A Protocol Compiler for Secure Sessions in ML

Ricardo Corin, Pierre-Malo Deniélou

INRIA—Microsoft Research Joint Centre

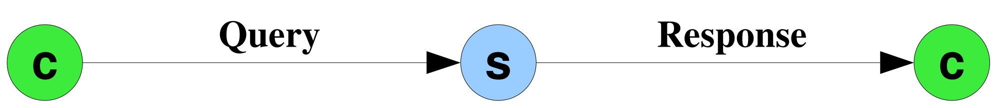
http://www.msr-inria.inria.fr/projects/sec/sessions/

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

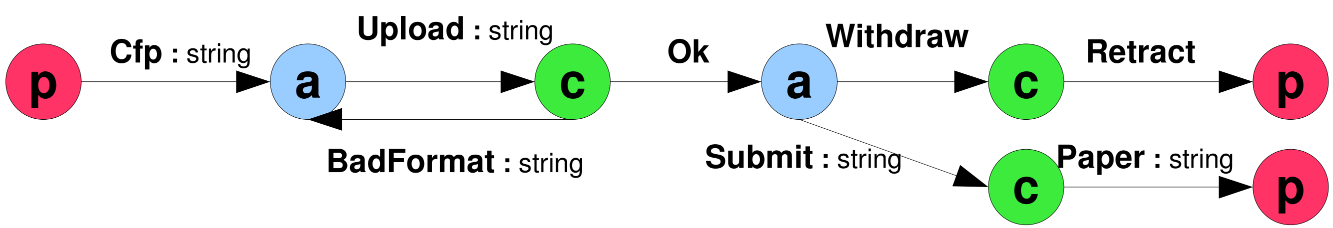


  - They hide implementation details

    (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.



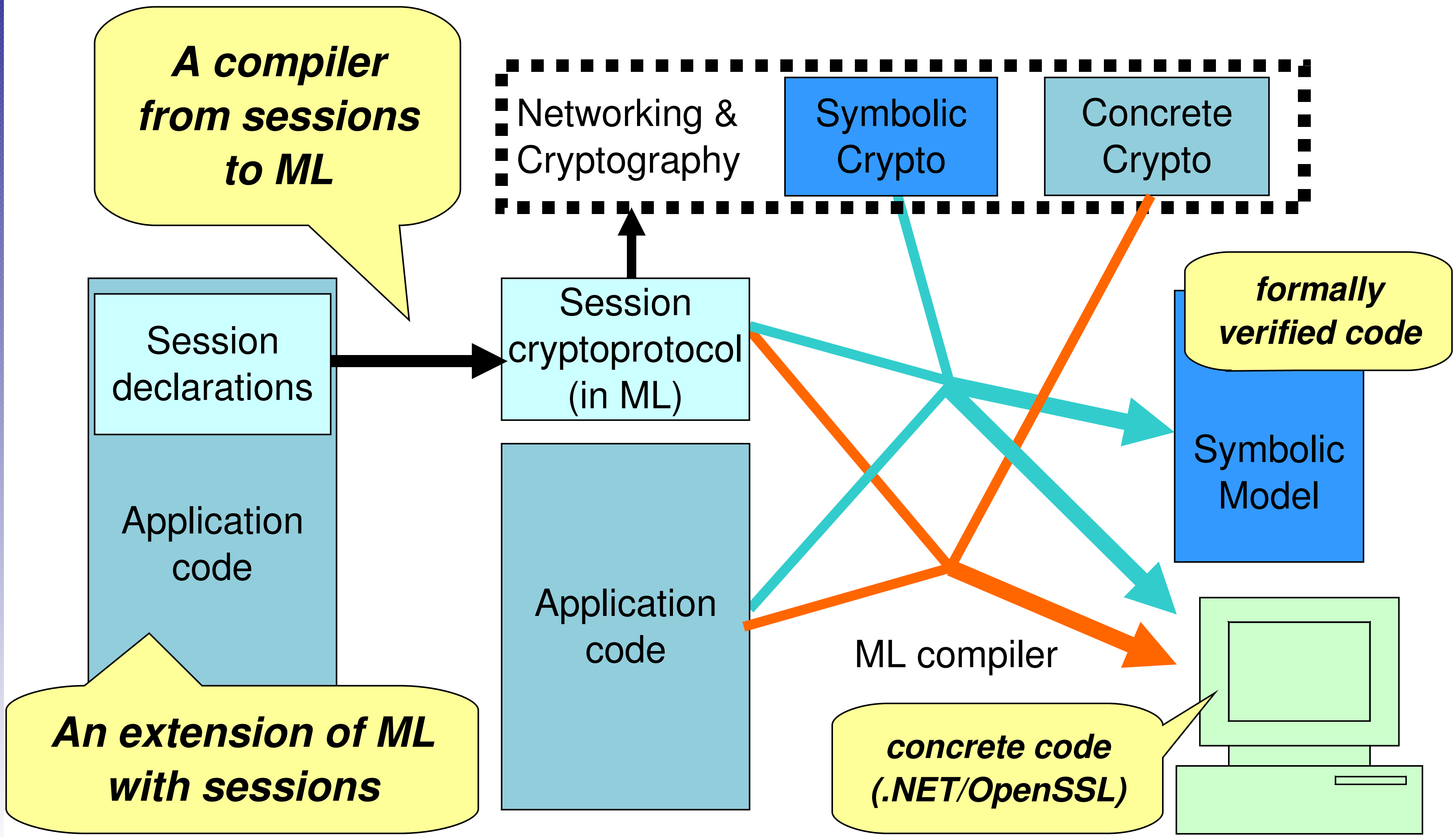
  - Active area for distributed programming
    - A.k.a. protocols, or contracts, or workflows
    - Pi calculus settings, web services, operating systems
    - Common strategy: type systems enforce protocol compliance
      "If every site program is well-typed, sessions follow their spec"

