Security Protocols: Principles and Calculi

Martín Abadi

University of California, Santa Cruz, and
Microsoft Research, Silicon Valley

Based on work by myself, Bruno Blanchet, Véronique Cortier, Cédric Fournet, Andy Gordon, Roger Needham, and many others.

Contents

Security protocols, informally.

Some design principles.

A formal calculus for protocol analysis:

the applied pi calculus.

Automated proof methods.

Not:

- a course on cryptography,
- a balanced introduction to all topics in the field.

Security Protocols

Security protocols

Security protocols are concerned with properties such as integrity and secrecy.

- Primary examples are protocols that establish communication channels with authentication and confidentiality.
- Other examples include protocols for commerce and electronic voting.

In distributed systems, security protocols invariably rely on cryptography.

They have to be right, but they often aren't.

Cryptography

One-way hashing (e.g., SHA).

Shared-key cryptography (e.g., DES, AES, RC4):

- Two principals know a key.
- Both principals use the key both for encrypting and for decrypting.

Public-key cryptography (e.g., RSA):

- Each principal has a secret key, that it uses for signing and for decrypting.
- The inverse of the secret key is a public key, for checking signatures and for encrypting.

Bits of wisdom (?)

Cryptography is not broken, it is circumvented.

(Shamir according to Rivest)

Bits of wisdom (?) (cont.)

If you think that cryptography is the answer to your problem then you don't understand cryptography and you don't understand your problem.

(Needham or Lampson)

Communication with public-key cryptography

To send a message confidentially,

- the sender encrypts it with the recipient's public encryption key,
- the recipient decrypts it with its secret decryption key.

To send a message with integrity,

- the sender signs it with its secret signature key,
- the recipient checks it with the corresponding public key.

Note:

- For both, sign before encrypting.
- Encryption keys and signature keys should be different.

Communication with shared-key cryptography

If the sender and recipient share a key K:

To send a message confidentially,
- the sender encrypts it with K,
- the recipient decrypts it with K.

To send a message with integrity,
- the sender includes a MAC with K,
- the recipient checks the MAC with K.

Note:
- For both, the order matters, but is less clear.
- Encryption keys and MAC keys should be different.
- Each direction of communication may have its own keys.

Another twist

Informal descriptions often assume that encryption guarantees integrity.

In practice, encryption often does guarantee integrity (in particular, because of plaintext redundancies).

In theory, there is much progress on authenticated encryption schemes.

Complications

We may need:

- key identifiers,
- timestamps,
- sequence numbers,
- various other tags,
- compression,
- padding.

Problems

Remaining issues:

- performance
  (particularly for public-key methods),
- associating keys with principals
  (particularly for shared-key methods),
- access control, auditing, . . .

Authentication protocols

(or channel-establishment protocols)

There are many authentication protocols.

They typically involve:

- two principals

  hosts, users, services

- secrets (possibly shared)

  usually encryption keys

- encryption

  shared-key or public-key

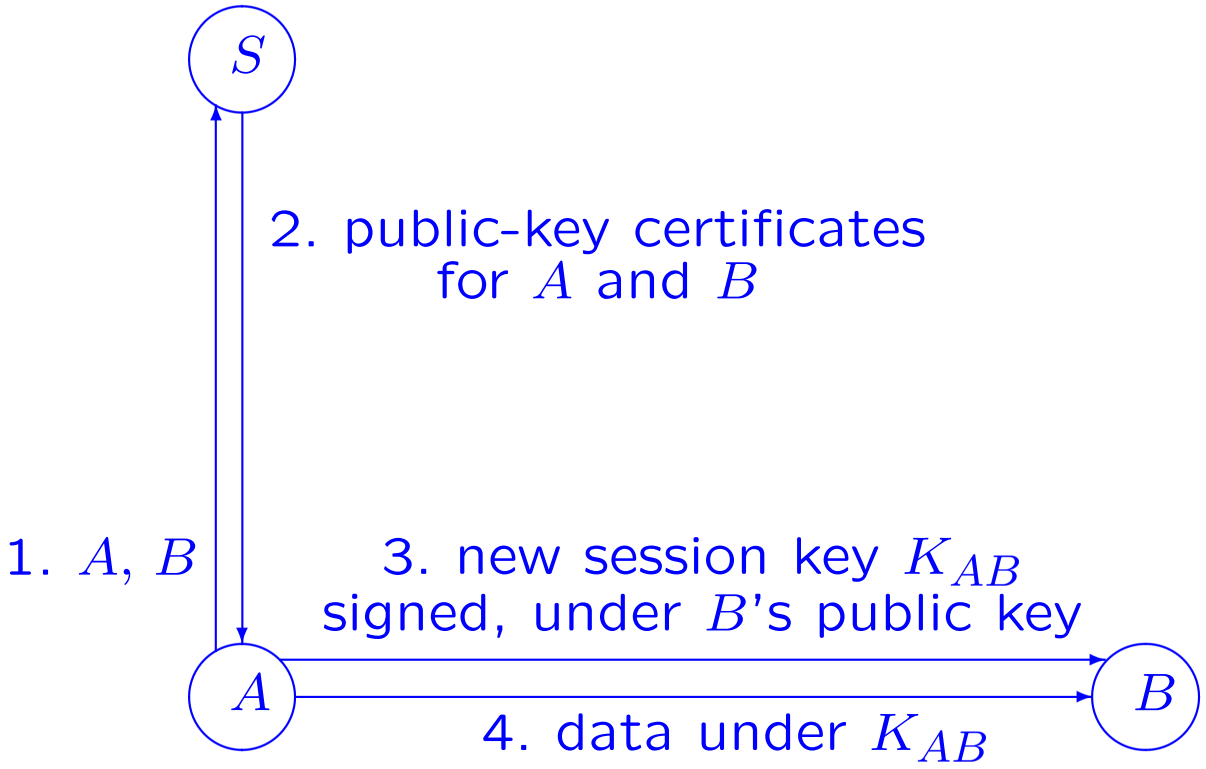
- trusted servers

- proofs of timeliness

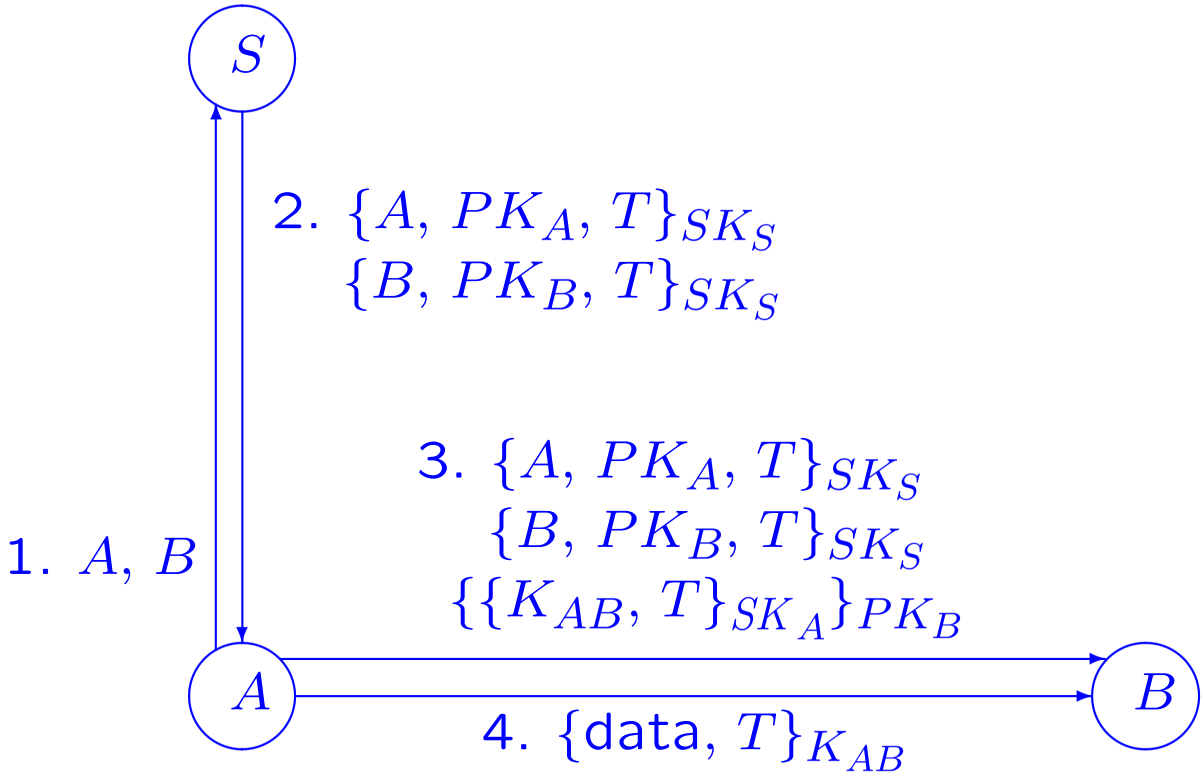
  nonces, timestamps

A typical protocol

- Each principal has a public-key pair.
- S is a certification authority for public keys.
- A and B agree on a session key K_{AB} and send data under K_{AB}:



The Denning-Sacco protocol



$\{X\}_{PK_B}$: X encrypted with B's public key, for secrecy

$\{X\}_{SK_A}$: X signed with A's secret key, for integrity

T: a timestamp

Questions

Does the protocol work?

