A Protocol Compiler for Secure Sessions in ML

Ricardo Corin, Pierre-Malo Deniélou

INRIA—Microsoft Research Joint Centre

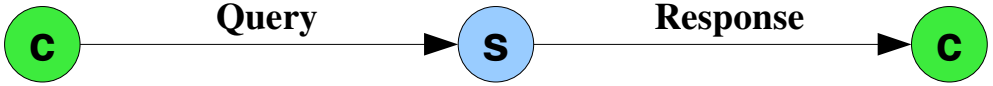
http://www.msr-inria.inria.fr/projects/sec/sessions/

Programming distributed applications

- How to program networked independent sites?
  - Little control over the runtime environment

    ➔ Can we trust the network?
  - Sites have their own code & security concerns

    ➔ Can we trust them?
- Communication abstractions simplify this task
  - Basic communication patterns, e.g. RPCs

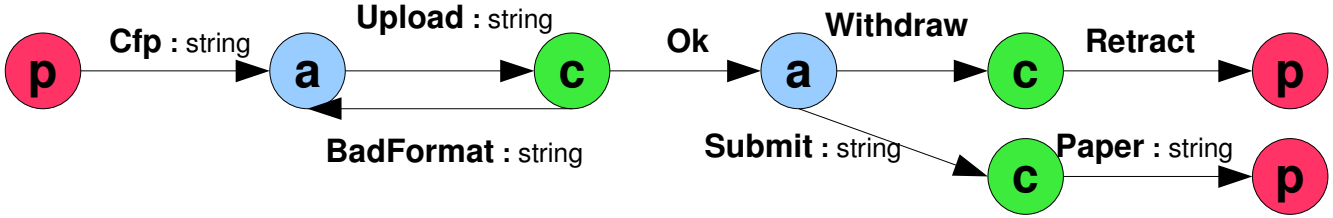


  - They hide implementation details

    (message format, routing, security,...)

Sessions

- Specification of a message flow between roles
  - Graph with roles as nodes and labelled messages as edges
  - Example: session with 3 parties, a loop and branches.



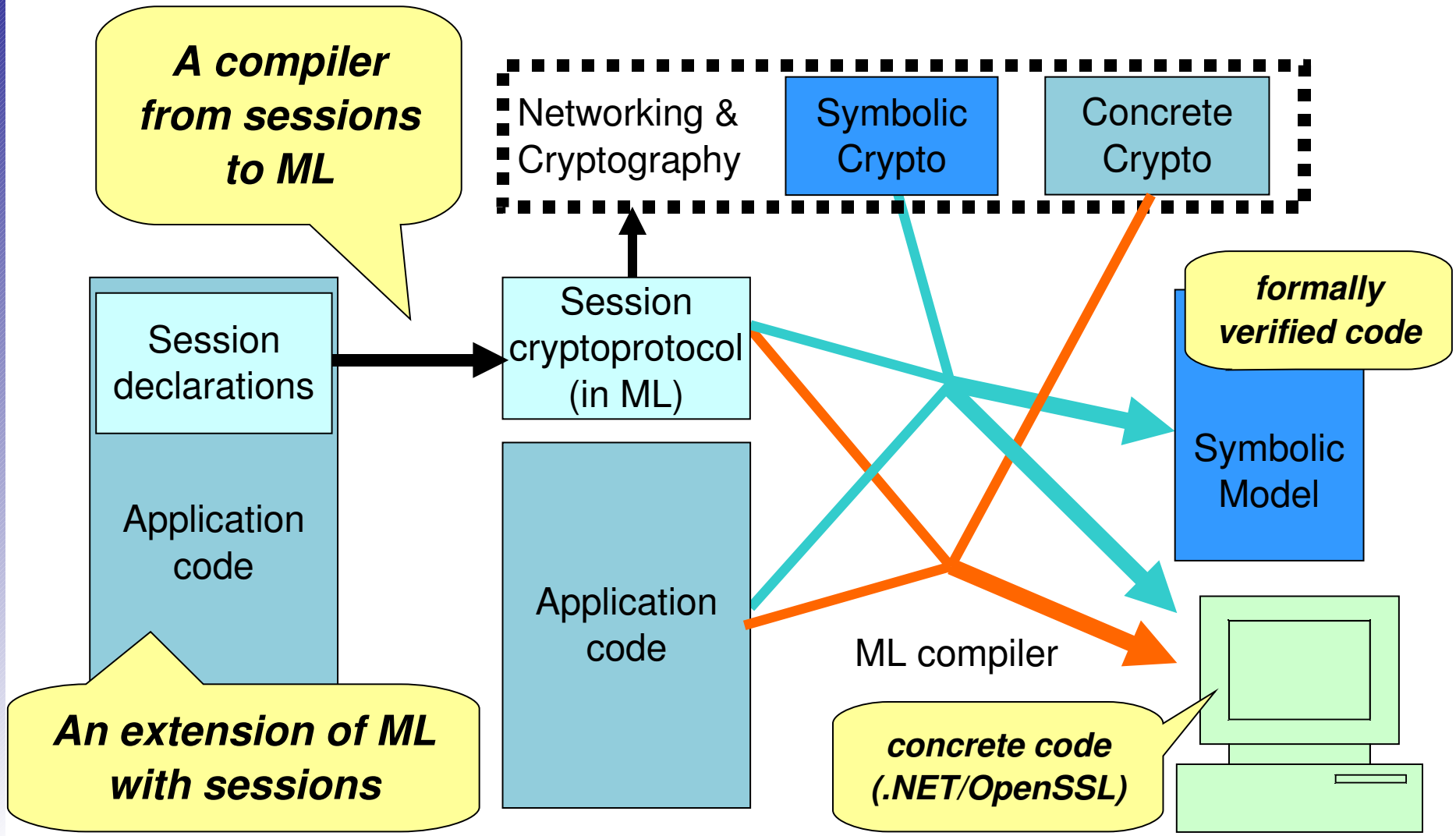
  - Active area for distributed programming
    - A.k.a. protocols, or contracts, or workflows
    - Pi calculus settings, web services, operating systems
    - Common strategy: type systems enforce protocol compliance
      "If every site program is well-typed, sessions follow their spec"

Compiling session to cryptographic protocols

- We extend ML with session declarations that express message flows
- Then we compile session declarations to protocols that shield our programs from any coalitions of remote peers
- We obtain that:

1. Well-typed programs always play their roles
   → functional result (uses ordinary ML-typechecking)
2. If a program uses sessions implemented with our compiler, then remote sites can be assumed to play their roles, without trusting their code
   → security theorem

Architecture



Outline

I. Programming with Sessions

1. Language description
2. Session usage and interface generation

II. Compiler internals

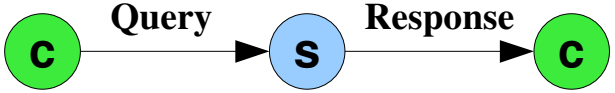
1. Security protocol
2. Module generation

A small session language

$\tau ::=$
  unit | int | string
$p ::=$
  $!(f_i : \tau_i \; ; \; p_i)_{i<k}$
  $?(f_i : \tau_i \; ; \; p_i)_{i<k}$
  $\mu\chi.p$
  χ
  0
$\Sigma ::=$
  $(r_i : T_i = p_i)_{i<n}$

Payload types
  base types
Role processes
  send
  receive
  recursion declaration
  recursion
  end
Sessions
  initial role processes

A very simple RPC session:

