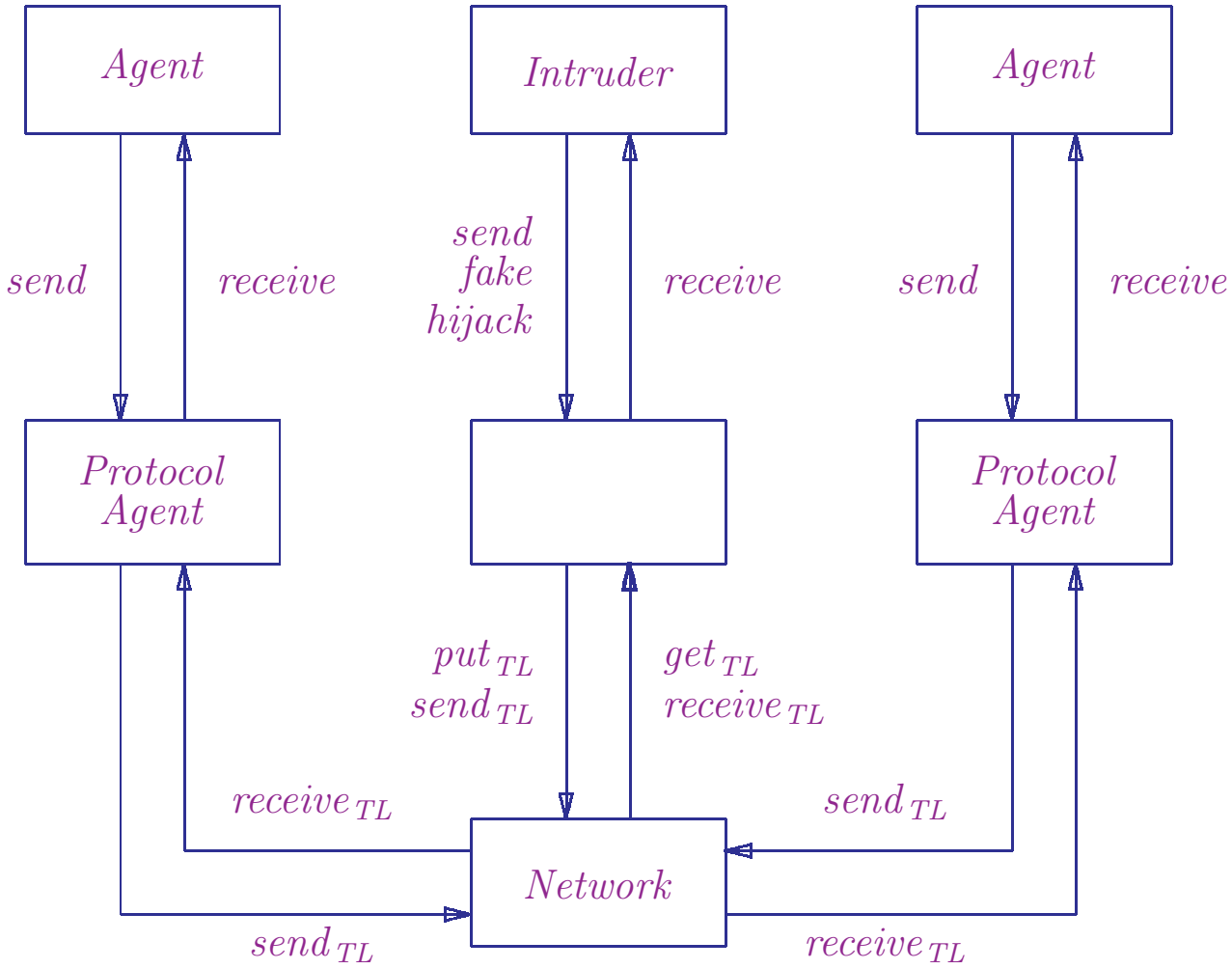
Re-ascribing

If application layer message m from A to B is encoded as the transport layer message $A, \{m\}_{PK(B)}$, then the intruder can change the transport layer message to $C, \{m\}_{PK(B)}$, thereby re-ascribing the application message.

Many transport layer protocols will prevent re-ascribing, for example if the encoding is $A, \{\{A, m\}_{PK(B))}\}_{SK(A)}$. We say that such channels satisfy no re-ascribing (NRA).

If application layer message m from A to B is encoded as the transport layer message $A, \{\{m\}_{PK(B)}\}_{SK(A)}$, then the intruder can re-ascribe it only to himself by sending $I, \{\{m\}_{PK(B)}\}_{SK(I)}$. We say that such channels satisfy no re-ascribing with honest identities or no honest re-ascribing (NRA^-).

Capturing properties of secure channels



Events

Assume a set of application layer roles (ranged over by R_A, etc.), and a set of connection identifiers (or handles) (ranged over by c_A, etc.).

All agents can:

- Send messages: $send.(A, R_A).c_A.(B, R_B).m$;
- Receive messages: $receive.(B, R_B).c_B.(A, R_A).m$.