What assumptions are we making?

Can we do better?

No single protocol will do.

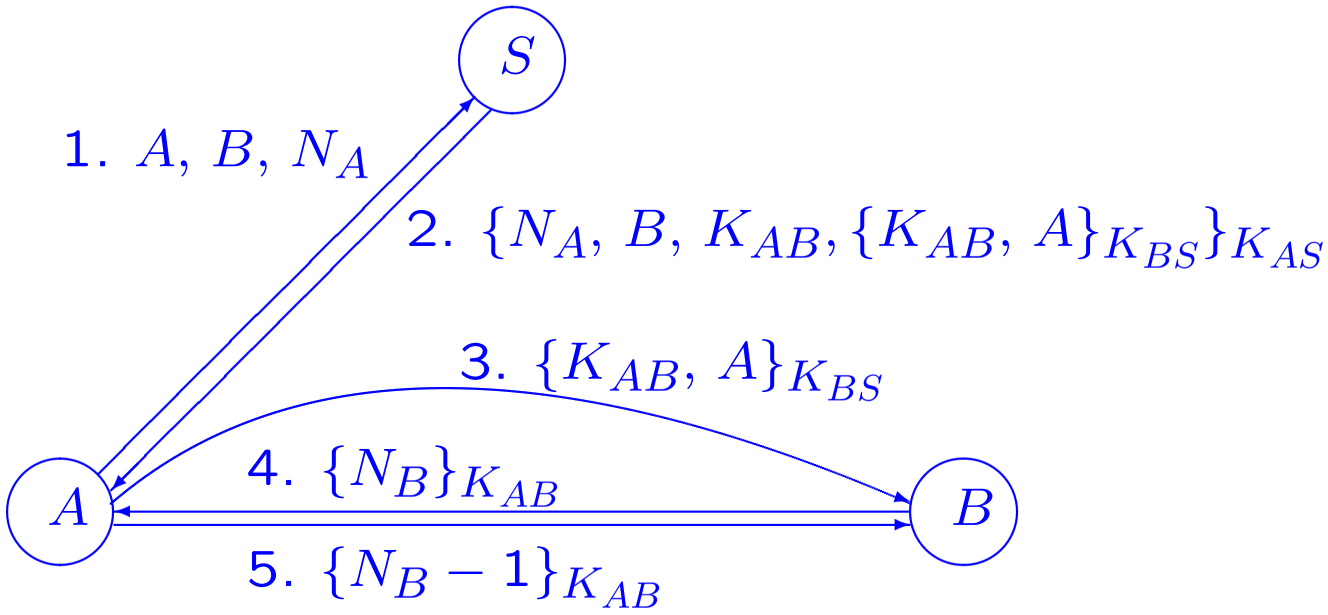
We may want:

- few messages,
- little encryption,
- little trust of other machines.

and also:

- asynchronous checking (for e-mail),
- different cryptosystems,
- little server state,
- one-way or two-way authentication,
- client anonymity.

The Needham-Schroeder shared-key protocol



$\{X\}_K$: X encrypted with the shared key K

N_A, N_B: nonces

K_{AB}: a session key for A and B, for encrypting subsequent communications

A criticism

Long after a run, an attacker may

– discover K_{AB},

– replay $\{K_{AB}, A\}_{K_{BS}}$ to B,

– conduct a handshake with B,

– send arbitrary data to B under K_{AB},
impersonating A. (Denning & Sacco)

Solutions:

- make K_{AB} strong, change K_{BS} often
- let B and S interact (Needham & Schroeder),
- use timestamps (Denning & Sacco, Kerberos).

Some observations

Most security protocols have subtleties and flaws.

Many of these have to do with cryptography.

Many of these don't have to do with the details of cryptography.

For design, implementation, and analysis, a fairly abstract view of cryptography is often practical.

Needham-Schroeder descendants

Initially, the Kerberos protocol was based on the Needham-Schroeder shared-key protocol plus timestamps.

(And an early version did not even check the timestamps properly!)

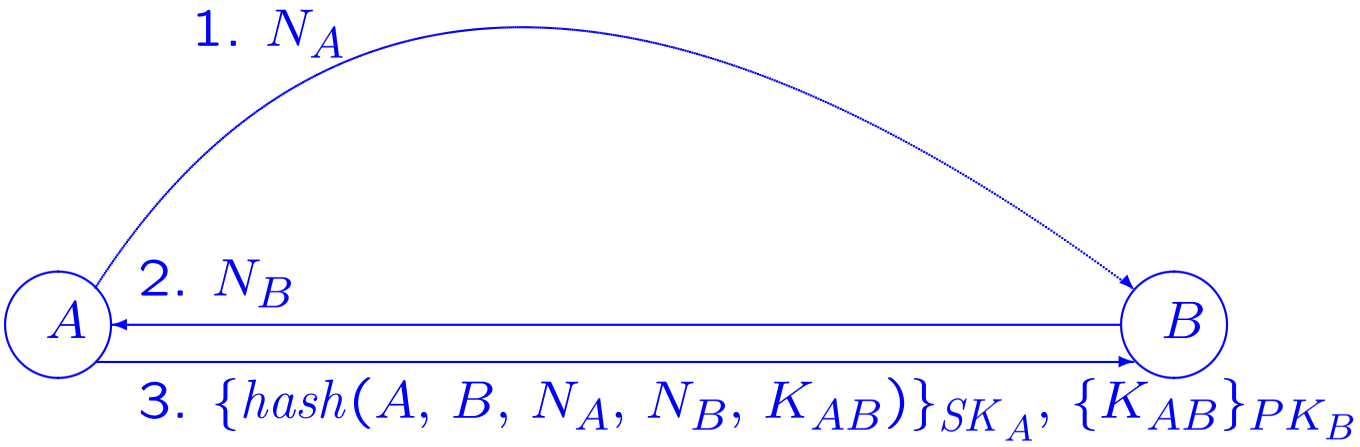
Over time, it incorporated improvements and extensions. For example:

- bootstrapping from passwords,
- cross-domain authentication (that is, several S's).

A variant is used in Windows 2000.

Netscape's SSL v3 protocol

(fragment, simplified)



$\{X\}_{PK_B}$: X encrypted with B's public key, for secrecy

$\{X\}_{SK_A}$: X signed with A's secret key, for integrity

K_{AB}, N_A, N_B are used in computing a session key

The full SSL (or TLS)

A more complete and complex protocol with a bulkier specification:

- Specifics of cryptographic operations.
- Several options:
  - various cryptographic algorithms,
  - protocol versions,
  - one-way or two-way authentication,
  - a fast mode.
- Option negotiation.
- Message transmission (“the record layer”).
- Alert messages.

Reading

A paper by Needham and Schroeder:
"Using encryption for authentication in large networks of computers"
in the ACM Digital Library.

The SSL protocol:

`http://wp.netscape.com/eng/ssl3/draft302.txt`

and/or

the TLS protocol:

`http://www.ietf.org/rfc/rfc2246.txt`

`http://www.ietf.org/rfc/rfc4346.txt`

Design Principles for Protocols

Design principles

We collected:

- some common-sense rules ($\simeq$ ten)
  - all fairly obvious
  - some general, some concrete
- published examples
  - largely wrong
  - as a proof of applicability

The principles seem generally useful for cryptographic protocols

- for authentication,
- for electronic commerce

⋮

Principle 1

Every message should say what it means: the interpretation of the message should depend only on its content.

It should be possible to write down an English sentence describing the content—though if there is a suitable formalism available that is good too.

An attack

The core of the Denning-Sacco protocol is:

$$A \rightarrow B : \ \{\{K_{AB}, T\}_{SK_A}\}_{PK_B}$$

A tells B that K_{AB} is a good key for A and B at time T.

An attack goes:

$$A \rightarrow C : \ \{\{K_{AC}, T\}_{SK_A}\}_{PK_C}$$

$$C \rightarrow B : \ \{\{K_{AC}, T\}_{SK_A}\}_{PK_B}$$

so any C may have the key that B believes shared with A. Then:

$$C \rightarrow B : \ \{\text{data}, T\}_{K_{AC}}$$

and B would believe that A sent data.

Diagnosis

Optimistic use of encryption.

Names are missing.

$\Rightarrow$ It is not possible to parse the message into the statement that represents its meaning.