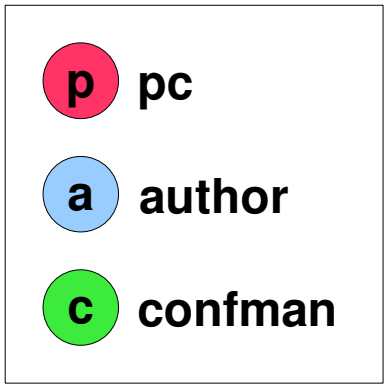


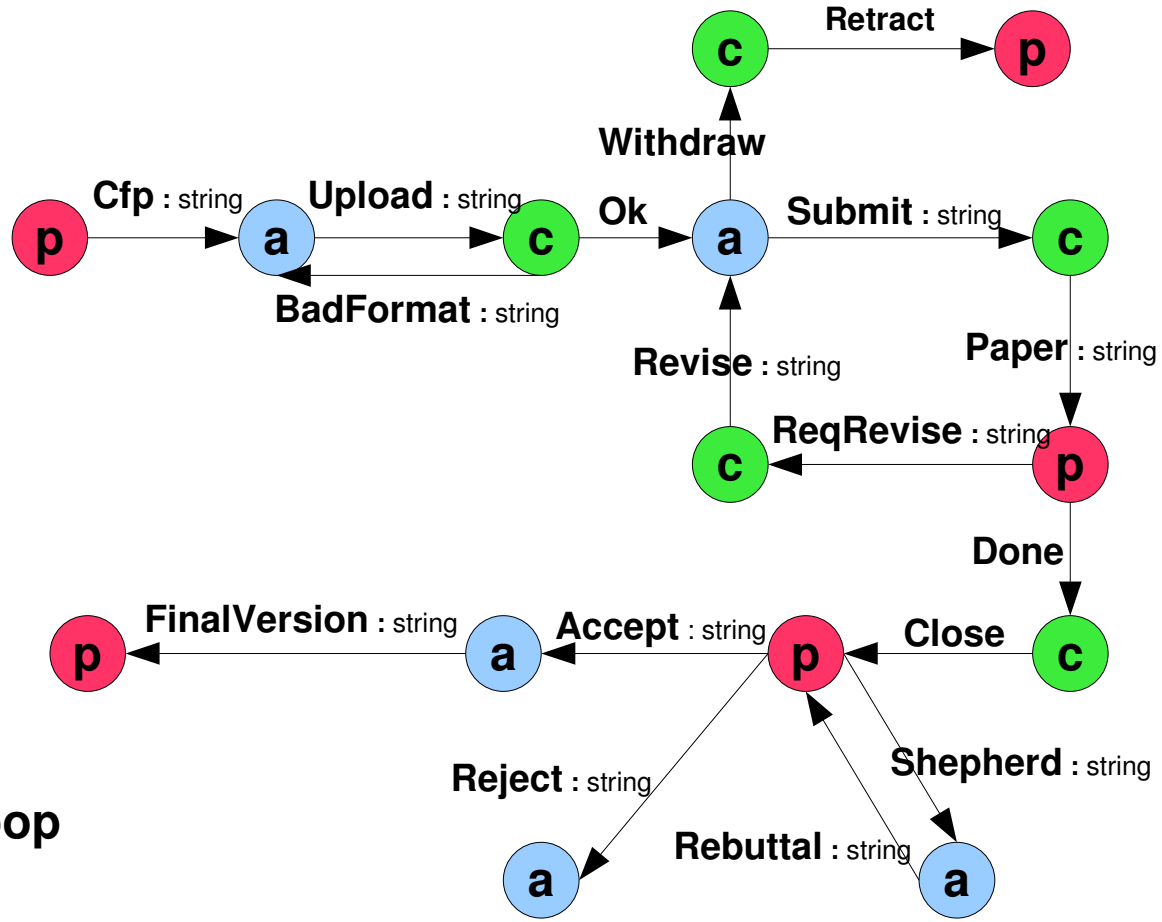
Session RPC =

role client:int = !**Query:string** ; ?**Response:int**

role server:unit = ?**Query:string** ; !**Response:int**

A Conference Management Session

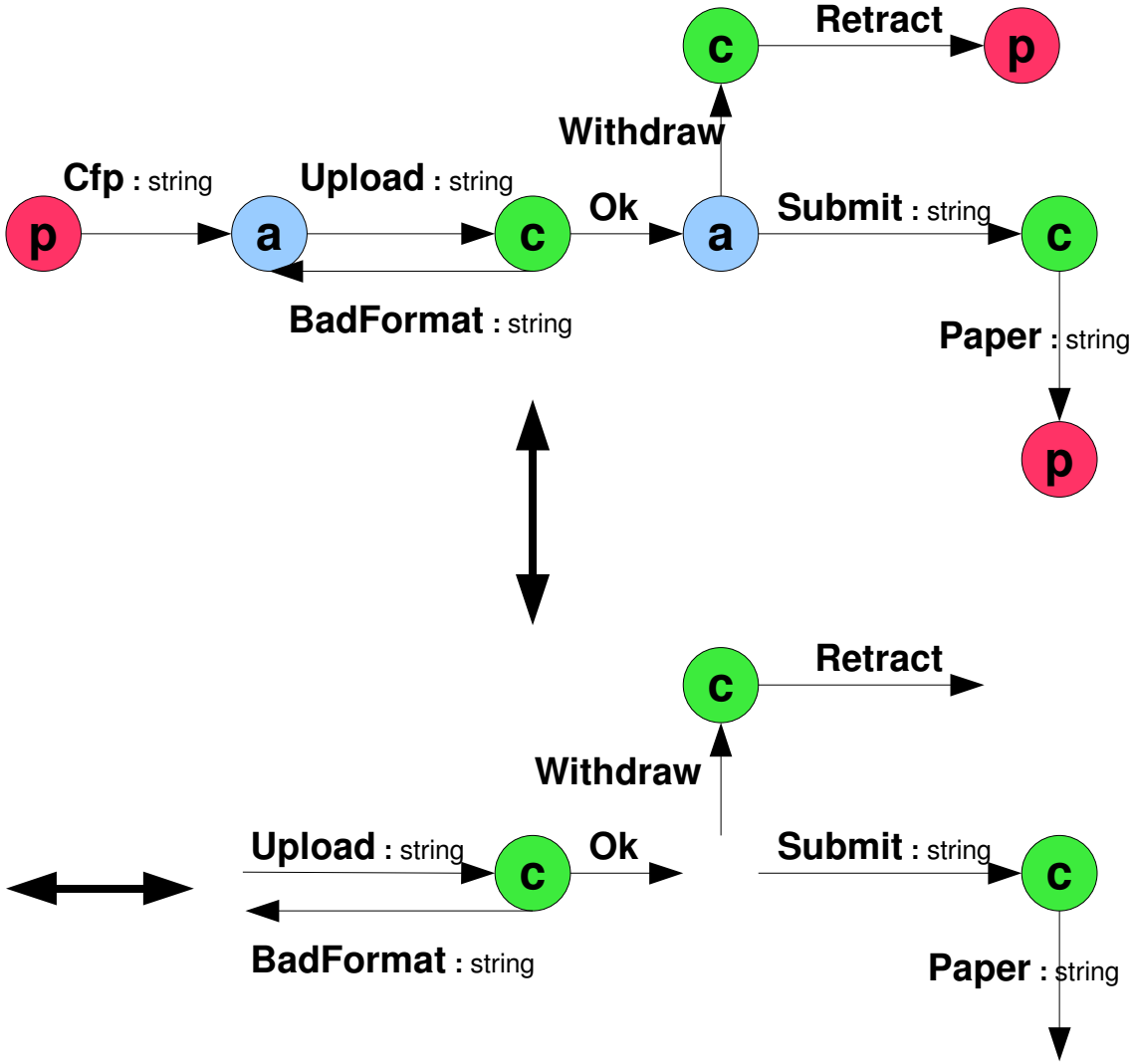




1. Call for paper
2. Upload sequence
3. Revision loop
4. Decision & Rebuttal Loop

Global and Local sessions

```
Session CMS =
  role pc:string =
    ! Cfp:string;
    mu start.
    ?(Paper:string
    + Retract)

  role author =
    ?Cfp:string;
    mu start.
    !Upload:string;
    ?(BadFormat:string;start
    + Ok;!(Submit:string
          + Withdraw))

  role confman =
    mu start.
    ?Upload:string;
    !(BadFormat:string;start
    + Ok;?(Submit:string;!Paper:string
          + Withdraw;!Retract))
```

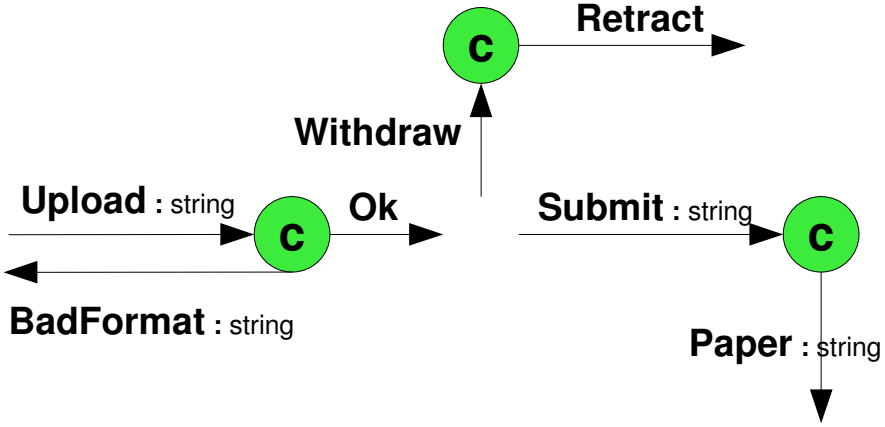


Source file `cms.session`

Generated Interface

```
Session CMS =
  role pc:string = (...)

  role author = (...)

  role confman =
    mu start.
    ?Upload:string;
    !(BadFormat:string;start
    + Ok;?(Submit:string;!Paper:string
          + Withdraw;!Retract))
```

Source file `cms.session`




```
type msg11 = {
  hUpload : (principals -> string -> msg12)}
and msg12 =
  | BadFormat of string * msg11
  | Ok of unit * msg13
and msg13 = {
  hSubmit : (principals -> string -> msg14);
  hWithdraw : (principals -> unit -> msg15)}
and msg14 = Paper of string * unit
and msg15 = Retract of string * unit

type confman = principal -> msg11 -> unit
```

Generated file `CMS.mli`

Each role is compiled to a role function "confman" that expects continuations to drive the session (CPS style).

The continuations are constrained by the generated types.

Role Programming

- Principal registration
  - Give crypto and network information (public/private keys, IP, ...)
- CPS programming

```
type msg11 = {
  hUpload : (principals -> string -> msg12)}
and msg12 =
  | BadFormat of string * msg11
  | Ok of unit * msg13
and msg13 = {
  hSubmit : (principals -> string -> msg14);
  hWithdraw : (principals -> unit -> msg15)}
and msg14 = Paper of string * unit
and msg15 = Retract of string * unit

type confman = principal -> msg11 -> unit
```

Generated file `CMS.mli`

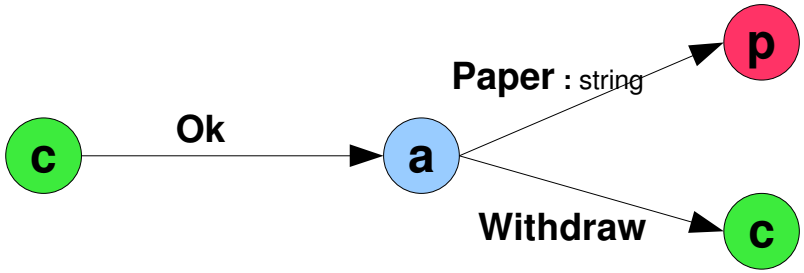
```
open CMS

let handler_submission =
 { hSubmit = fun _ s -> Paper(s, ()) ;
   hWithdraw = fun _ () -> Retract((), ()) }

let rec handler_paper prins draft =
  if String.length draft > 12
  then BadFormat("Make it shorter!",
                 {hUpload = handler_paper})
  else Ok((), handler_submission)

let result =
  confman "bob" {hUpload = handler_paper}
```

User code `foo.ml`

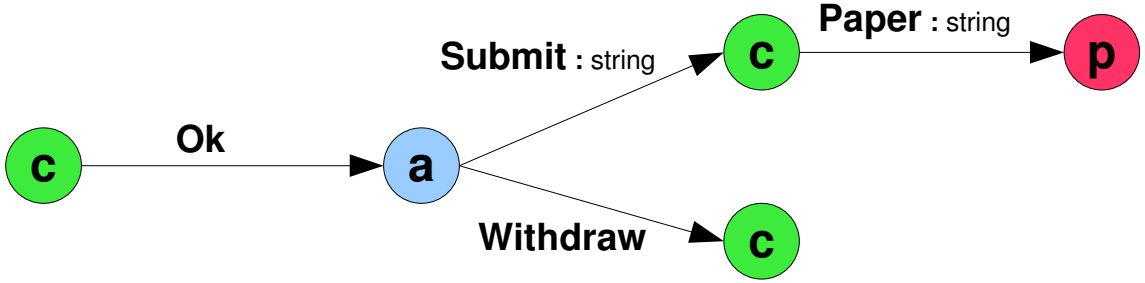
Ordinary ML type-checking provides functional guarantees!

Implementability conditions

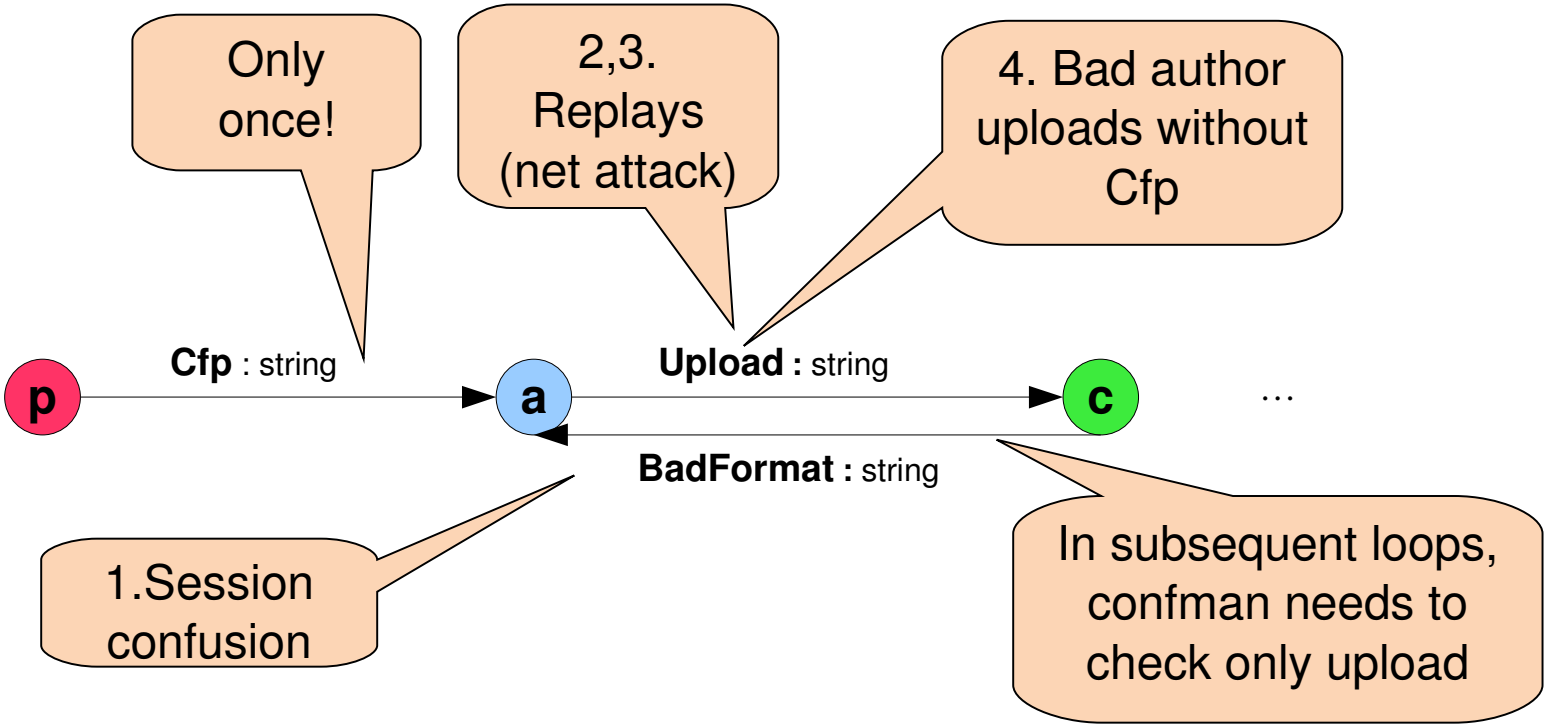
- We want session integrity.
- Some sessions are always vulnerable:



- We detect them and rule them out
  - They can also be turned into safe sessions with extra messages:

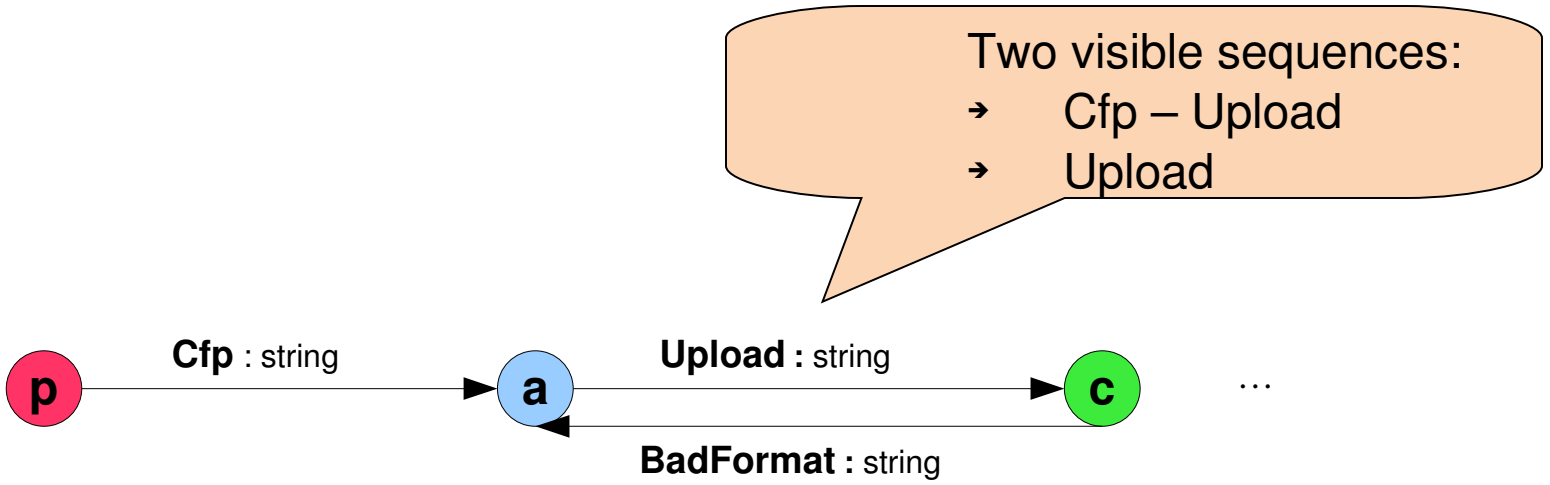


Protocol outline & (Potential) attacks



1. Use unique session id = hash(session decl + nonce N + principals)
2. Use cache for initial session messages
3. Use logical clock for loop session messages
4. Sign labels and session ids
   - ➔ What evidence do we forward?