The intruder can also:

- Create new messages: $fake.(A, R_A).(B, R_B).c_B.m$;
- Change sender/receiver fields in existing messages: $hijack.(A, R_A) \rightarrow (A', R_A).(B, R_B) \rightarrow (B', R_B).c_B.m$. This captures both redirecting and re-ascribing.

We can define network rules, e.g. to say that every *receive* is preceded by a corresponding *send*, *fake* or *hijack*.

Defining security properties

We can capture the no faking (NF), no re-ascribing (NRA), no honest re-ascribing(NRA$^-$), no redirecting (NR) and no honest redirecting (NR$^-$) by specifying that no messages of the appropriate form occur in any trace.

We can capture confidentiality (C) by specifying that if the intruder can deduce an application layer message m from the messages sent across the network, then he can deduce m from the messages sent to him and the messages sent on non-confidential channels.

We therefore have six building blocks from which we can build specifications for secure channels.

Combining the building blocks

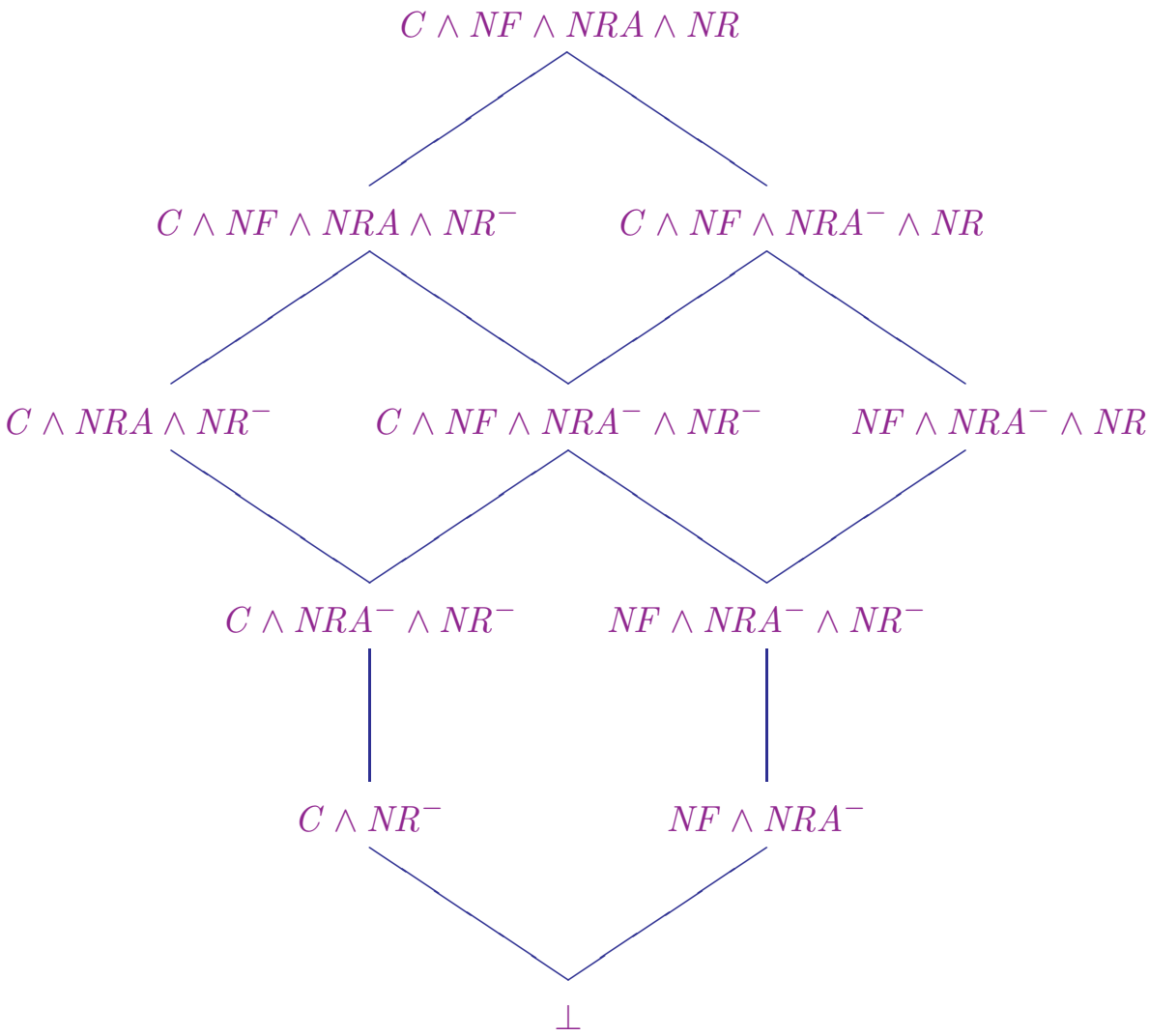
The building blocks are not independent: no-re-ascribing implies no-honest-re-ascribing, and likewise for no-redirecting.