Solution

$$A \rightarrow B: \ \{\{A \textit{ says } K_{AB} \ldots B \ldots T\}_{SK_A}\}_{PK_B}$$

or

$$A \rightarrow B: \ \{\{A, B, K_{AB}, T\}_{SK_A}\}_{PK_B}$$

or

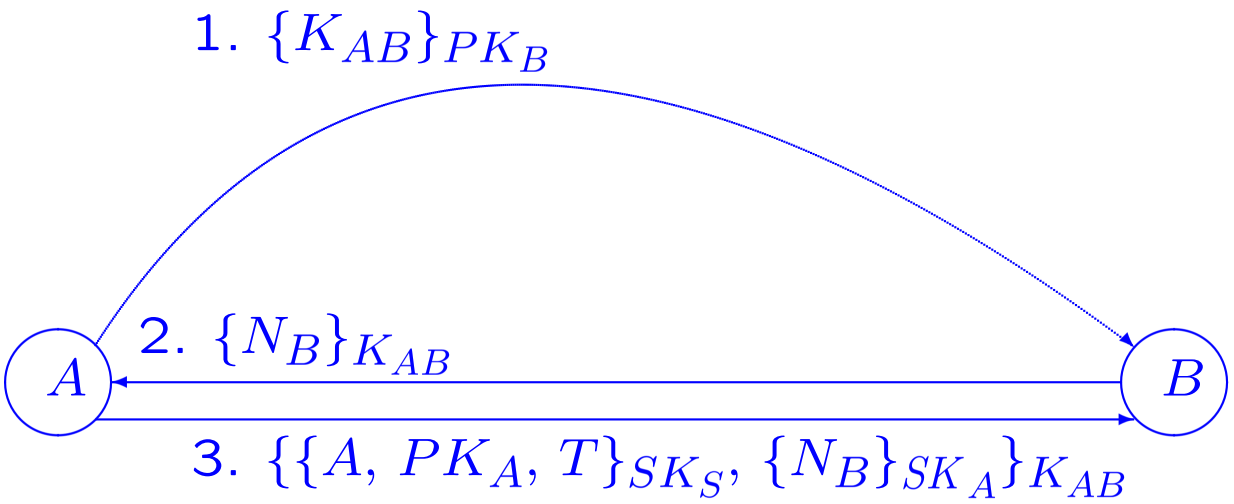
any other unambiguous encoding of the meaning of the message.

Principle 2

If the identity of a principal is important for the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

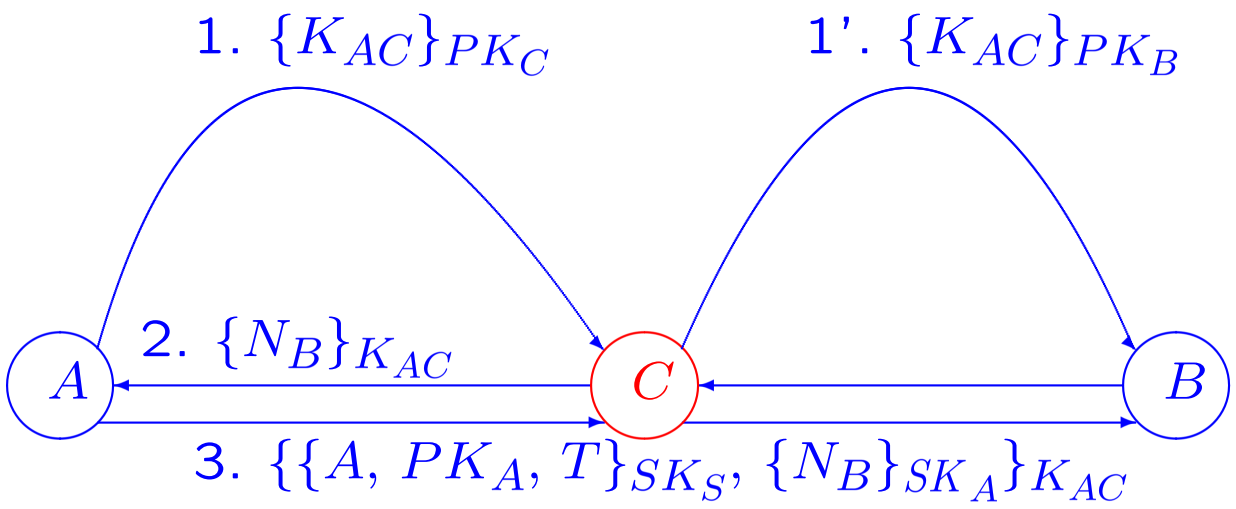
Netscape's SSL v1 protocol

RFC of Oct. 31, 1994 (simplified):

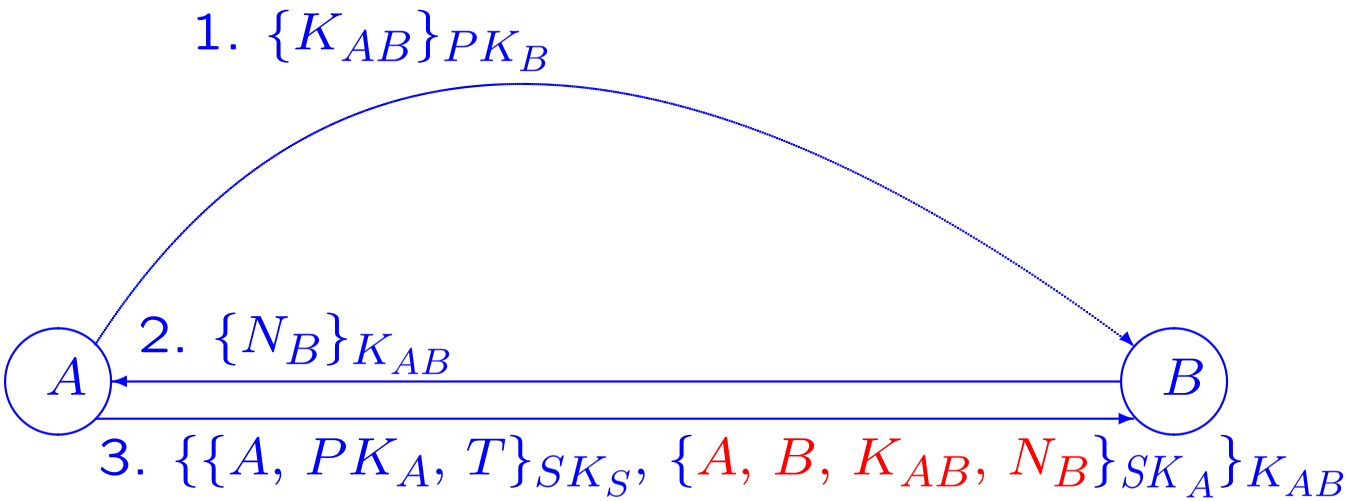


An attack

If A wants to talk to C, then C can impersonate A:

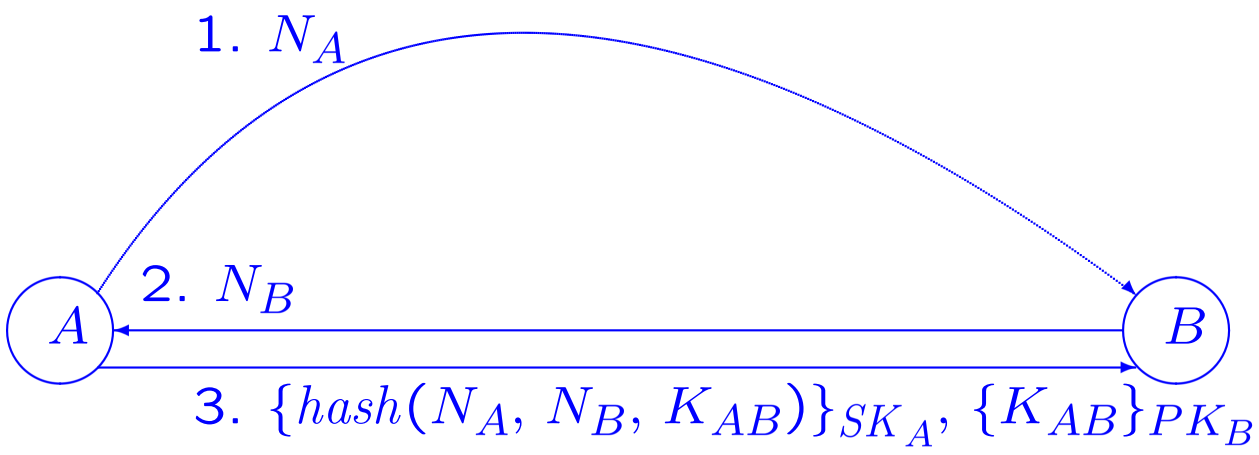


Solution



Netscape's SSL v3 protocol

Specification of Nov. 10, 1995 (simplified):



An analogous attack works.

SSH v2 protocol

(Internet draft of June 1996)

Option with public-key client authentication:

$$A \rightarrow B: \ N_A$$

$$B \rightarrow A: \ N_B$$

$$B \rightarrow A: \ PK_{Bh}, PK_{Bs}$$

$$A \rightarrow B: \ \{\{hash(previous\ \ msgs.), K\}_{PK_{Bs}}\}_{PK_{Bh}}$$

$$A \rightarrow B: \ \{A, PK_A, \{hash(A, N_A, N_B)\}_{SK_A}\}_{K'}$$

PK_{Bh} is a long-term host key.