Efficient Forwarding



Visibility =

minimum information needed to update state of local role

- Can be computed statically from the session graph
- Any less information would break integrity
- More work to the compiler = less runtime tests
- This actually simplifies formal proofs!

Session Integrity, Formalized

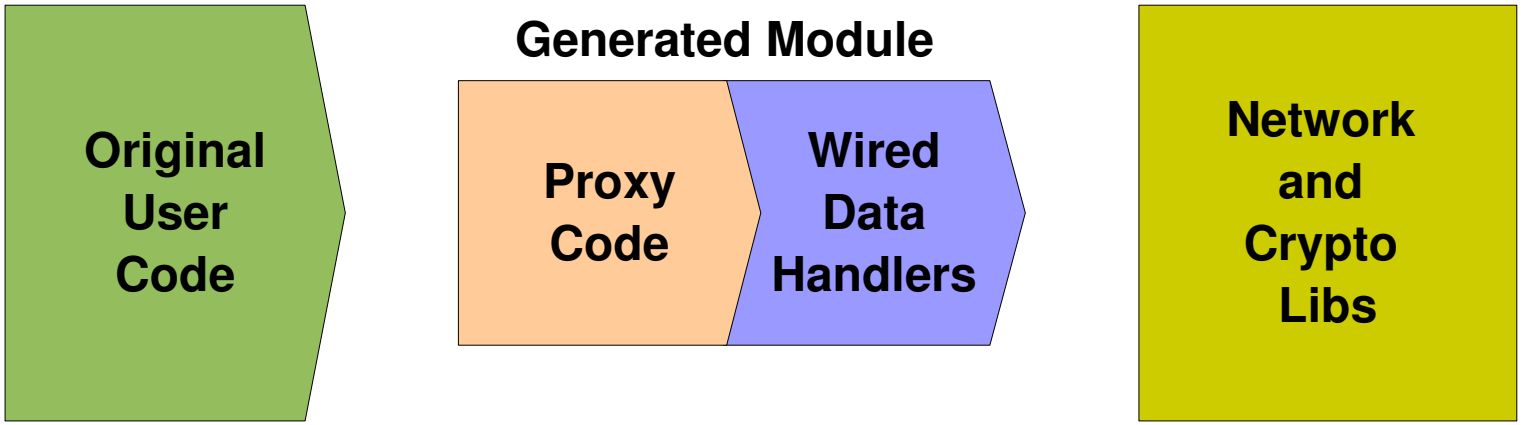
- For any run of any choice of honest principals running roles of compiled session declarations plus any coalition of dishonest principals + network attacker

  ➔ there exist valid paths in the session declarations that are consistent with all the messages sent and received by the honest principals

- Formalized as two semantics (previous work):
  - one “ideal” with hardwired sessions,
  - one “real” using our compiler and symbolic libraries
- We show a may-testing simulation from the real to the ideal

Compilation outline

- Generation of the global graph
  - Well-formed and Implementability conditions
  - Visible sequence generation
- For each role, generation of the local side of the crypto protocol



Wired Data handling

- Receive functions (receiveWired*node*) : Message analysis
  - Receive the message on the network, decompose, check session id
  - Match label against possible incoming messages
  - Check signatures (using visibility) and logical time-stamps
  - Update local store and logical clock
  - Check against the cache
- Send functions (sendWired*label*): Message generation
  - Session id, msg headers (session id+sender id+receiver id)
  - Marshall payload
  - Build signature, update the local store and logical clock
  - Send the full message on the network

Proxy code

Links the user code with sendWired/receiveWired functions

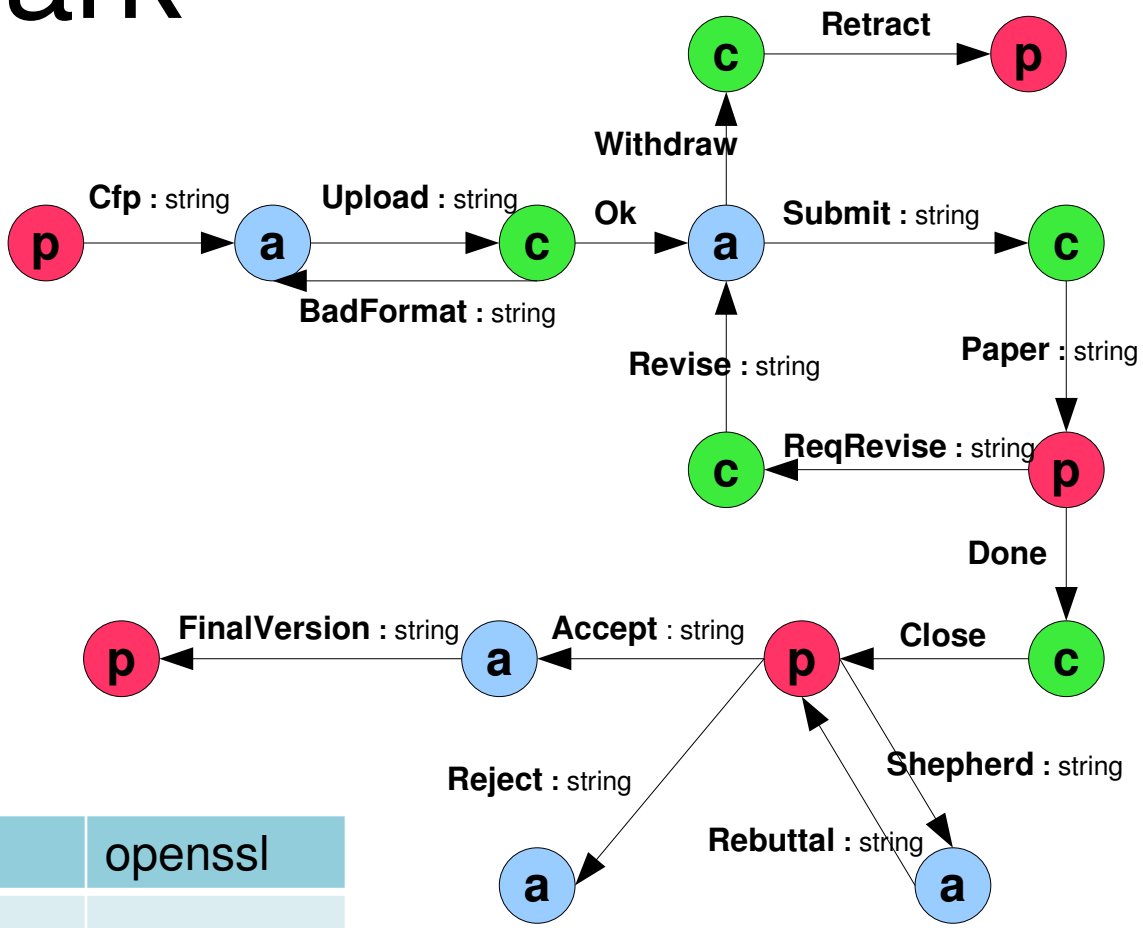
```
type msg11 = {
  hUpload : (principals -> string -> msg12)}
and msg12 =
  | BadFormat of string * msg11
  | Ok of unit * msg13
and msg13 = {
  hSubmit : (principals -> string -> msg14);
  hWithdraw : (principals -> unit -> msg15)}
and msg14 = Paper of string * unit
and msg15 = Retract of string * unit

type confman = principal -> msg11 -> unit
```

Generated file `CMS.mli`

```
(...) (* header sending *)
and confman_msg12  (st:state) : msg12 -> unit =
 function
  | Ok(x,next) ->
    let newSt = sendWiredOk host 1 (WiredOk(st, x)) in
    confman_msg13 newSt next
  | BadFormat(x,next) ->
    let newSt =
      sendWiredBadFormat host 1 (WiredBadFormat(st, x)) in
    confman_msg11 newSt next
(* header receiving *)
and confman_msg11  (st:state) : msg11 -> unit =
 function handlers ->
  let r = receiveWired11 1 host st () in
    match r with
    | WiredUpload (newSt, x) ->
    let next = handlers.hUpload newSt.prins  x in
    confman_msg12 newSt next
(...)
```

Generated file `CMS.ml`

Benchmark



500 iterations in each loop
(4000 messages in total)

	No crypto	crypto	openssl
1st loop	0.23s	2.95s	
2nd loop	0.46s	6.11s	
3rd loop	0.24s	2.98s	
total	0.94s	12.04s	8.38s

Conclusion & Future Work

Cryptographic protocols can sometimes be derived (and verified) from application security requirements

- Strong, simple security model
- Safer, more efficient than ad hoc design & code

Improvements to session expressiveness

- Enable access control over payloads
  - Roles can deliver data to other roles securely
- Enable dynamic principal selection
  - As opposed to the initiator picking everyone

Improve performance (symmetric cryptography?)

Thanks to
Karthikeyan Bhargavan, Cédric Fournet, James J. Leifer,
Jean-Jacques Lévy

Try our session compiler!

http://www.msr-inria.inria.fr/projects/sec/sessions/

Existing approaches

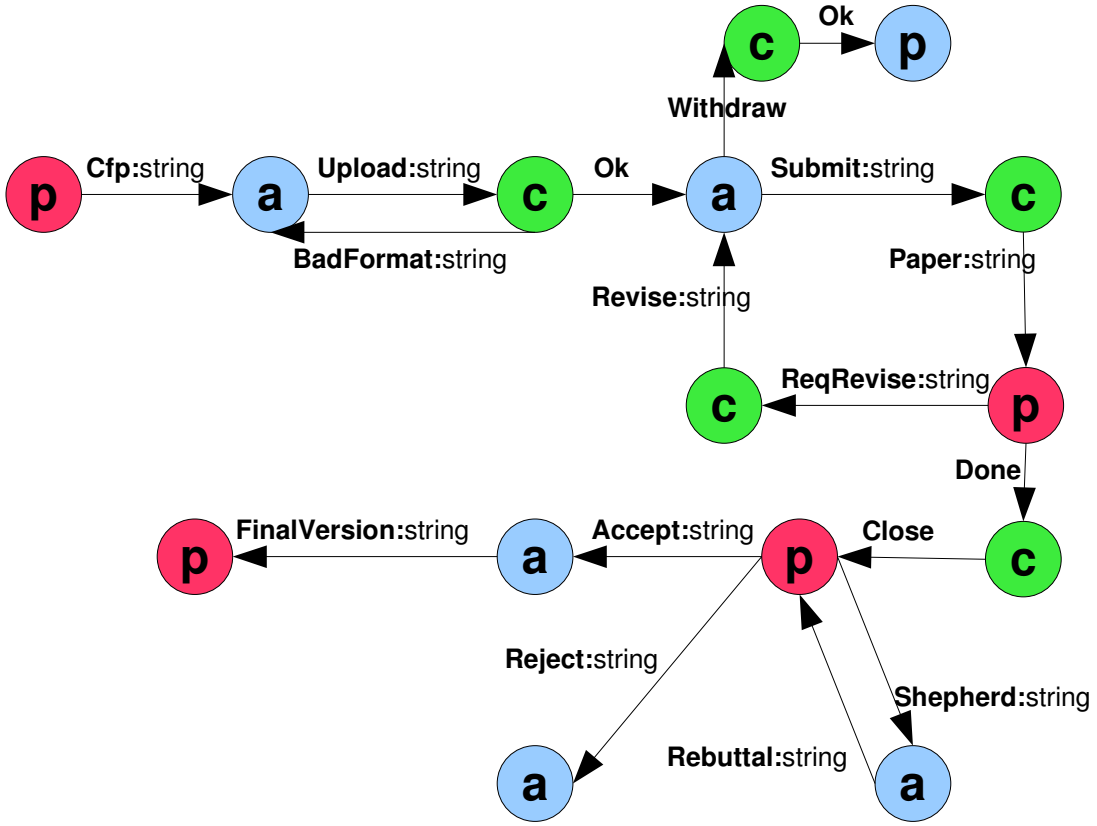
- Session types:
  - First ‘session types’: Pi-calculus based [Honda&Vasconcelos 98, Gay & Hole 99]
    - Describe message flows on single channels
  - ‘Behavioral types’ [Kobayashi & Igarashi 01]
    - Multi channel flows (types are CCS processes)
  - ‘Contracts’ in Singularity OS [Fahndrich et al. 06]
  - ‘Workflows’ in Web services
    - WSDL, WS-SecureConversation [Bhargavan et al. 05]
    - Global interactions [Carbone et al. 07]
- Protocol Analysis, Synthesis, and Transformation :
  - Lots of work, but on abstract, isolated protocols
  - Next challenge: integrate with expressive, real-life (distributed) languages
    - Secure Channels Implementation [Abadi, Fournet, Gonthier 02]