Further, not all combinations are essentially different: certain pairs of combinations allow essentially the same intruder behaviours: we therefore collapse such combinations.

For example, a re-ascribable channel that prevents faking can simulate a channel that allows faking: the intruder can send messages with his own identity and then re-ascribe them; this activity simulates a fake.

Confidential channels that allow redirecting can simulate non-confidential channels: the intruder can redirect messages sent on them to himself, and so learn the messages.

The hierarchy of secure channels



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?

We say that a channel provides sender authentication if $NF \wedge NRA$.

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.

With unilateral TLS (i.e. the standard web model), the channel from the server to the client provides authentication of the server's identity, but not the client's.

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 say that a channel provides a guarantee of intent if *NR*: the recipient of a message knows that the sender intended him to receive it.

Intent can be achieved by encrypting a messages with the intended recipient's public key. With unilateral TLS, the channel from the client to the server provides a guarantee of the sender's (the client's) intent, but it does not provide authentication of the client's identity.

Strong authentication

We can combine the previous two properties so that whenever B receives a message from A, A previously sent that message to B.

We say that a channel provides strong authentication if it provides sender authentication and intent $NF \wedge NRA \wedge NR$.

We can achieve strong authentication by encoding m as $\{B, m\}_{SK(A)}$.

Also, bilateral TLS provides strong authentication.

Credit

When an agent B receives a message m from an authenticated agent A, 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.

A channel can be used to give credit if $C \wedge NRA \wedge NR^{-}$.

Credit can be achieved by encoding message m as $\{A, k\}_{PK(B)}, \{m\}_k$.

Responsibility

When an agent B receives a message m from an authenticated agent 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.

A channel can be used to assign responsibility if $NF \wedge NR \wedge NRA^{-}$.

Session channels

So far we have dealt only with properties of single messages. Many secure transport protocols group messages together into sessions.

We say that a channel is a session channel if all messages received in a single connection were sent in a single connection.

This can be achieved by adding a session identifier to each message, cryptographically bound to the application layer message. For example, the single-message protocol $\{\{m, B\}_{SK(A)}\}_{PK(B)}$ can be adapted to $\{\{m, B, s\}_{SK(A)}\}_{PK(B)}$.

Stream channels

A session channel allows the intruder to reorder messages within a connection.

We say that a channel is a stream channel iff all messages received in a connection were sent in a single connection, in the same order. The messages received will be a prefix of the messages sent.

This can be achieved by adding a sequence number to each message, cryptographically bound to the application layer message.

Independence of layers

We originally thought that a suitable transport layer protocol (probably SSL/TLS) would satisfy these properties regardless of the choice of application layer protocol. This turns out not to be true: we need to ensure that there are no interactions between the application layer protocol and the transport layer protocol that allow the intruder to replay messages from one protocol to another to his own advantage.

Independence of layers

One message (slightly simplified) of the SSL/TLS key-establishment phase is:

Msg *7*. $c \rightarrow s \ : \ \{p\}_{PK(s)}$.

p is the *premaster secret*; it is essential that p is kept secret.

Suppose the application layer protocol includes the following messages:

Msg *1*. $a \rightarrow b \ : \ \{x\}_{PK(b)}$
Msg *2*. $b \rightarrow a \ : \ x$.

Then the intruder could capture the above message 7 between some honest c and s; replay it as message 1 of the application layer protocol, in a session where he takes the role of a and s takes the role of b; and so learn the value of p.

TLS satisfies the secrecy and authentication properties

Theorem 2 Suppose the application layer protocol is restricted so that:

- No encrypted component can be replayed from one layer to another;
- The agent's long-term secret keys are not sent in the application layer;
- Values freshly generated for the application layer protocol are independent of those generated for SSL/TLS (i.e. nonces and premaster secrets).

Then bilateral TLS provides a confidential, strongly authenticated stream channel.

Analysis

How should we model and analyse a security application layered on top of a secure transport protocol?

In principal one could explicitly model both the high and low level protocols, using the Dolev-Yao model; but, it is clearly better to abstract away from the details of the low level protocol, and consider just the services it provides to the high level protocol.

Modelling the intruder

It is possible to adapt the CSP model of the intruder to capture the properties of the secure channels. For example, suppose the intruder overhears a message sent from A to B on some secure channel. Then the model ensures:

- if the channel satisfies no redirecting, then the intruder can only send the message to B;
- if the channel satisfies no re-ascribing, the intruder can only send the message so that it appears to come from A;
- if the channel provides confidentiality, the intruder cannot make deductions from the message.

Conclusions

- Many modern security architectures combine a (general-purpose) secure transport protocol with a (special-purpose) application layer security protocol.
- This leads to a cleaner design; it allows the designer to concentrate on the main goals of the security transaction without having to worry so much about issues of authentication and secrecy.
- What properties do these secure transport protocols provide to the application layer?
- How should we analyse these layered architectures?

Further and future work

- Chaining of secure channels.
- Layering of secure channels.
- Formalising the abstraction.
- Independence of layers and of different secure transport protocols.
- Design techniques using this methodology.