PK_{Bs} is a shorter-term server key.

K' is a shared key derived from K, N_A, and N_B.

An analogous attack works again.

The Woo-Lam protocol

$$A \rightarrow B: \ A$$

$$B \rightarrow A: \ N_B$$

$$A \rightarrow B: \ \{N_B\}_{K_{AS}}$$

$$B \rightarrow S: \ \{A, \{N_B\}_{K_{AS}}\}_{K_{BS}}$$

$$S \rightarrow B: \ \{N_B\}_{K_{BS}}$$

A wants to talk to B.

B sends a random challenge N_B.

A responds with N_B encrypted with key K_{AS} shared by A and the server S.

B queries S, who translates to key K_{BS} between B and S.

An attack

Imagine a concurrent execution:

$$C \to B : \ A$$

$$C \to B : \ C$$

$$B \to A : \ N_B$$

$$B \to C : \ N'_B$$

$$C \to B : \ \{N_B\}_{K_{CS}}$$

$$C \to B : \ \{N_B\}_{K_{CS}}$$

$$B \to S : \ \{A, \{N_B\}_{K_{CS}}\}_{K_{BS}}$$

$$B \to S : \ \{C, \{N_B\}_{K_{CS}}\}_{K_{BS}}$$

$$S \to B : \ \{N_B\}_{K_{BS}}$$

$$S \to B : \ \textit{error or junk}$$

so any C can convince B that A is present.

Diagnosis

It is not possible to parse S's message into the statement that represents its meaning:

A's name is missing.

Nonces are good for ensuring freshness
but not always association.

Double encryption is no cause for optimism.

Solution

$$A \rightarrow B: \ A$$

$$B \rightarrow A: \ N_B$$

$$A \rightarrow B: \ \{N_B\}_{K_{AS}}$$

$$B \rightarrow S: \ A, \{N_B\}_{K_{AS}}$$

$$S \rightarrow B: \ \{A \textit{ said } N_B\}_{K_{BS}}$$

S is explicit.

N_B is used only for freshness.

No double encryption.

Other principles concern

- encryption
- timeliness
- trust
- secrecy

The principles serve to

- simplify protocols
- simplify formal analysis
  (see Paulson's work with Isabelle,
  Blanchet's with ProVerif)
- avoid many mistakes

Principle 3

Be clear as to why encryption is being done.

Encryption is not synonymous with security.

Principle 4

Be clear as to what properties you assume of nonces.

What may do for ensuring timeliness may not do for ensuring association—and perhaps association is best established by other means.

Principle 5

The protocol designer should know which trust relations the protocol depends on.

Principles for secrecy

Information-flow ideas lead to other principles for secrecy:

- Classify data, channels, and keys into levels.
  - Secret data encrypted under secret keys may be made public.
- Do not send secret data on public channels (and also avoid implicit information flows).
- Distinguish secret inputs from public inputs.

Principles for secrecy (fuzzier)

- Secrets should be strong enough for the data that they protect.
  - Beware of weak passwords.
  - Change and diversify keys.
- Do not expect attackers to obey rules.
- Do not underestimate the initial knowledge or the observations of attackers.

Staying out of trouble

Keep your protocols simple.

Be suspicious of clever optimizations.

Be explicit:

- include sufficient proofs of freshness,
- include sufficient names,
- do not count on context,
- use evident classifications.

Interpreting a message should be a simple matter of parsing.

Staying out of trouble (cont.)

Cryptography helps, but it is not the whole story.

We can and should go beyond patching security holes one by one.

Security Protocols and Specifications (Generalities)

Protocol specifications

"Security by obscurity" hides specifications.

Open design is often superior.

Specifications of security protocols abound. They vary in

– quality

– scope

– purpose

– vocabulary

There are at least two kinds:

– step-by-step narrations ("bits on the wire")

– properties (e.g., secrecy)

Protocol narrations

Typical protocol narrations combine lots of prose, a few bubble diagrams, ad hoc notations, and message sequences like:

Message 3	$A \rightarrow B:$	$\{K_{AB}, A\}_{K_{BS}}$
Message 4	$B \rightarrow A:$	$\{N_B\}_{K_{AB}}$
Message 5	$A \rightarrow B:$	$\{N_B - 1\}_{K_{AB}}$

They often include an informal security analysis.

Communication model

What does "Message n $X \rightarrow Y : M$" mean?

> We assume that an intruder can interpose a computer in all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. [. . .]
>
> We also assume that each principal has a secure environment in which to compute, such as is provided by a personal computer or would be by a secure shared operating system.
>
> (Needham and Schroeder)

Model of cryptographic operations

Sometimes, the description of cryptographic operations is fairly informal or abstract.

Often, encryption is assumed (implicitly or explicitly) to guarantee more than secrecy. In particular:

- $\{M\}_K$ cannot be produced by anyone who does not know M and K.

  Thus, encryption is not malleable.

  $$\{N_B\}_K \text{ vs. } \{N_B - 1\}_K$$

- Decryption with an incorrect key is evident.

Other ambiguities

- Which pieces of data are known in advance and which are freshly generated.

- Checking of messages.

- Ordering of messages.

- Concurrent executions and roles.

- Properties of nonces.

- Objectives, trust relations, . . .

A mundane ambiguity

> The simplest fix is to require that a SSL implementation receive a **change cipher spec** message before accepting a finished message. (Indeed, there is some language in the specification which could be interpreted to mandate this restriction, although it is not entirely clear.) [. . .] at least one implementation has fallen for this pitfall.
>
> (Wagner and Schneier)

We can do better!

Design and analysis methods

There are now several methods for describing and verifying security protocols, based on:

- rigorous but informal frameworks,
- temporal logics,
- process algebras (CSP, the pi calculus),
- theorem-proving tools,
- special-purpose formalisms.

They have resulted in:

- increased confidence in some protocols,
- discovery of many protocol flaws,
- a better understanding of how to design robust protocols.

Protocol objectives

Narrations need not specify the meaning or the purpose of messages.

Some standard protocol properties are:

– authenticity (known message origin)

– secrecy (known message destination)

and also:

– forward secrecy (resilience to future leak)

– anonymity

– service availability

– non-repudiation

– deniability

As usual, there is no precise, unique definition of security.

Common themes

- Interaction with an uncertain environment.

  (Contrast with mutual exclusion.)

- Some security even against lucky, powerful, or persistent attackers.

  – Even if the attacker controls the network.

  – Even if a session key is compromised.

  (See Needham-Schroeder, SSL.)

  – Even if an insider is dishonest.

  (See Denning-Sacco, SSL, SSH.)

- Doing without full functional correctness.

  E.g., message origin, not message correctness.

Specifying the Denning-Sacco protocol

We expect the following properties:

- Secrecy:

  Only A and B know data.

- Authenticity:

  If B is convinced that data is from A,

  then it is.

These depend on auxiliary properties about K_{AB}.

Some good news

Usually, a property is a predicate on behaviors.

Authenticity may be phrased as a property.

We can turn

"B is convinced that data is from A"

into a statement about B's actions,

e.g., "B writes data to a file".

A difficulty

But this is a property of a system in which there is an active attacker.

The set of all behaviors of the system has transitions for A, B, S, and an attacker.

It is hard to model the attacker (at least without complexity theory):

- It can choose random numbers.
- It cannot get too lucky in guessing keys.

Another difficulty

Secrecy is not a standard property.

Secrecy is not true or false of a behavior, but of a set of behaviors.

Formal Methods for Security Protocols

Design and analysis methods

(reminder)

There are now several methods for describing and verifying security protocols, based on:

- rigorous but informal frameworks,
- temporal logics,
- process algebras (CSP, the pi calculus),
- theorem-proving tools,
- special-purpose formalisms.

Protocol descriptions

We may specify a protocol as a process, or as the corresponding set of behaviors.

In addition to the expected participants, we should model an attacker, who:

- may participate in some protocol runs,
- knows some data in advance,
- may intercept messages on the public network,
- may inject any messages that it can produce into the public network.

The origin of formal cryptology

What messages can the attacker produce?

- The attacker can be non-deterministic.
- But it cannot get too lucky in guessing keys.

A simple solution:

- Arrange that non-deterministic choice of a key always yields a fresh key.
- Distinguish keys from bitstrings.
- Treat cryptographic operations symbolically.

Common assumptions

Some informal assumptions commonly underly formal methods, e.g., for symmetric encryption:

- Given K, anyone can compute $\{M\}_K$ from M and vice versa.
- $\{M\}_K$ cannot be produced by anyone who does not know M and K.
- $\{M\}_K$ cannot be understood by anyone who does not know K.
- Decryption with an incorrect key is evident.

The formal view: strengths

An often appropriate level of abstraction.

- Some simple, effective intuitions.
- Easy human reasoning (sometimes).
- "Enough" interesting detail.