Discussion

- Session types are an active area of study
  - we address their secure implementation
- Protocol verification:
  - We verify a reference implementation—not a simplified model
  - Our results hold for any number of (concurrent) sessions
  - Even for a single session, this is beyond automated verification tools (loops and branching)
  - Crypto is Dolev-Yao, not far from computational model
  - Integrity, not liveness (so no progress or global termination)
- Related work on secure implementations of process calculi, on automated protocol transformations

Differences with CSF'07 paper

- Typed interface
- Compiler released
- Internals of the compiler
- More serious case study

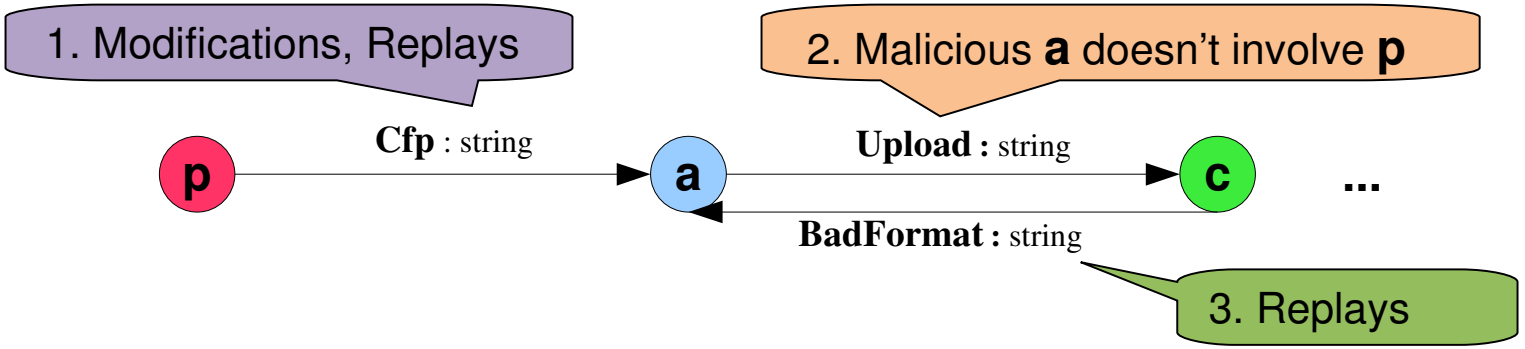

c
Ok
p
Withdraw
p
Cfp:string
a
Upload:string
c
Ok
a
Submit:string
c
BadFormat:string
Paper:string
Revise:string
c
ReqRevise:string
p
Done
p
FinalVersion:string
a
Accept:string
p
Close
c
Reject:string
Shepherd:string
a
Rebuttal:string
a

Sessions and Security

- Secure implementation problem:
  1. *"If every site program is well-typed, sessions follow their spec"*

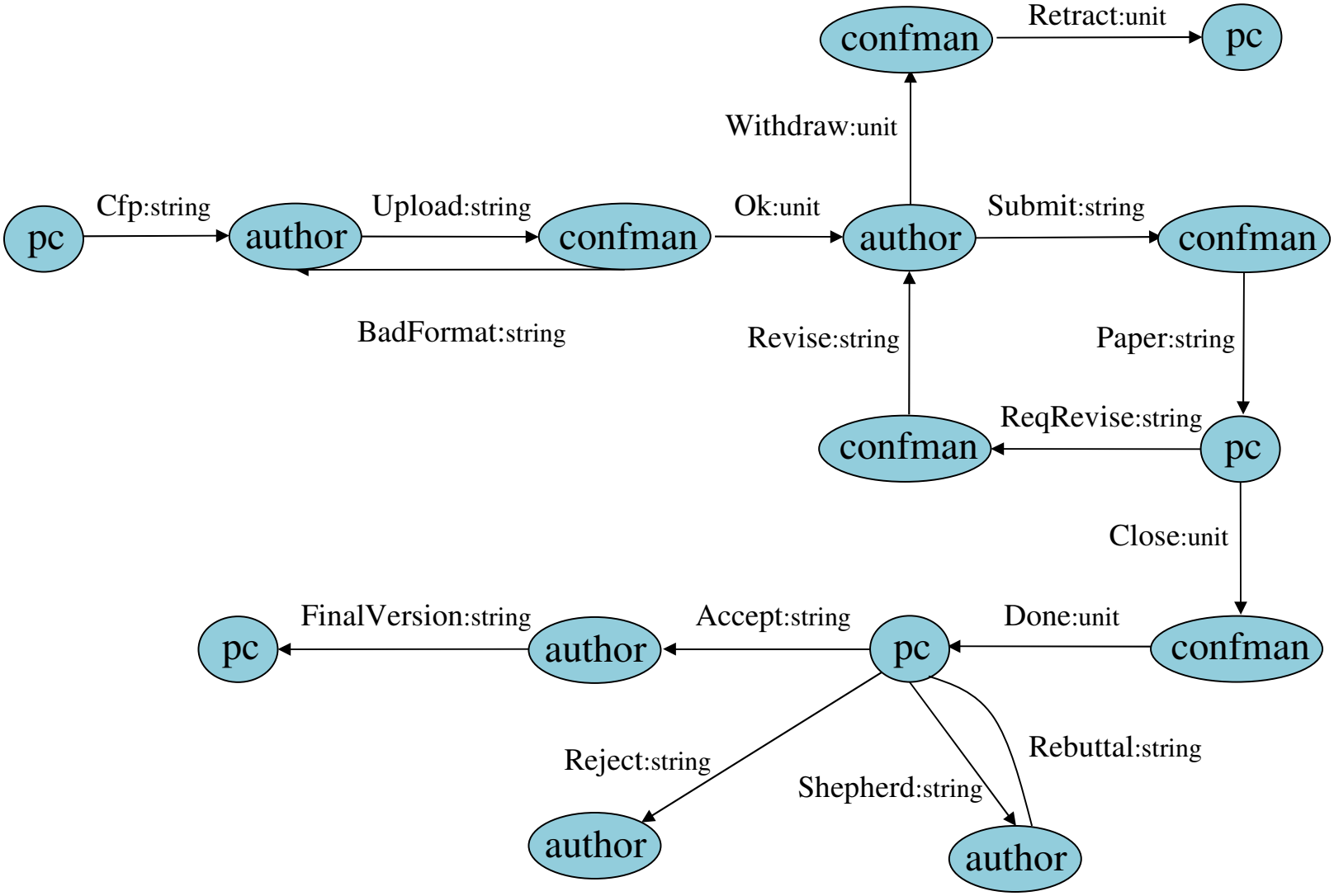
     ➔ only if we can trust the network
  2. However *"Sites wish to interact, but they have their own code & security concerns"*

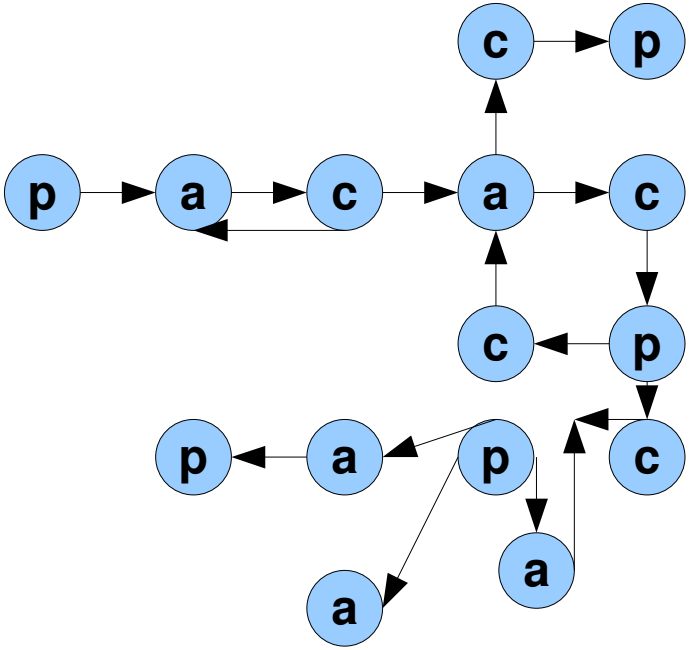
     ➔ they do not necessarily trust one another

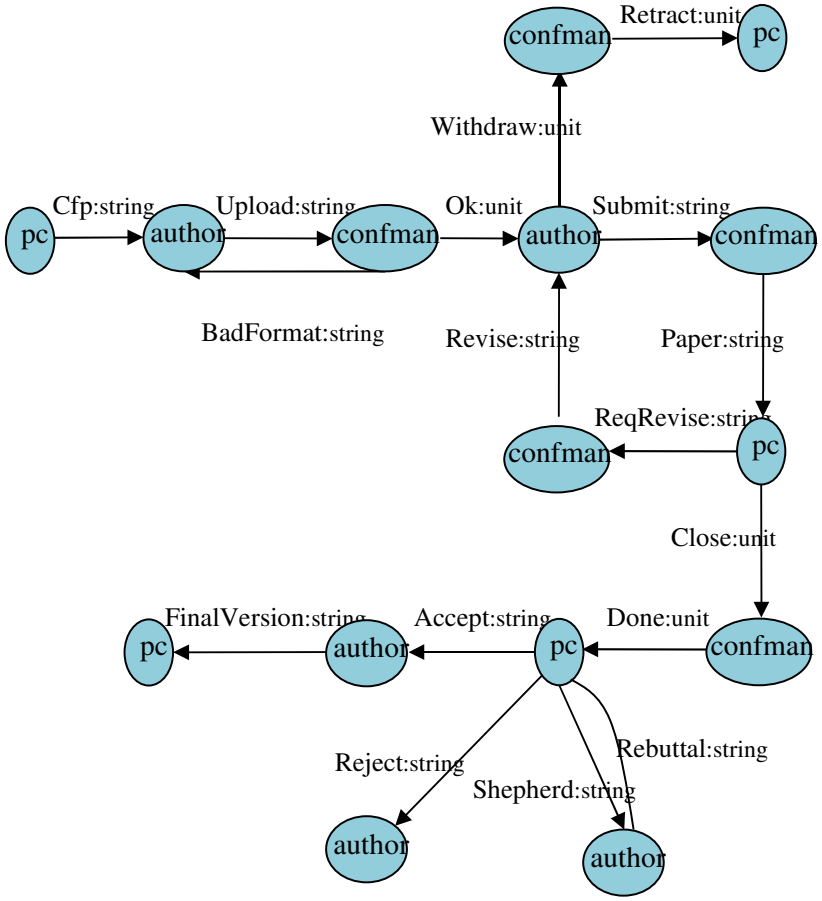


- We need a specialized secure implementation
  - Prevents replay attacks (using caches, counters, nonces)
  - Provides authentication using cryptographic signatures

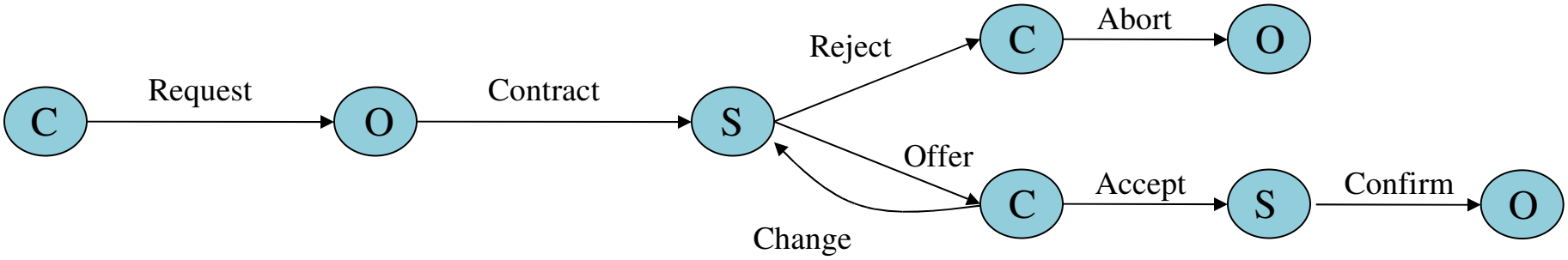
A Conference Management Session

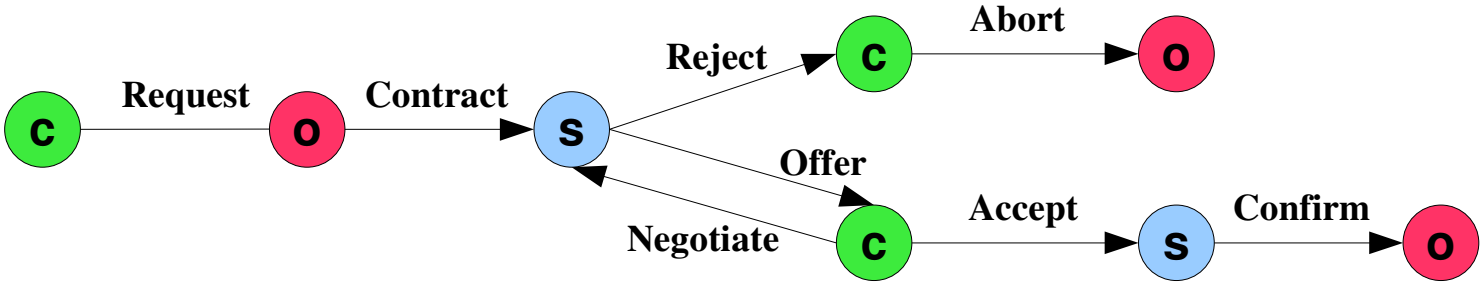



c
p
p
a
c
a
c
c
p
p
a
p
c
a
a


confman
Retract:unit
pc
Withdraw:unit
Cfp:string
Upload:string
Ok:unit
Submit:string
pc
author
confman
author
confman
BadFormat:string
Revise:string
Paper:string
ReqRevise:string
confman
pc
Close:unit
FinalVersion:string
Accept:string
Done:unit
pc
author
pc
confman
Reject:string
Shepherd:string
Rebuttal:string
author
author