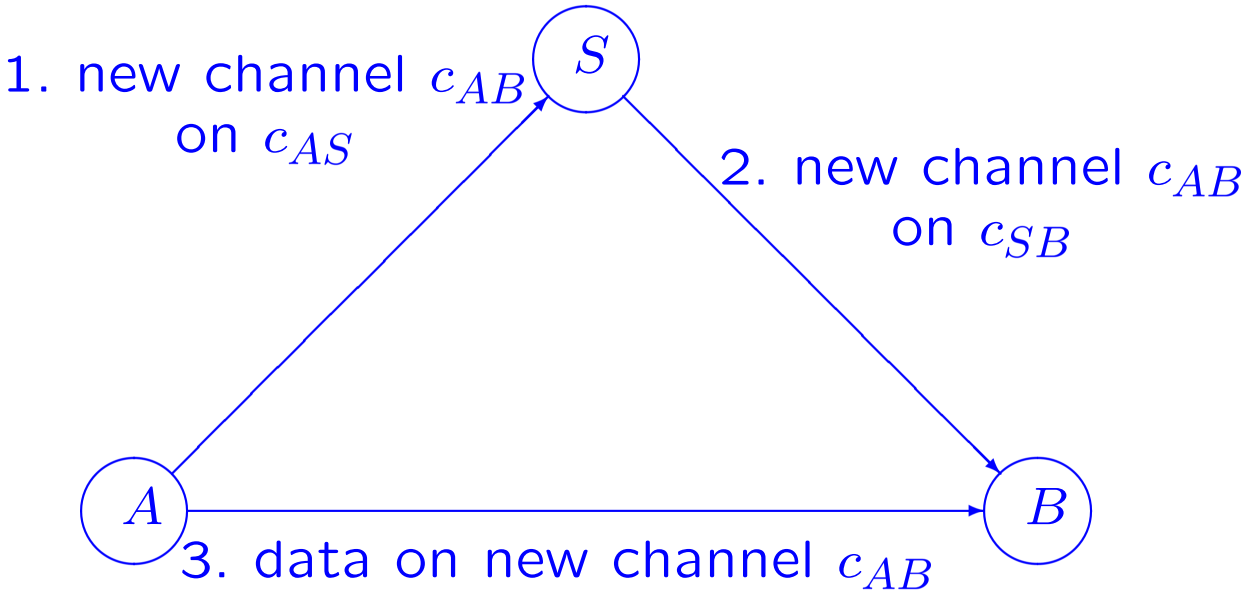
Techniques and tools for automated reasoning.

Logical methods and foundations
(in modal logics, process algebras, . . .).

The Applied Pi Calculus

A typical protocol

Establishment and use of a secure channel:



A and B: two clients S: an authentication server

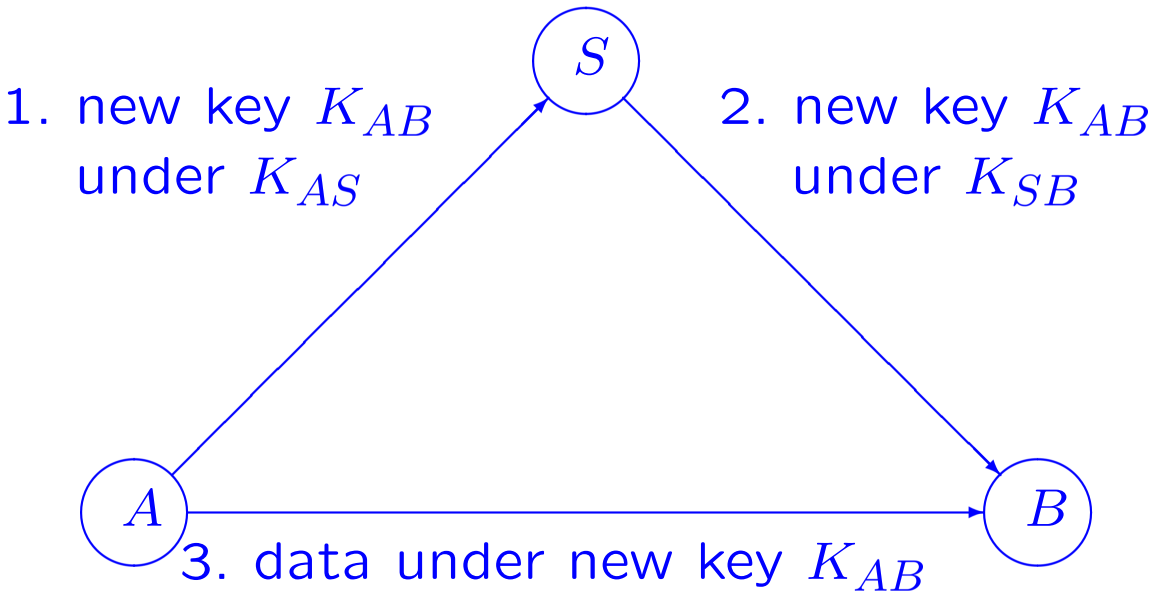
c_{AS} and c_{SB}: channels with the server

c_{AB}: a new channel for the clients

The pi calculus should help with such protocols.

A typical protocol, in more detail

Establishment and use of a secure channel:



A and B: two clients S: an authentication server

K_{AS} and K_{SB}: shared keys with the server

K_{AB}: a new session key for the clients

The applied pi calculus

(joint work with C. Fournet)

applied pi calculus = pi calculus + function symbols

The pi calculus part is enough

- for general programming,
- for manipulating secure channels abstractly.

The functions enable us to show more detail, e.g.,

- constructing a secure channel by encryption,
- deriving a key from another key.

Syntax of a pi calculus

We start out with a sort of variables (x) and a sort of names (n).

We define a sort of terms (data):

$$
\begin{array}{lll}
M, N & ::= & \text{terms} \\
& \mid \quad x & \text{variable} \\
& \mid \quad n & \text{name}
\end{array}
$$

We let u range over variables and names.

Syntax of a pi calculus (cont.)

We also define a sort of processes:

$P, Q ::=$		processes
	nil	nil process
|	$\overline{u}\langle N \rangle.P$	sending
|	$u(x).P$	receiving
|	$!P$	replication
|	$P \mid Q$	parallelism
|	$(\nu n)P$	restriction

nil may be omitted.

Secrecy from scoping

$$
\begin{aligned}
A(M) &\triangleq \overline{c}\langle M\rangle \\
B &\triangleq c(x).nil \\
P(M) &\triangleq (\nu c)(A(M) \mid B)
\end{aligned}
$$

P represents a protocol where

- A sends M to B over a channel c,
- B swallows its input,
- communication is secure: only A and B can access c.

Secrecy as equivalence

We obtain a secrecy property:

$P(M)$ and $P(N)$ are equivalent,
for all M and N.

- Almost any definition of equivalence (e.g., bisimilarity) will do in this example.

- The equivalence should mean that no attacker can distinguish $P(M)$ and $P(N)$.

- The restriction (νc) is crucial: without it, an attacker $c(x)\ldots$ could get the message.

Secrecy: an alternative formulation

For the special case where M is a name n:

No attacker can learn n from $P(n)$:

> for all Q in which n does not occur,
> Q will not get n from $P(n)$ in $P(n) \mid Q$.

Equivalently (but more precisely):

> for all Q in which n does not occur free,
> $P(n) \mid Q$ will never output n.

Initial comments

We can write down some protocols.

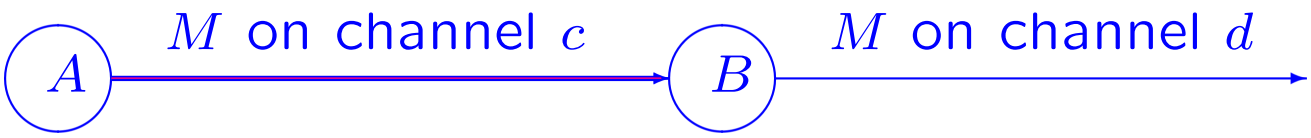
We can express some security properties, using tools from process calculi.

We can model an (arbitrary) attacker as a process Q.

This process runs in parallel with our protocol, and may interact with it.

Authenticity from scoping

Here B forwards its input over a public channel d.



$$
\begin{aligned}
A(M) &\triangleq \overline{c}\langle M\rangle \\
B &\triangleq c(x).\overline{d}\langle x\rangle \\
P(M) &\triangleq (\nu c)(A(M) \mid B)
\end{aligned}
$$

An authenticity property

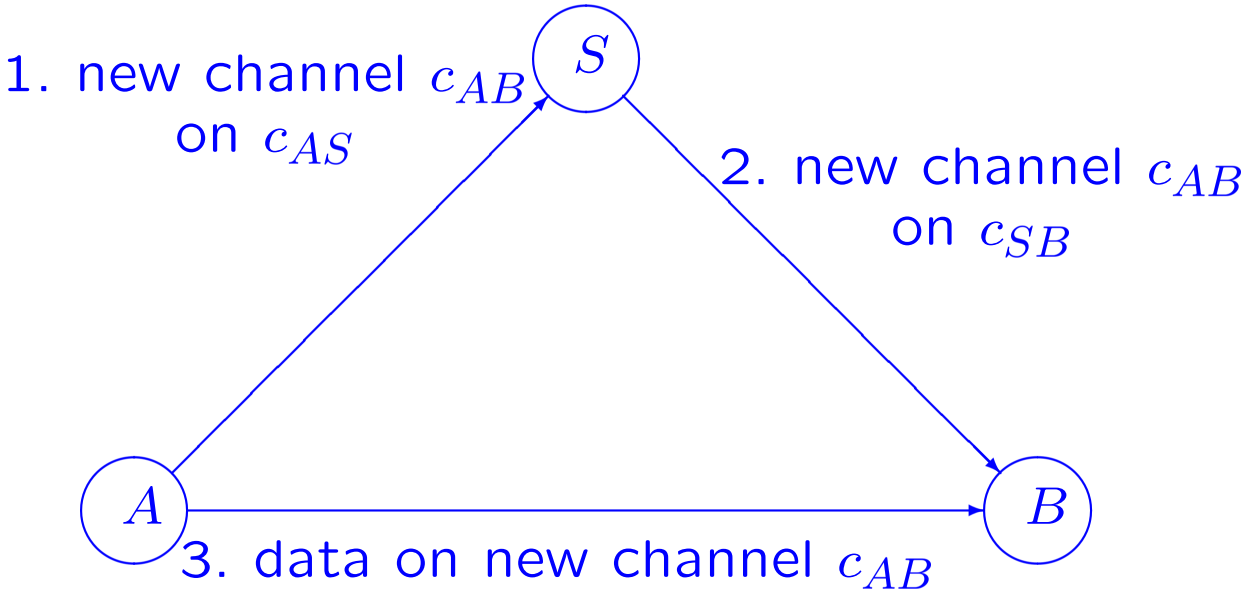
We obtain an authenticity property:

for all N,
if B outputs N on d
then A sent N to B.

This is an ordinary safety property.

(Authenticity can also be paraphrased as an equivalence.)

Passing channels: a typical protocol, revisited



An abstract version of the protocol:

$$
\begin{aligned}
A(M) &\triangleq (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle \\
S &\triangleq c_{AS}(x).\overline{c_{SB}}\langle x\rangle \\
B &\triangleq c_{SB}(x).x(y).\overline{d}\langle y\rangle \\
P(M) &\triangleq (\nu c_{AS})(\nu c_{SB})(S \mid A(M) \mid B)
\end{aligned}
$$

- A makes up a channel c_{AB} and sends it to server S on preexisting channel c_{AS},
- then S forwards c_{AB} to B on preexisting channel c_{SB},
- then A sends M to B on c_{AB},
- and finally B outputs its input on d.

We can add concurrent sessions.

For example:

$$
\begin{aligned}
A(M) &\triangleq (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle \\
S &\triangleq !c_{AS}(x).\overline{c_{SB}}\langle x\rangle \\
B &\triangleq !c_{SB}(x).x(y).\overline{d}\langle y\rangle \\
P(M_1,\ldots,M_n) &\triangleq (\nu c_{AS})(\nu c_{SB})(S \mid A(M_1) \mid \ldots \mid A(M_n) \mid B)
\end{aligned}
$$

The pi calculus and security

The channels of the pi calculus have nice built-in properties, such as:

- integrity,
- confidentiality,
- exactly-once semantics,
- mobility,
- forward secrecy.

These properties are useful in protocol descriptions.

The applied pi calculus retains these properties.

Toward the applied pi calculus

We wish to express cryptographic operations.

- Interestingly, we can encode some forms of encryption in the pi calculus.
- In the end, we added primitives for certain operations.

This resulted in the spi calculus (work with A. Gordon).

The applied pi calculus is a more general extension, with a generic treatment of function symbols.

Applied lambda calculus

We start out with

- a sort of variables (x),
- a set of function symbols, such as f and pair,
  each with an arity.

We may give types and equations for the functions.

We define the sort of terms:

M, N	::=		terms
	|	x	variable
	|	$\lambda x.M$	abstraction
	|	$M(N)$	application
	|	$f(M_1, \ldots, M_k)$	function application

Syntax of the applied pi calculus

We start out with

- a sort of variables (x),
- a sort of names (n),
- a set of function symbols, such as f, encrypt, and pair, each with an arity.

We define the sort of terms (data):

M, N	::=		terms
	|	x	variable
	|	n	name
	|	$f(M_1, \ldots, M_k)$	function application

Syntax (cont.): processes

We define the sort of processes:

$P, Q ::=$	processes
$\mid \; nil$	nil process
$\mid \; \overline{u}\langle N \rangle.P$	sending
$\mid \; u(x).P$	receiving
$\mid \; !P$	replication
$\mid \; P \mid Q$	parallelism
$\mid \; (\nu n)P$	restriction
$\mid \; \mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q$	conditional

Syntax (cont.): types for terms

A simple type system for terms:

- a set of base types, such as Integer, Key,
  or simply a universal base type Data,
- a type Channel$\langle\tau\rangle$ for those channels that convey
  messages of type τ.

For simplicity, function symbols take arguments and produce results of the base types only.

Equations

Given a signature Σ, we equip it with an equational theory, that is, with an equivalence relation on terms with some closure conditions.

We write $\Sigma \vdash M = N$ when the equation $M = N$ is in the theory associated with Σ.

We write $\Sigma \nvdash M = N$ for the negation of $\Sigma \vdash M = N$.

Examples: pairs

Σ includes pair, fst, and snd, with the equations:

$$\mathsf{fst}(\mathsf{pair}(x, y)) = x$$
$$\mathsf{snd}(\mathsf{pair}(x, y)) = y$$

We may write (M, N) for $\mathsf{pair}(M, N)$.

Examples: shared-key encryption

Σ includes the binary symbols senc and sdec, with the equation:

$$\mathsf{sdec}(\mathsf{senc}(x, y), y) = x$$

Dealing with errors

Types cannot statically avoid all "errors"'.

We may add a constant wrong, with equations such as:

$$\mathsf{sdec}(M, N) = \mathsf{wrong}$$

for ground terms M and N such that $\Sigma \not\vdash M = \mathsf{senc}(M', N)$ for any M'.

We may then code derived constructs, such as:

$$\mathit{case}\ N\ \mathit{of}\ \{x\}_M\ \mathit{in}\ P \triangleq$$
$$\mathit{if}\ \mathsf{sdec}(N, M) = \mathsf{wrong}\ \mathit{then}\ \mathit{nil}\ \mathit{else}\ P[\mathsf{sdec}(N, M)/x]$$

(This is the construct for decryption in the spi calculus.)

Dealing with errors (cont.)

An alternative is to add checking constructs, as in:

$$\mathsf{scheck}(\mathsf{senc}(M', N), N) = \mathsf{ok}$$

Then we may leave $\mathsf{sdec}(M, N)$, for M not of the form $\mathsf{senc}(M', N)$, without any equations.

Checking constructs also help elsewhere, for example in checking signatures.

Examples: public-key encryption

Σ includes unary function symbols pk and sk for generating public and secret keys from a seed, and the equation:

$$\mathsf{pdec}(\mathsf{penc}(x, \mathsf{pk}(y)), \mathsf{sk}(y)) = x$$

For simplicity, we may omit sk.

Optionally, we may for example add:

$$\mathsf{pdec}(\mathsf{penc}(x, y), z) = \mathsf{penc}(\mathsf{pdec}(x, z), y)$$

Examples: one-way hashing

Σ includes a unary symbol hash, with no equations.

hash(M) represents the result of applying a one-way hash function to M.

Other easy examples

Nondeterministic ("probabilistic") encryption.

Digital signatures.

Message authentication codes.

Exponentiation as used in Diffie-Hellman.

XOR.

⋮

Operational semantics

Structural equivalence $\equiv$ is defined much as usual.

It is also closed by substitution of equal terms.

So:

$$
\begin{array}{rcll}
P \mid nil & \equiv & P & \\
P \mid Q & \equiv & Q \mid P & \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \\
!P & \equiv & P \mid !P & \\
(\nu m)(\nu n)P & \equiv & (\nu n)(\nu m)P & \\
(\nu n)nil & \equiv & nil & \\
(\nu n)(P \mid Q) & \equiv & P \mid (\nu n)Q & \text{if } n \notin fn(P) \\
P[M/x] & \equiv & P[N/x] & \text{if } \Sigma \vdash M = N
\end{array}
$$

Operational semantics (cont.)

Reduction $\rightarrow$ is the smallest relation on closed processes that is closed by structural equivalence and application of evaluation contexts such that:

$$\overline{a}\langle M\rangle.P \mid a(x).Q \;\rightarrow\; P \mid Q[M/x]$$

$$\mathit{if}\ M = M\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; P$$

$$\mathit{if}\ M = N\ \mathit{then}\ P\ \mathit{else}\ Q \;\rightarrow\; Q$$

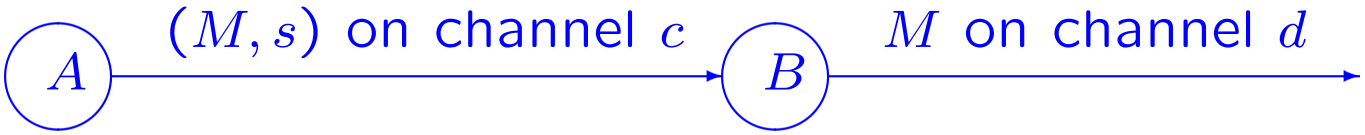
for any ground terms M and N such that $\Sigma \nvdash M = N$

Names as data

In the pi calculus, channels can be dynamically created and communicated.