Demo


C
Request
O
Contract
S
Reject
C
Abort
O
Offer
C
Accept
S
Confirm
O
Change


c
Request
o
Contract
s
Reject
c
Abort
o
Offer
c
Accept
s
Confirm
o
Negotiate

Expressing sessions

- Terminology:
  - *Roles* : behaviour of the session participants
  - *Principals* : instantiate roles at runtime
  - *Messages* : consists of labels and payloads

- Two ways to represent sessions:
  - As a graph → useful to globally reason on sessions
  - As a collection of local roles
    → useful for the language semantics and implementation
  - Representations are interconvertible

Security Goal: Global session integrity

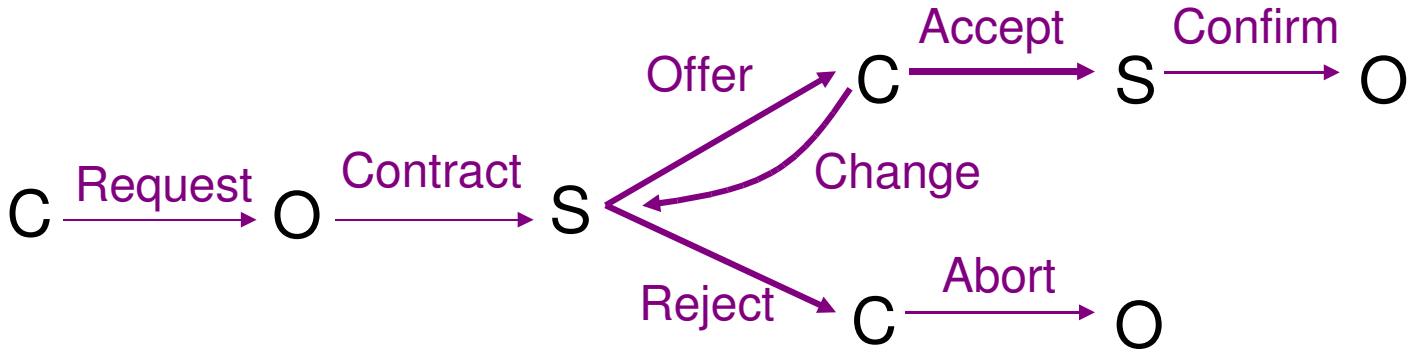
- For any run of any choice of honest principals running roles of compiled session declarations plus any coalition of dishonest principals + network attacker

  ➔ there exist valid paths in the session declarations that are consistent with all the messages sent and received by the honest principals

  - This generalizes correspondence assertions

- Our compiler generates code that enforces this.

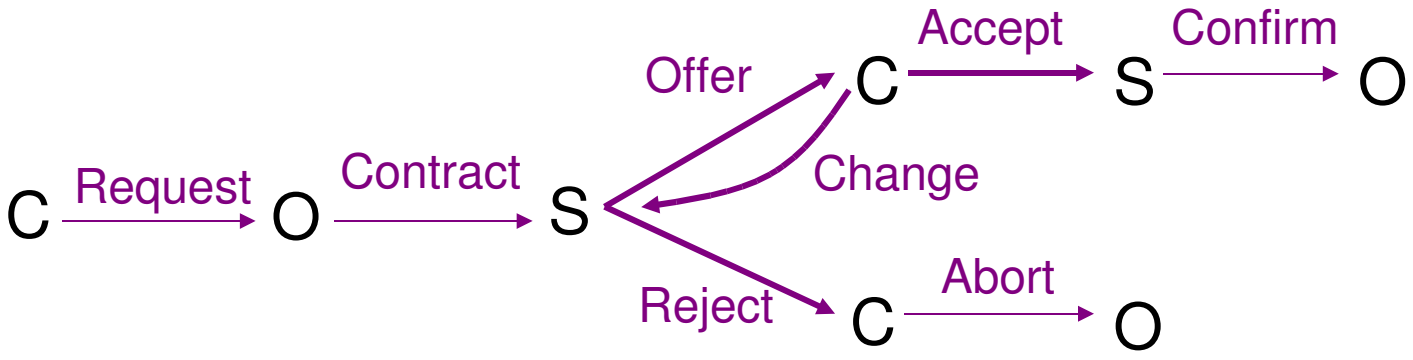
Example



"Customer C negotiates the delivery of an item with the store S; the transaction is registered by an officer O."

```
session S3 =
  role store:string =
    ?Contract:string; mu start.
    !( Offer:string;
        ?( Change:string; start
        + Accept; !Confirm )
      + Reject )

  role officer = (...)
  role customer = (...)
```

Demo



"Customer C negotiates the delivery of an item with the store S; the transaction is registered by an officer O."

F+S programming language

F

+

S

$T ::=$	Type expressions
$\quad t$	$\quad$ type variable
$\quad \mathsf{int}, \mathsf{string}, \mathsf{unit}$	$\quad$ base types
$\quad T$ chan	$\quad$ channel types
$\quad T_1 \to T_2$	$\quad$ arrow type
$v ::=$	Values (also used as Patterns)
$\quad x$	$\quad$ variable
$\quad 0, 1, \ldots,$ Alice, Bob, $\ldots, ()$	$\quad$ constants for base types
$\quad l, c, n, \ldots$	$\quad$ names for functions, channels, nonces
$\quad f(v_1, \ldots, v_k)$	$\quad$ constructed term (when f has arity k)
$e ::=$	Expressions
$\quad v$	$\quad$ value
$\quad l\ v_1 \ldots v_k$	$\quad$ function application
$\quad \mathsf{match}\ v\ \mathsf{with}\ (\lvert v_i \to e_i)_{i<k}$	$\quad$ value matching
$\quad \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	$\quad$ value definition
$\quad \mathsf{let}\ (l_i\ x_0 \ldots x_{k_i} = e_i)_{i<k}\ \mathsf{in}\ e$	$\quad$ mutually-recursive function definition
$\quad \mathsf{type}\ (t_i = (\lvert f_{j_i}\ \mathsf{of}\ \widetilde{T}_{j_i})_{j_i<k_i})_{i<k}\ \mathsf{in}\ e$	$\quad$ mutually-recursive datatype definition
$\quad \mathsf{session}\ S = \Sigma\ \mathsf{in}\ e$	$\quad$ session type definition
$\quad S.r^b\ \widetilde{v}\ (v)$	$\quad$ session entry
$\quad s.p(e)$	$\quad$ session role (run-time only)
$E[\cdot] ::=$	Evaluation contexts
$\quad [\cdot]$	$\quad$ top level
$\quad \mathsf{let}\ x = E[\cdot]\ \mathsf{in}\ e_2$	$\quad$ sequential evaluation
$\quad s.p(E[\cdot])$	$\quad$ in-session evaluation (run-time only)
$P ::=$	Processes
$\quad e$	$\quad$ running thread
$\quad P \mid P$	$\quad$ parallel composition
$\quad \mathbf{0}$	$\quad$ inert process

F+S semantics

- Role semantics $\rightarrow_r$:

$$(\textsc{Send}) \quad !(f_i : \widetilde{\tau}_i \; ; \; p_i)_{i<k} \xrightarrow{\overline{f}_i}_r p_i \qquad (\textsc{Receive}) \; ?(f_i : \widetilde{\tau}_i \; ; \; p_i)_{i<k} \xrightarrow{f_i}_r p_i$$

- F+S semantics is a "centralized session monitor"
  - layered semantics using $\rightarrow_r$, $\rightarrow_s$ and $\rightarrow_e$

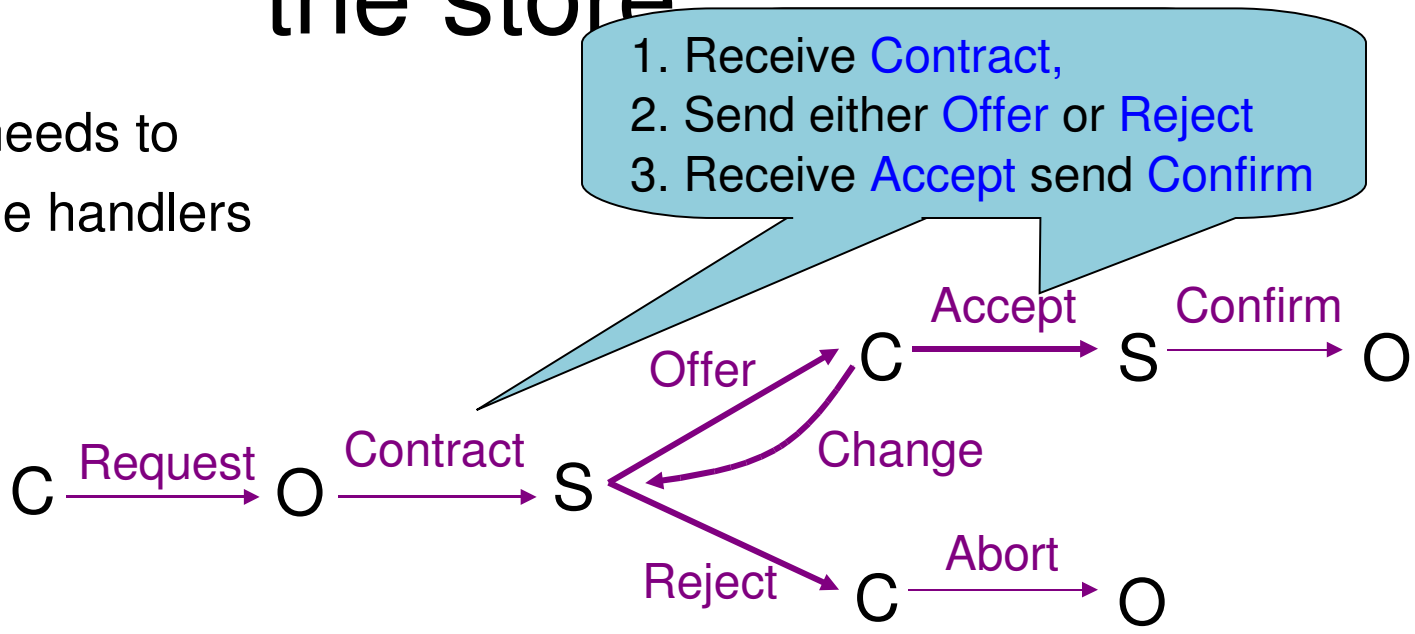
$$(\textsc{Step}) \quad \frac{p \xrightarrow{\eta}_r p'}{\rho, s.p \xrightarrow{\eta}_s \rho, s.p'}$$

$$(\textsc{SendS}) \quad \frac{\rho, s.p \xrightarrow{\overline{g}}_s \rho', s.p'}{\rho, s.p \; (g(\widetilde{v}), w) \xrightarrow{s\overline{g}\,\widetilde{v}}_e \rho', s.p' \; (w)}$$

  - constitutes our global specification for sessions
  - does not exist in F, our implementation language

Using a compiled session: the store

- User code needs to provide message handlers (CPS style)

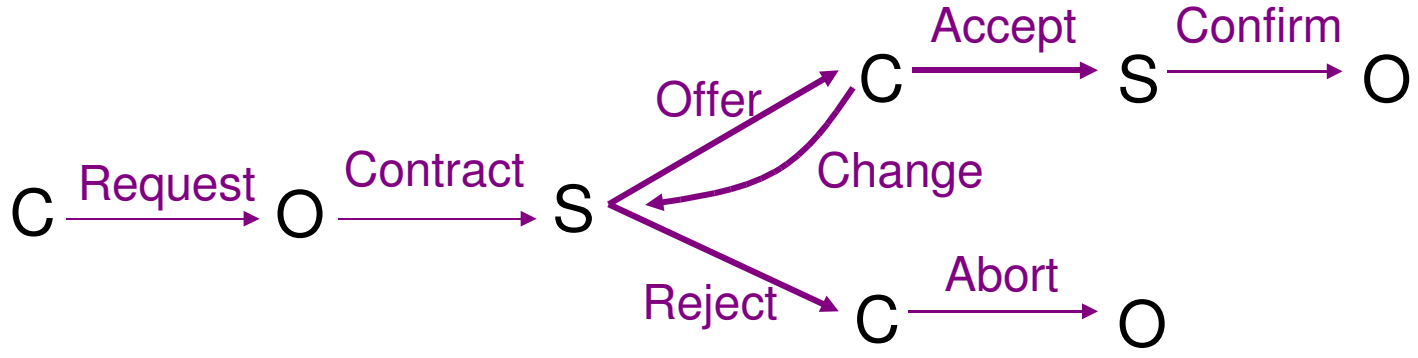



```
type msg11 = {
    hContract : (principals → string → msg12)}
and msg12 =
    Offer of (string ∗ msg13)
  | Reject of (unit ∗ string)
and msg13 = {
    hChange : (principals → string → msg12) ;
    hAccept : (principals → unit → msg14)}
and msg14 =
    Confirm of (unit ∗ string)
```

```
val store : principal → msg11 → string
```

Ordinary ML type-checking provides functional guarantees!

Coding the store



- A store that offers deliveries in Redmond (default), Cambridge, or Orsay:

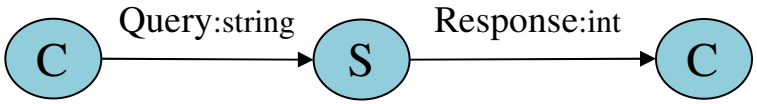
```
let offer loc = List.assoc loc
  [ "Default", "Redmond, 8am-9am";
    "Redmond", "Redmond, 3pm-4pm";
    "Orsay", "Orsay, lunchtime";
    "Cambridge", "Cambridge, 6pm-7pm" ]

let server prins req =
  printf "Server: session starting for %s.\n\n" req;
  let rec new_offer prins (loc:string) =
  try
    let o = offer loc in
    Offer(o, {
      hChange = new_offer;
      hAccept = (fun _ () → Confirm((),"in "^o)); })
  with _ → Reject((),"no offer available") in
  new_offer prins "Default"

  let status = S3.store "bob" { hContract = server; } in
  printf "Store: Done! %s.\n\n" status)
```

```
val store : principal → msg11 → string
```

Demo…

RPC example

- Global description:



- Equivalent to a local description:

```
session Rpc =
        role C:int = !Query:string; ?Response:int
        role S:unit = ?Query:string; !Response:int
```

- Our compiler generates functions “C” and “S” in module “Rpc”
- Programmers drive their roles using CPS-style continuations:
  - “C” expects a Query string + a continuation for handling the response
  - “S” expects a handler for the Query that generates the

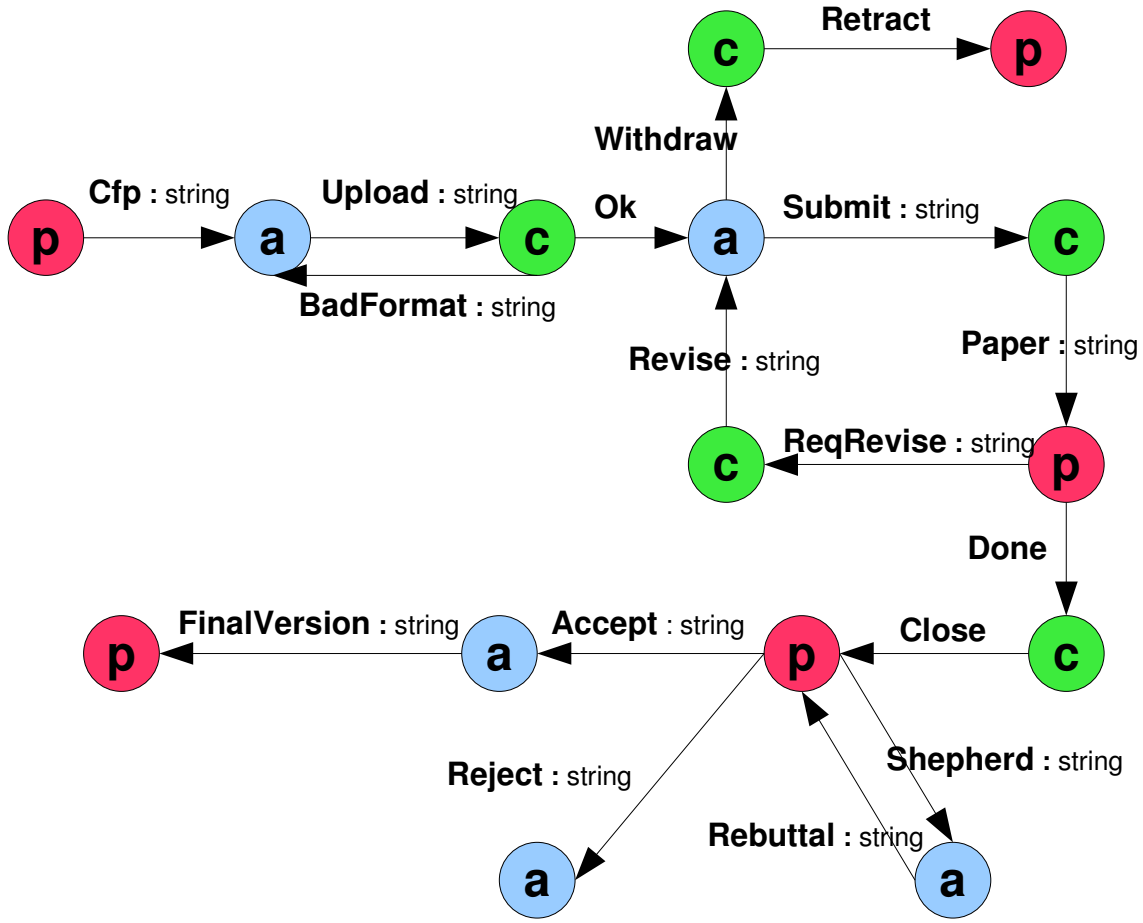
Session Integrity Goal

- For any run of any choice of honest principals running roles of compiled session declarations plus any coalition of dishonest principals + network attacker ➔ there exist valid paths in the session declarations that are consistent with all the messages sent and received by the honest principals

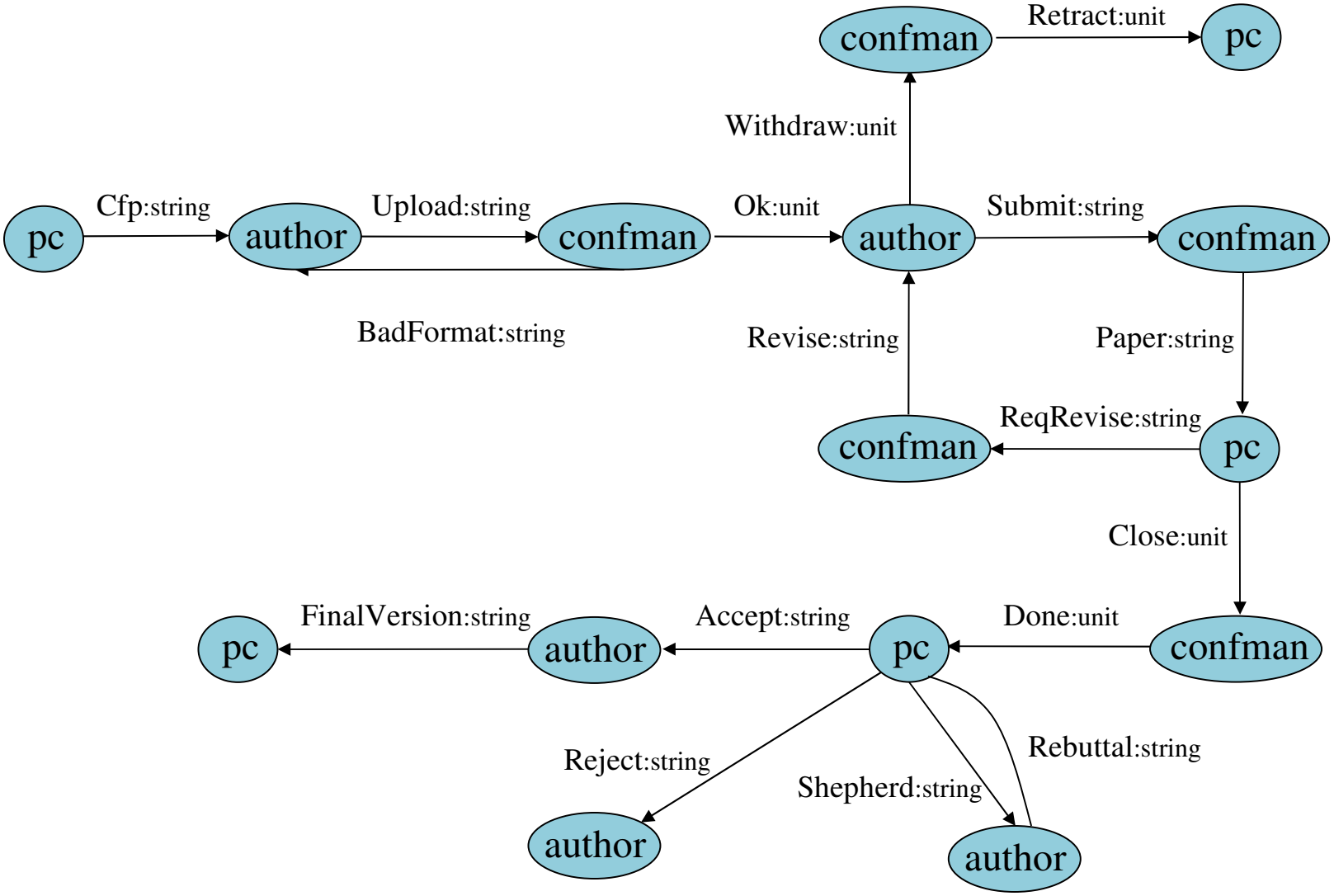
- This generalizes correspondence assertions

Global and Local sessions

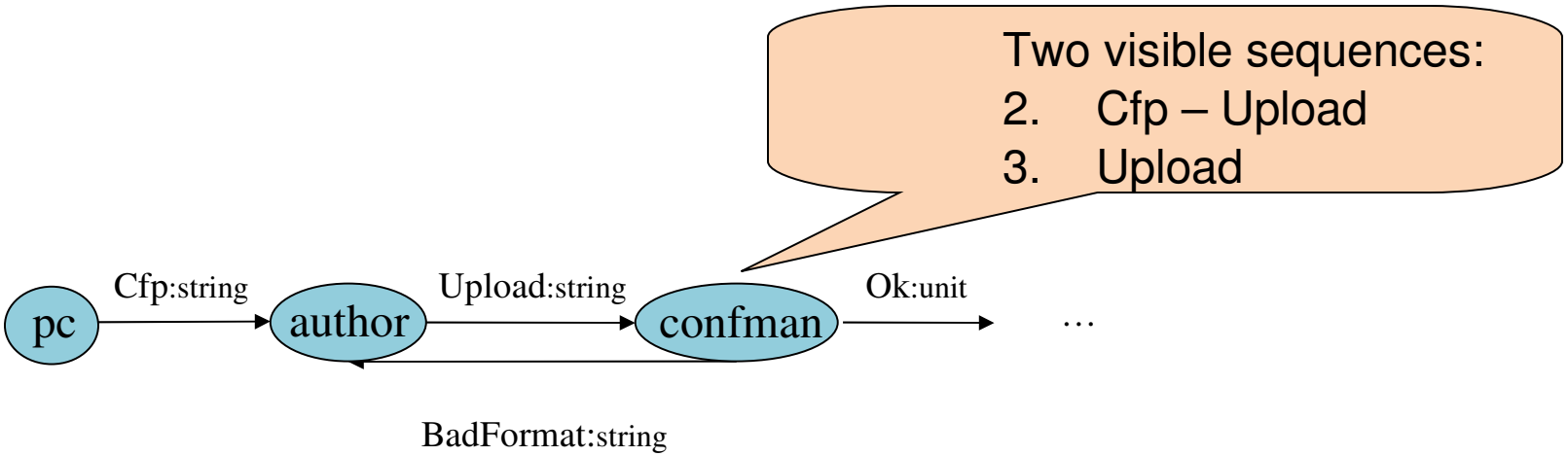
```
Session CMS =
  role pc:string =
    !Cfp:string;
    mu start.
    ?(Paper:string;
      !(ReqRevise:string;
        ?(Change:string;start
        + Accept; !Confirm)
      +Close;?Done;
        mu decision.
        !(Shepherd:string;
          ?Rebuttal:string;decision
        + Accept:string;
            ?FinalVersion:string
        + Reject:string))
    + Retract)

  role author = ( ...)

  role confman = ( ...)
```



A conference management session



Efficient Forwarding



Visibility = minimum information needed to update state of local role

- Can be computed statically from the session graph
  - Any less information would break integrity
  - More work to the compiler = less runtime tests
  - This actually simplifies formal proofs!