In the applied pi calculus, keys, nonces, and other names can be dynamically created and communicated as well.

Names as data: example



$$
\begin{aligned}
A(M) &\triangleq \overline{c}\langle (M, s)\rangle \\
B &\triangleq c(x).\mathit{if}\ \mathsf{snd}(x) = s\ \mathit{then}\ \overline{d}\langle \mathsf{fst}(x)\rangle \\
P(M) &\triangleq (\nu s)(A(M) \mid B)
\end{aligned}
$$

P represents a protocol where

- A sends M to B tagged with s over a public channel c,
- then, if the tag matches, B outputs its input on another channel d.

So s serves as a capability, but can be intercepted.

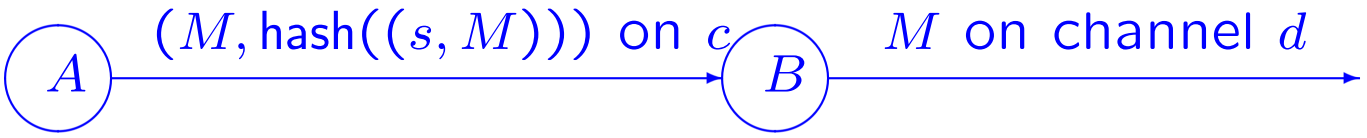
Interesting extrusions

In the applied pi calculus, a process may reveal a term that contains a fresh name s without revealing s itself.

This possibility does not arise in the pure pi calculus.

It is a source of expressiveness and complications.

Interesting extrusions: example

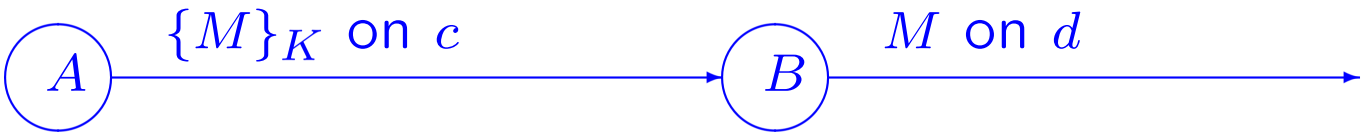


$$
\begin{aligned}
A(M) &\triangleq \overline{c}\langle(M, \mathsf{hash}((s, M)))\rangle \\
B &\triangleq c(x).\mathit{if}\ \mathsf{hash}((s, \mathsf{fst}(x))) = \mathsf{snd}(x)\ \mathit{then}\ \overline{d}\langle\mathsf{fst}(x)\rangle \\
P(M) &\triangleq (\nu s)(A(M) \mid B)
\end{aligned}
$$

Although $(M, \mathsf{hash}((s, M)))$ travels on the public channel c, no other process can extract s from this, or produce $(N, \mathsf{hash}((s, N)))$ for some other N.

So only the intended term M will be forwarded on d.

Another variant



$$
\begin{aligned}
A(M) &\triangleq \overline{c}\langle \mathsf{senc}(M, K)\rangle \\
B &\triangleq c(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_K\ \mathit{in}\ \overline{d}\langle y\rangle \\
P(M) &\triangleq (\nu K)(A(M) \mid B)
\end{aligned}
$$

Bigger examples (sketch)

Suppose we have n clients plus a server, and m instances $I_1, \ldots, I_m$ (sender, receiver, message). We may write:

$$
\begin{aligned}
Send(i,j,M) &\triangleq \overline{c_S}\langle i\rangle \mid \\
&\quad c_i(x_{nonce}). \\
&\quad (\nu K_{AB})(\overline{c_S}\langle \mathsf{senc}((i,i,j,K_{AB},x_{nonce}),K_{iS})\rangle \mid \\
&\quad \overline{c_j}\langle (i,\mathsf{senc}(M,K_{AB}))\rangle \\
Recv(j) &\triangleq \ldots \overline{d}\langle y\rangle \\
S &\triangleq \ldots \\
Sys(I_1,\ldots,I_m) &\triangleq (\nu K_{1S})\ldots(\nu K_{nS}) \\
&\quad (\nu K_{S1})\ldots(\nu K_{Sn}) \\
&\quad (Send(I_1) \mid \cdots \mid Send(I_m) \mid \\
&\quad !S \mid \\
&\quad !Recv(1) \mid \cdots \mid !Recv(n))
\end{aligned}
$$

Again:

- Scoping is the basis of security.
- We can discuss security properties formally.

The notion of equivalence becomes delicate:

$(\nu K)\overline{c}\langle \mathsf{senc}(M, K)\rangle$ and $(\nu K)\overline{c}\langle \mathsf{senc}(N, K)\rangle$ send different messages,

but they should be equivalent.

This is important for getting a sensible criterion for security properties.

Defining equivalence

Two processes are equivalent if no environment can distinguish them.

Technically, we use a testing equivalence or observational congruence.

Defining equivalence (cont.)

A definition is:

A test is a process R plus a channel name w.

Intuitively, R is the environment and w is used for signaling the test outcome.

A process P passes a test (R, w) if $P \mid R$ may communicate on w.

Two processes are equivalent if they pass the same tests.

Observational congruence pays attention to branching structure, in addition.

Features of these equivalences

Equivalence captures observational indistinguishability, with respect to an implicit, arbitrary attacker process.

Equivalence is "coarse enough".

When an equivalence fails, we can often find an attack.

Equivalence is a congruence.

Equivalence can be hard to prove! It is useful to develop proof techniques.

Using the applied pi calculus

Like the spi calculus, the applied pi calculus is rather abstract.

- We are able to ignore details.
- In particular, we ignore details of encryption.

Using the applied pi calculus (cont.)

The applied pi calculus is also expressive.

- We can represent process behavior and cryptography.
- We need not commit to a fixed cryptosystem.
- We can describe:

  every message,

  how it is acted on, and

  under what circumstances it is sent.

It allows expressing and proving security properties.

It may serve as a basis for higher-level reasoning.

Further developments

Study of equivalences.

Type systems.

Decision procedures for some problems.

Automated analysis via logic programs.

Other proof techniques and tools.

Partial complexity-theoretic foundations.

Examples.

Some logics.

Automated Protocol Analysis

The state of the art

By now there are a variety of methods for protocol analysis that rely at least in part on automatic tools.

They are effective on abstract but detailed models of important protocols.

Some of them employ elaborate proof techniques —some general, some specific to this area.

We will cover only a fraction of this work.

Decision procedures

Going back to Dolev and Yao, there has been much work on special decision procedures.

In recent years, these have been most successful for finite-state systems.

An important example: deduction

Static knowledge (of a participant or attacker):

$$\nu\tilde{n}.\{M_1/x_1, \ldots, M_l/x_l\}$$

where

- $\tilde{n}$ is a list of names
  (fresh, not known to the attacker a priori),
- M_1, ..., M_l are terms
  (known to the attacker),
- x_1, ..., x_l are variables
  (which can be used to refer to the terms).

Example: $\nu k.\{\mathsf{senc}(M, k)/x_1, k/x_2\}$

Someone with this knowledge can deduce $\mathsf{sdec}(x_1, x_2)$,
which equals M if $\mathsf{sdec}(\mathsf{senc}(x, y), y) = x$.

Deduction is central to methods for protocol analysis.

Deduction ($\vdash$)

$$\frac{}{\nu\widetilde{n}.\,\sigma \vdash x\sigma}\ x \in dom(\sigma) \qquad \frac{}{\nu\widetilde{n}.\,\sigma \vdash s}\ s \notin \widetilde{n}$$

$$\frac{\phi \vdash M_1 \quad \cdots \quad \phi \vdash M_k}{\phi \vdash f(M_1, \ldots, M_k)}\ f \in \Sigma \qquad \frac{\phi \vdash M \quad M =_E M'}{\phi \vdash M'}$$

where Σ is the underlying signature and $M =_E M'$ is equality in the underlying equational theory E.

Example: $\nu k.\{\mathsf{senc}(M, k)/x_1, k/x_2\} \vdash M$

Decidability of deduction

Deduction is undecidable for some theories
(even some decidable ones).

Deduction is in PTIME for theories given by a finite, convergent set of rewrite rules of the form $M \to N$ where N is a proper subterm of M or a constant.

- These conditions hold often.
  (And decidability was known in many special cases.)
- For example, $\mathsf{sdec}(\mathsf{senc}(x, y), y) \to x$.

And one can go a bit further . . .

- Other classes of theories.
- Other decision problems.

Model checking

Since the mid 1990s, many general model checking techniques and tools have been applied in this area.