Existing approaches

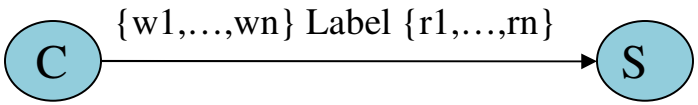
- Session types:
  - First ‘session types’: Pi-calculus based [Honda&Vasconcelos 98, Gay & Hole 99]
    - Describe message flows on single channels
  - ‘Behavioral types’ [Kobayashi & Igarashi 01]
    - Multi channel flows (types are CCS processes)
  - ‘Contracts’ in Singularity OS [Fahndrich et al. 06]
  - ‘Workflows’ in Web services
    - WSDL, WS-SecureConversation [Bhargavan et al. 05]
    - Global interactions [Carbone et al. 07]
- Protocol Analysis, Synthesis, and Transformation :
  - Lots of work, but on abstract, isolated protocols
  - Next challenge: integrate with expressive, real-life (distributed) languages
    - Secure Channels Implementation [Abadi, Fournet, Gonthier

Extensions

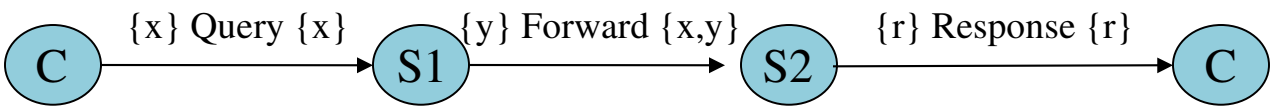
- Improve session expressiveness
  - Enable access control over payloads
    - Roles can deliver data to other roles securely
  - Enable dynamic principal selection
    - As opposed to the initiator picking everyone

Payload secrecy

- Session payloads have “variable” names

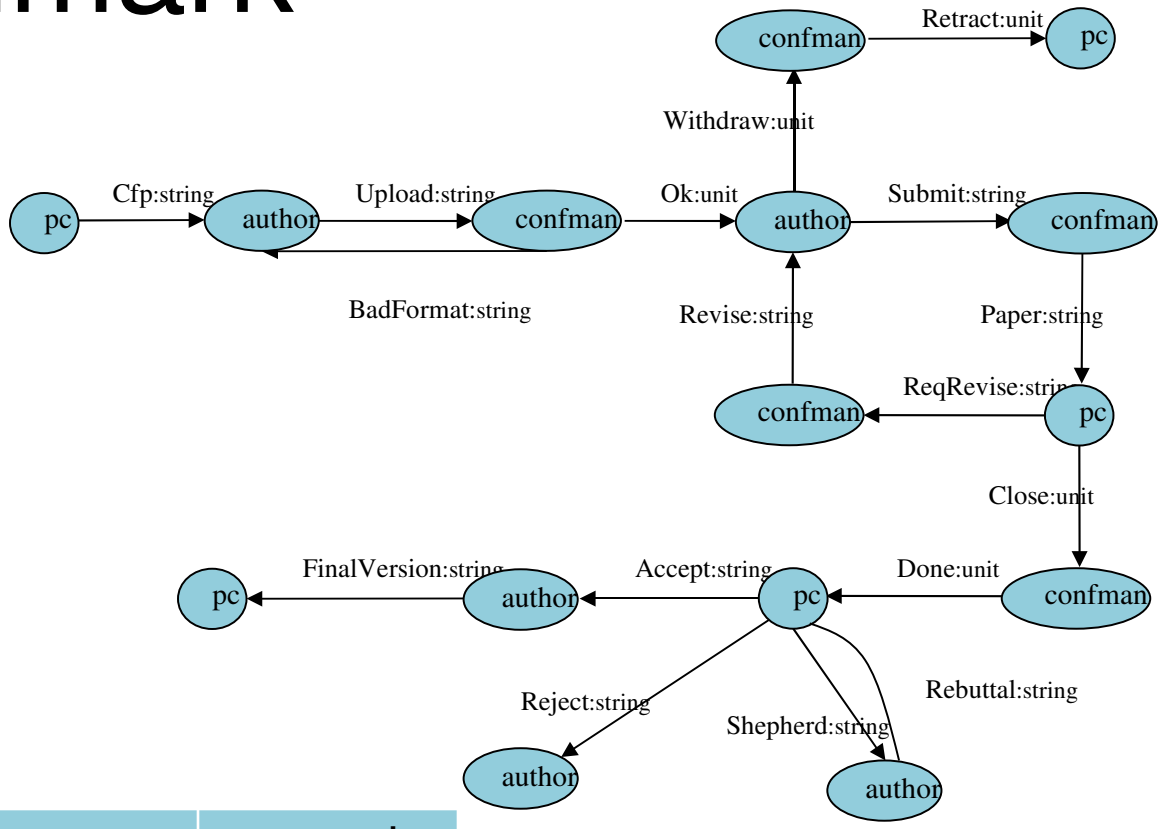


- Example for secrecy: RPC variant



- Here, malicious S1 shouldn’t read “r”, and a malicious C shouldn’t read “y” (assuming everyone else honest)

Benchmark

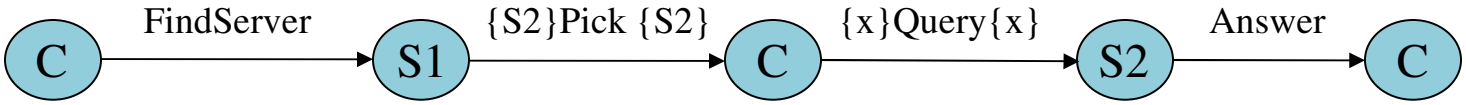


500 iterations in each loop
(4000 messages in total)

	No	crypto	openssl
1st loop	0.23s	2.95s	
2nd loop	0.46s	6.11s	
3rd loop	0.24s	2.98s	
total	0.94s	12.04s	8.38s

Dynamic principal selection

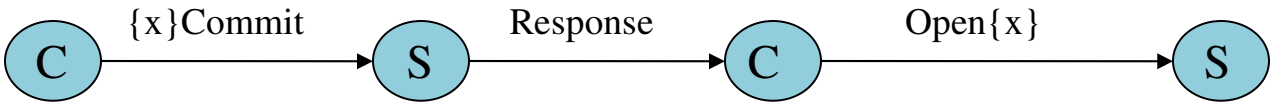
- Example:



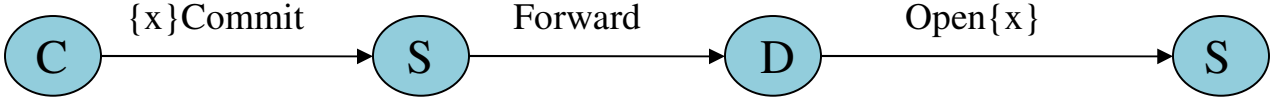
- Here, S1 gets to pick S2

Pitfalls / Challenges

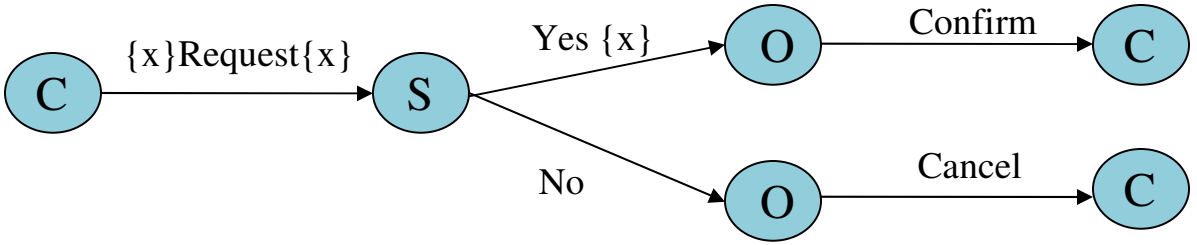
- Commitments



- Delegated Commitments

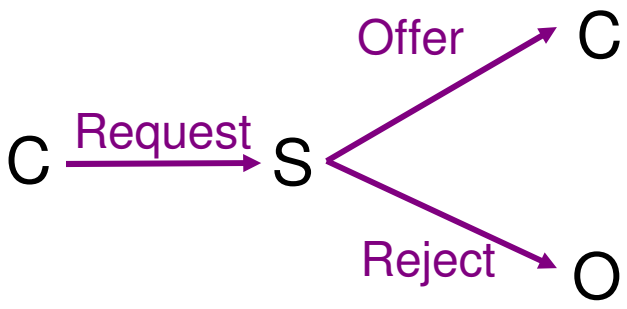


- Forks



Implementability conditions

- Some sessions are always vulnerable:



- We detect them and rule them out
  - They can be turned into safe sessions but only with extra messages

Security Protocol

- We combine standard mechanisms
  - X509 digital signatures
  - Logical timestamps for loop control
  - Anti-replay cache
    - Per principal, based on session identifier Hash(S, **a**, N) + role
- Which evidence to sign & forward?

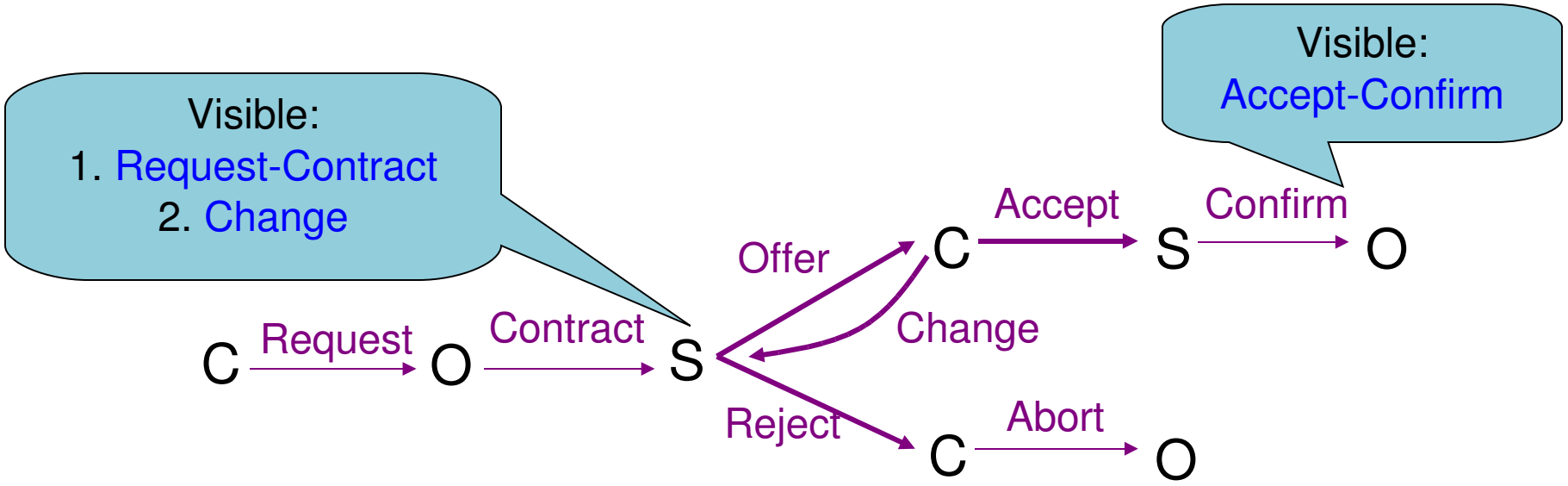
Forwarding history

- Complete history
  - Every sender countersigns the whole history so far
  - Every receiver checks signatures and simulates the history vs. session spec
  - Large overhead (unbounded crypto processing)
- We can do much better

Visibility

- Visibility = minimum information needed to update local role
  - Any less information would break integrity

- Can be computed statically from the session graph
  - More work to the compiler = less runtime tests
  - This actually simplifies formal proofs!



Our session compiler

- Generates interface (types for all messages)
- Generates specific sending and receiving code for each visible sequence
  - Checks exactly what is expected
  - Zero dynamic graph computation

- 5000 lines in F# + dual F# libraries

Dual libraries (CSFW'06)

- Crypto library:

```
type bytes
type keybytes
val nonce: name → bytes
val hash: bytes → bytes
val genskey: name → keybytes
val genvkey: keybytes → keybytes
val sign: bytes → keybytes → bytes
val verify: bytes → bytes → keybytes → bool
```

- Principals library:

```
val skey : principal → keybytes
val vkey : principal → keybytes
val psend : principal → bytes → unit
val precv : principal → bytes
val safe : principal → bool
val psend• : (principal ∗ bytes) chan
val chans• : (principal ∗ bytes chan) list
val skeys• : (principal ∗ bytes) list
```

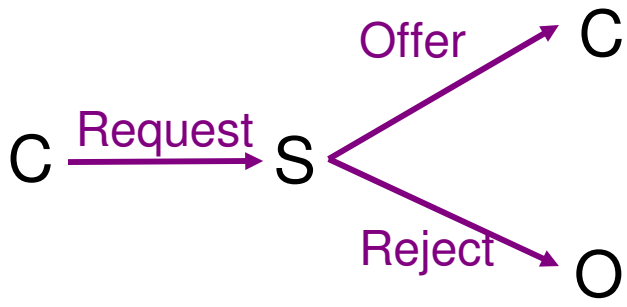
- Dual implementations
  - Symbolic: using algebraic datatypes and type abstraction
  - Concrete: using actual system (.NET) operations

Integrity theorems

- Configuration =
  Libraries + Session Declarations + User Code + Opponent Code

Theorem 1 (Security, reduction-based). If $L\ M_{\widetilde{S}}\ U\ O'$ may fail in F for some O' where ω does not occur, then $L\ \widetilde{S}\ U\ O$ may fail in F+S for some O where ω does not occur.

- counter-example if
  we allowed session forks:



Theorem 2 (Security, labelled-transition based). Let W be a valid implementation of H. For all transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F, where φ represents the observable trace of those transitions, there exists W° valid implementation of H°, such that $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{KD}} W''$ and $W' \rightarrow^*_{\mathsf{KD}} W''$ and $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ$ with φ the translation of ψ.

Discussion

- Session types are an active area of study
  - we address their secure implementation
- Protocol verification:
  - We verify a reference implementation—not a simplified model
  - Our results hold for any number of (concurrent) sessions
  - Even for a single session, this is beyond automated verification tools (loops and branching)
  - Crypto is Dolev-Yao, not far from computational model
  - Integrity, not liveness (so no progress or global termination)
- Related work on secure implementations of process calculi, on automated protocol transformations

Conclusion

- Cryptographic protocols can sometimes be derived (and verified) from application security requirements
  - Strong, simple security model
  - Safer, more efficient than ad hoc design & code

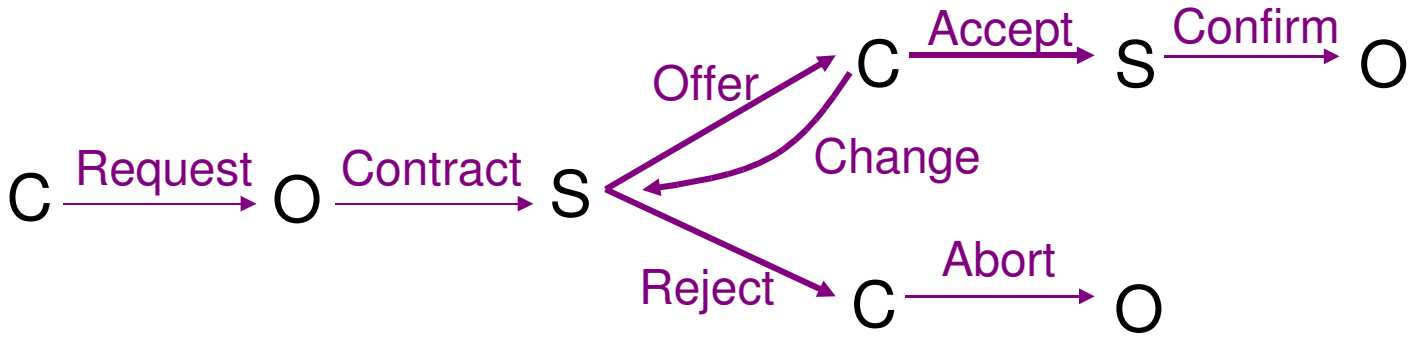
- Try it out now!

http://www.msr-inria.inria.fr/projects/sec/sessions/

Extra Slides

Example: Client-store-officer

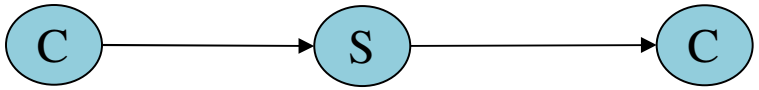
1. A client C requests the delivery of an item on a given date
2. An officer O records the transaction
3. A store S and a client C negotiate more details (e.g. delivery time and place), and inform the officer O



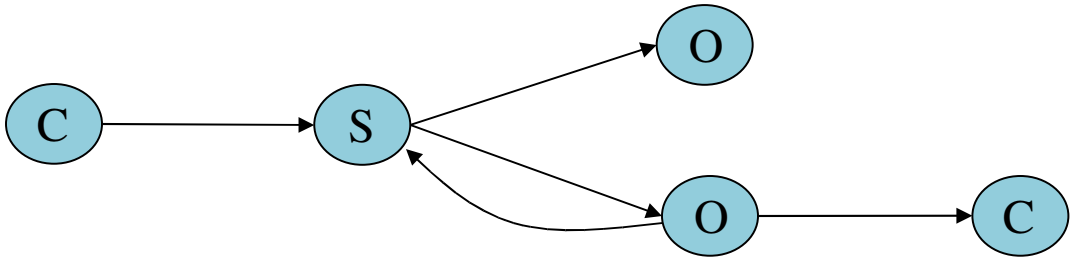
(Only labels displayed)

Programming Distributed applications

- How to program networked independent sites?
  - Little control over the runtime environment
  - Each site has its own code & security concerns
  - Sites may interact, but they do not trust one another
- Communication abstractions can help
  - Hide implementation details (message format, routing,…)
  - Basic communication patterns, e.g. RPCs or private channels

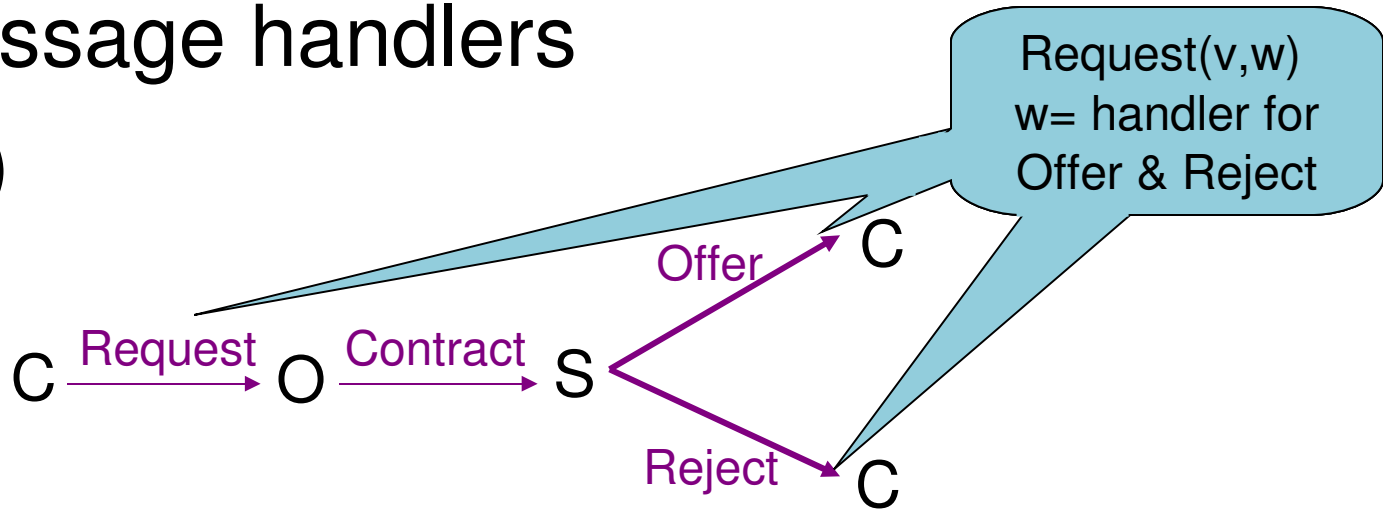


  - Sessions, (aka protocols, or contracts, or workflows)



Using a compiled session: the customer

- User code needs to provide message handlers (CPS style)




```
type msg0 =
    Request of (string * msg1)
and msg1 = {
    hOffer : (principals → string → msg2) ;
    hReject : (principals → unit → msg4)}
and ...
```

Labelled session semantics

(SESSION) $\rho, \mathsf{session}\ S = \Sigma\ \mathsf{in}\ e \rightarrow_e \rho \uplus \{S = \Sigma\}, e$ up to renamings of S

(INIT) $$\frac{p_0 \xrightarrow{\overline{g}}_r p' \quad S = (r_i : \widetilde{\tau}_i = p_i)_{i<n} \in \rho \quad s\ \mathit{fresh}}{\rho, S.r_0^b\ (a_i)_{i<n} \xrightarrow{\overline{g}}_s \rho \uplus \{s\ (a_i)_{i<n}\ \{r_0\} : S\}, s.p'}$$

(JOIN) $$\frac{p_j \xrightarrow{f}_r p' \quad S = (r_i : \widetilde{\tau}_i = p_i)_{i<n} \in \rho \quad \rho' = \rho \uplus \{s\ (a_i)_{i<n}\ \delta : S\}}{\rho', S.r_j^b\ a_j \xrightarrow{f}_s \rho \uplus \{s\ (a_i)_{i<n}\ (\delta \uplus \{r_j\}) : S\}, s.p'}$$

We use a centralized session monitor

(STEP) $$\frac{p \xrightarrow{\eta}_r p'}{\rho, s.p \xrightarrow{\eta}_s \rho, s.p'}$$

(SENDS) $$\frac{\rho, \sigma \xrightarrow{\overline{g}}_s \rho', s.p \quad \mathsf{safe}\ \sigma}{\rho, \sigma\ (g(\widetilde{v}), w) \xrightarrow{s\overline{g}\ \widetilde{v}}_e \rho', s.p\ (w)}$$

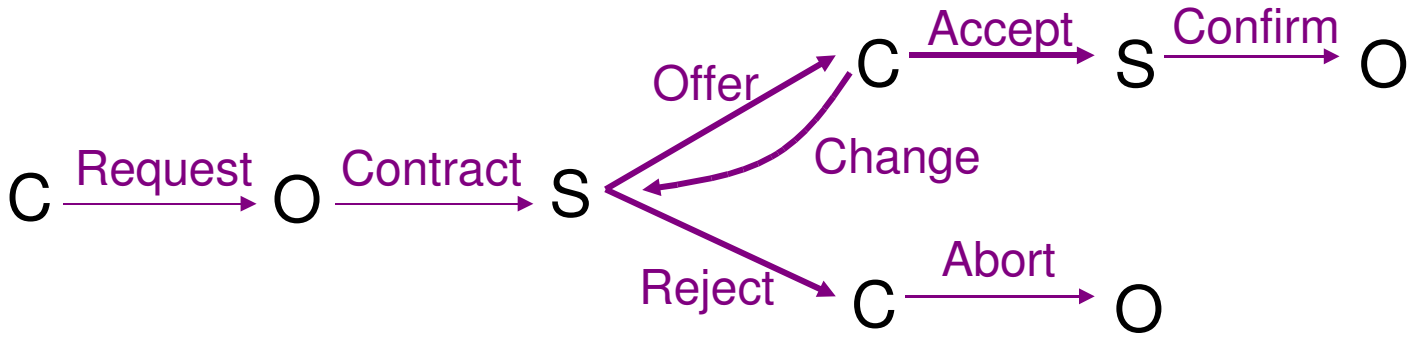
(RECVS) $$\frac{\rho, \sigma \xrightarrow{g}_s \rho', s.p \quad s\ \widetilde{a}\ \delta : S \in \rho \quad \mathsf{safe}\ \sigma}{\rho, \sigma\ (w) \xrightarrow{sg\ \widetilde{v}}_e \rho', s.p\ (w.g\ \widetilde{a}\ \widetilde{v})}$$

These rules are our sessions spec!

(ENDS) $\rho, s.0\ (v) \rightarrow_e \rho, v$

Example: Client-store-officer

1. A client C requests the delivery of an item on a given date
2. An officer O records the transaction
3. A store S and a client C negotiate more details (e.g. delivery time and place), and inform the officer O

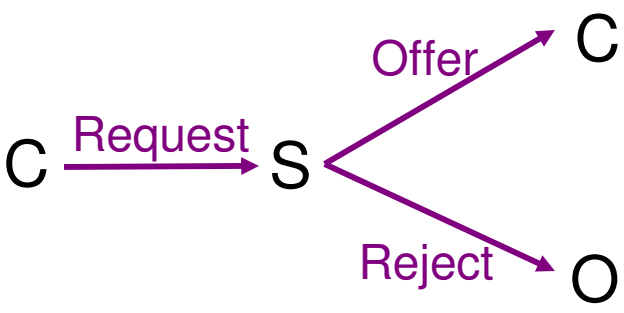


(Only labels displayed)

Results

Theorem 1 (Security, reduction-based). If $L\ M_{\widetilde{S}}\ U\ O'$ may fail in F for some O' where ω does not occur, then $L\ \widetilde{S}\ U\ O$ may fail in F+S for some O where ω does not occur.

- Counter example if we allow forks:




```
let pr = { client = "Alice"; server = "Eve"; officer = "Bob"; } in
let x = new() in
let acceptbranch _ _ = send x "OK" in
let rejectbranch pr' _ = if pr = pr' then let _ = recv x in send ω () in
let office () = S.officer "Bob" {hReject=rejectbranch} in
fork office;
S.client pr (Request (42,{hAccept=acceptbranch}))
```

Theorem 2 (Security, labelled-transition based). Let W be a valid implementation of H. For all transitions $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W'$ in F, where φ represents the observable trace of those transitions, there exists W° valid implementation of H°, such that $W \overset{\varphi}{\Rightarrow}_{\mathsf{K}} W^\circ \rightarrow^*_{\mathsf{KD}} W''$ and $W' \rightarrow^*_{\mathsf{KD}} W''$ and $H \overset{\psi}{\Rightarrow}_{\mathsf{K}} H^\circ$ with φ the translation of ψ.