They are typically limited to finite-state systems.

Proof assistants

Some automated proofs require a fair amount of expert human guidance.

Paulson's work with Isabelle is probably the best example.

The results are sophisticated theorems (and attacks), for infinite-state systems.

Other techniques

Several other approaches rely on programming-language techniques:

- control-flow analysis,
- typing,
- abstract interpretation,
- logic-program analysis.

These techniques are often incomplete but useful in examples and (relatively) easy to use.

Some of these techniques are equivalent

—but not trivially.

Blanchet's ProVerif

A checker for the applied pi calculus.

- A somewhat modified input syntax.
- An internal representation in terms of Horn clauses.
- Algorithms for certain classes of proofs.
    - Secrecy.
    - Authenticity, as correspondence assertions.
    - Most recently, certain equivalences.

ProVerif input syntax

$M, N ::=$	terms
x	variable
n	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\overline{M}\langle N \rangle.P$	output
$M(x).P$	input
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$(\nu n)P$	"new"
$let\ x = g(M_1, \ldots, M_n)\ in$	
$P\ else\ Q$	destructor application

A small process

$$(\nu k)(\overline{c}\langle \mathsf{senc}(M,k)\rangle.0 \mid c(x).let\ y = \mathsf{sdec}(x,k)\ in\ \overline{d}\langle H(y)\rangle.0)$$

Internal syntax

Each protocol is compiled into a set of rules.

- Terms (patterns):

$$M ::= x \mid n[M_1, \ldots, M_n] \mid f(M_1, \ldots, M_n)$$

- Facts:

$$F ::= attacker(M) \mid mess(M, M')$$

$attacker(M)$: the attacker may have the term M

$mess(M, M')$: M' may be sent on M

- Rules:

$$F_1 \wedge \ldots \wedge F_n \rightarrow F$$

Internal protocol representation

For example, if a process sends M on channel c when it receives N on channel d, then the rules imply

$$mess(d, N) \rightarrow mess(c, M)$$

In addition, some rules deal with communication and cryptographic functions, for example

$$attacker(x) \wedge attacker(y) \rightarrow mess(x, y)$$

$$attacker(x) \wedge attacker(y) \rightarrow attacker(\mathsf{penc}(x, y))$$

Proofs with the checker

If $attacker(M)$ is derivable from the rules, then the protocol may reveal M. Otherwise, it keeps M secret.

In addition, the checker can establish correspondence assertions, of the form "if some event happens, then some other events must have happened".

Sound approximations

The rules do not completely model protocols:

- Freshness (of keys, nonces) is only partially captured.
- The state of principals is only partially captured.
- The number of times a message appears is ignored.
  Only the fact that it appears is taken into account.

These approximations help for efficient verification

with infinite state spaces,

with any number of concurrent protocol runs.

(Cf. methods with linear-logic programming.)

A small protocol

Message 1. $A \rightarrow B$: $\mathsf{penc}((n, pK_A), pK_B)$

Message 2. $B \rightarrow A$: $\mathsf{penc}((n, k), pK_A)$

Message 3. $A \rightarrow B$: $\mathsf{senc}(s, k)$

n is a fresh nonce, k is a fresh shared key.

The code

$$P \triangleq (\nu sK_A)(\nu sK_B)$$
$$let\ pK_A = \mathsf{pk}(sK_A)\ in\ let\ pK_B = \mathsf{pk}(sK_B)\ in$$
$$\overline{e}\langle pK_A\rangle.\overline{e}\langle pK_B\rangle.(A \mid B)$$

$$A \triangleq (\nu n)(\overline{e}\langle \mathsf{penc}((n, pK_A), pK_B)\rangle \mid$$
$$e(z).let\ (x, y) = \mathsf{pdec}(z, sK_A)\ in$$
$$if\ x = n\ then\ \overline{e}\langle \mathsf{senc}(s, y)\rangle$$

$$B \triangleq e(z).let\ (x, y) = \mathsf{pdec}(z, sK_B)\ in$$
$$(\nu k)(\overline{e}\langle \mathsf{penc}((x, k), y)\rangle \mid e(z').let\ s' = \mathsf{sdec}(z', k)\ in\ 0)$$

(And there are more elaborate variants.)

Compilation into rules (a little optimized)

$$attacker(x) \wedge attacker(y) \rightarrow attacker(\mathsf{penc}(x, y))$$
$$attacker(x) \rightarrow attacker(\mathsf{pk}(x))$$
$$attacker(\mathsf{penc}(m, \mathsf{pk}(k))) \wedge attacker(k) \rightarrow attacker(m)$$
$$attacker(x) \wedge attacker(y) \rightarrow attacker(\mathsf{senc}(x, y))$$
$$attacker(\mathsf{senc}(m, k)) \wedge attacker(k) \rightarrow attacker(m)$$
$$attacker(x) \wedge attacker(y) \rightarrow mess(x, y)$$
$$mess(x, y) \wedge attacker(x) \rightarrow attacker(y)$$

$$attacker(e[])$$

$$attacker(\mathsf{pk}(sK_A[]))$$
$$attacker(\mathsf{pk}(sK_B[]))$$

$$attacker(\mathsf{penc}((n[], \mathsf{pk}(sK_A[])), \mathsf{pk}(sK_B[])))$$
$$attacker(\mathsf{penc}((n[], x), \mathsf{pk}(sK_A[]))) \rightarrow attacker(\mathsf{senc}(s[], x))$$

$$attacker(\mathsf{penc}((x, y), \mathsf{pk}(sK_B[])))$$
$$\rightarrow attacker(\mathsf{penc}((x, k[\mathsf{penc}((x, y), \mathsf{pk}(sK_B[]))]), y))$$

Can we derive $attacker(s[])$?

Solving algorithm (sketch)

1. Completion of the rules by resolution (with a particular selection strategy). For example:

$$\frac{\begin{array}{l} attacker(x) \wedge attacker(y) \rightarrow attacker(\mathsf{penc}(x, y)) \\ attacker(\mathsf{penc}(\mathsf{sign}(z, sk_A[]), \mathsf{pk}(sk_B[]))) \\ \quad \rightarrow attacker(\mathsf{senc}(s[], z)) \end{array}}{\begin{array}{l} attacker(\mathsf{sign}(z, sk_A[])) \wedge attacker(\mathsf{pk}(sk_B[])) \\ \quad \rightarrow attacker(\mathsf{senc}(s[], z)) \end{array}}$$

2. Backwards search (with loop detection).

The problem of looping

A natural tool for determining derivability of a fact from clauses is resolution.

Basic SLD-resolution (as in Prolog) would never terminate:

$$attacker(\mathsf{senc}(m, k)) \wedge attacker(k) \rightarrow attacker(m)$$

A key idea is to avoid resolving on $attacker(x)$.

Optimizations

Various optimization help prune the search space:

– We can eliminate subsumed clauses.

– We can eliminate duplicate hypotheses in clauses.

– We can eliminate hypotheses $attacker(x)$ when x does not appear elsewhere in the clause.

– We can assume, then verify secrecy assumptions.

Termination

The resolution algorithm does not always terminate,
but it does terminate for tagged protocols:

A protocol is tagged when each encryption, signature, ...
in the protocol is distinguished from others by a constant
tag c_i

$$\{c_i, M_1, ..., M_n\}_K$$

- This is a large class of protocols.
- It is easy to add tags.
- Tagging is generally a good design practice.

Examples

Pentium III, 1 GHz.

Protocol	Result	ms
Needham-Schroeder public key	Attack [Lowe]	10
Needham-Schroeder public key corrected	Secure	7
Needham-Schroeder shared key	Attack [Denning]	52
Needham-Schroeder shared key corrected	Secure	109
Denning-Sacco	Attack [AN]	6
Denning-Sacco corrected	Secure	7
Otway-Rees	Secure	10
Otway-Rees, variant of Paulson98	Attack [Paulson]	12
Yahalom	Secure	10
Simpler Yahalom	Secure	11
Main mode of Skeme	Secure	23

Some larger examples

- A protocol for certified email.

- The JFK ("Just Fast Keying") protocol for IP security.

- Password-based protocols
  (such as the Encrypted Key Exchange).

- Web-services protocols.

- Electronic-voting protocols.

Some of the automatic proofs take minutes.

Some manual proofs are combined with automatic proofs.

Code (fragment)

```
let processS =
  (* The attacker chooses possible recipients of the message *)
  in(c, recipient);
  (* Build the message to send *)
  new msgid;
  let m = Message(recipient,msgid) in
  (* Step 1 *)
  new k;
  let em = senc(k,m) in
  let hs = H((cleartext, em)) in
  let S2TTP = penc(TTPEncKey, (Sname, (Give, k, recipient, hs))) in

  out(recipient, (TTPname, em, cleartext, S2TTP));

  (* Step 4 *)
  !
  in(Sname, inmess4);
  let (=Released, =S2TTP, =recipient) = checkS(TTPVerKey, inmess4) in
  (* S knows that R has read the message *)

  0
  else out(Sname, inmess4).
```