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

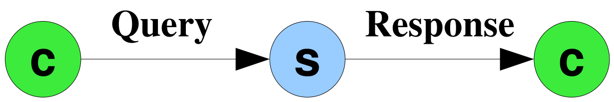
I. Programming with Sessions
   1. Language description
   2. Session usage and interface generation

II. Compiler internals
   1. Security protocol
   2. Module generation

A small session language

τ ::=	Payload types
unit | int | string	base types
p ::=	Role processes
$!(f_i : \tau_i \ ; \ p_i)_{i<k}$	send
$?(f_i : \tau_i \ ; \ p_i)_{i<k}$	receive
$\mu\chi.p$	recursion declaration
χ	recursion
0	end
Σ ::=	Sessions
$(r_i : T_i = p_i)_{i<n}$	initial role processes

A very simple RPC session:

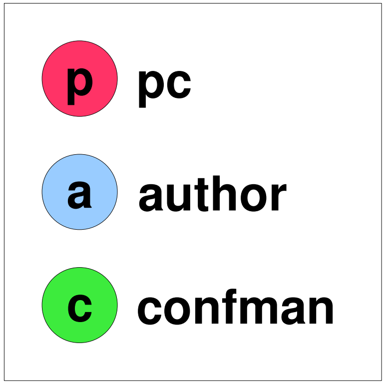


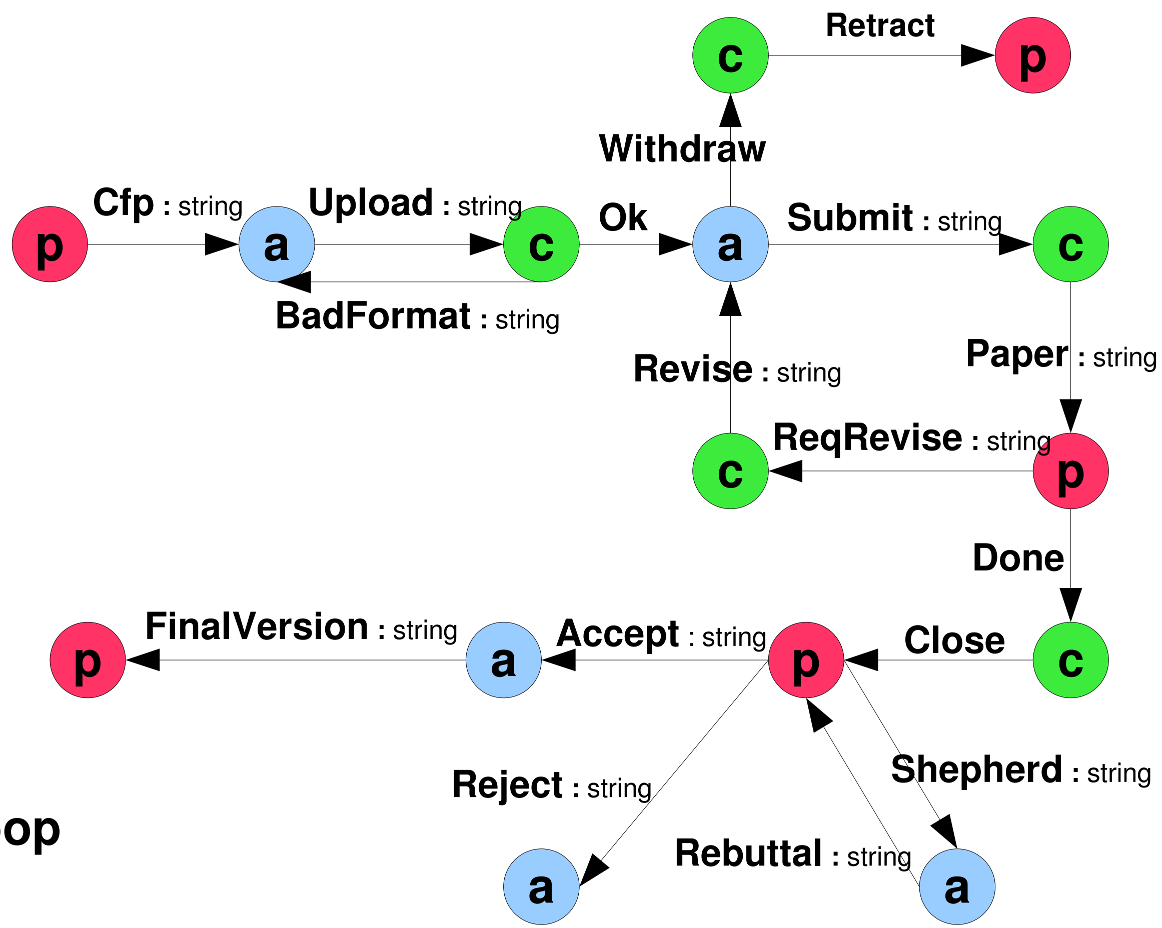
Session RPC =

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

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

A Conference Management Session



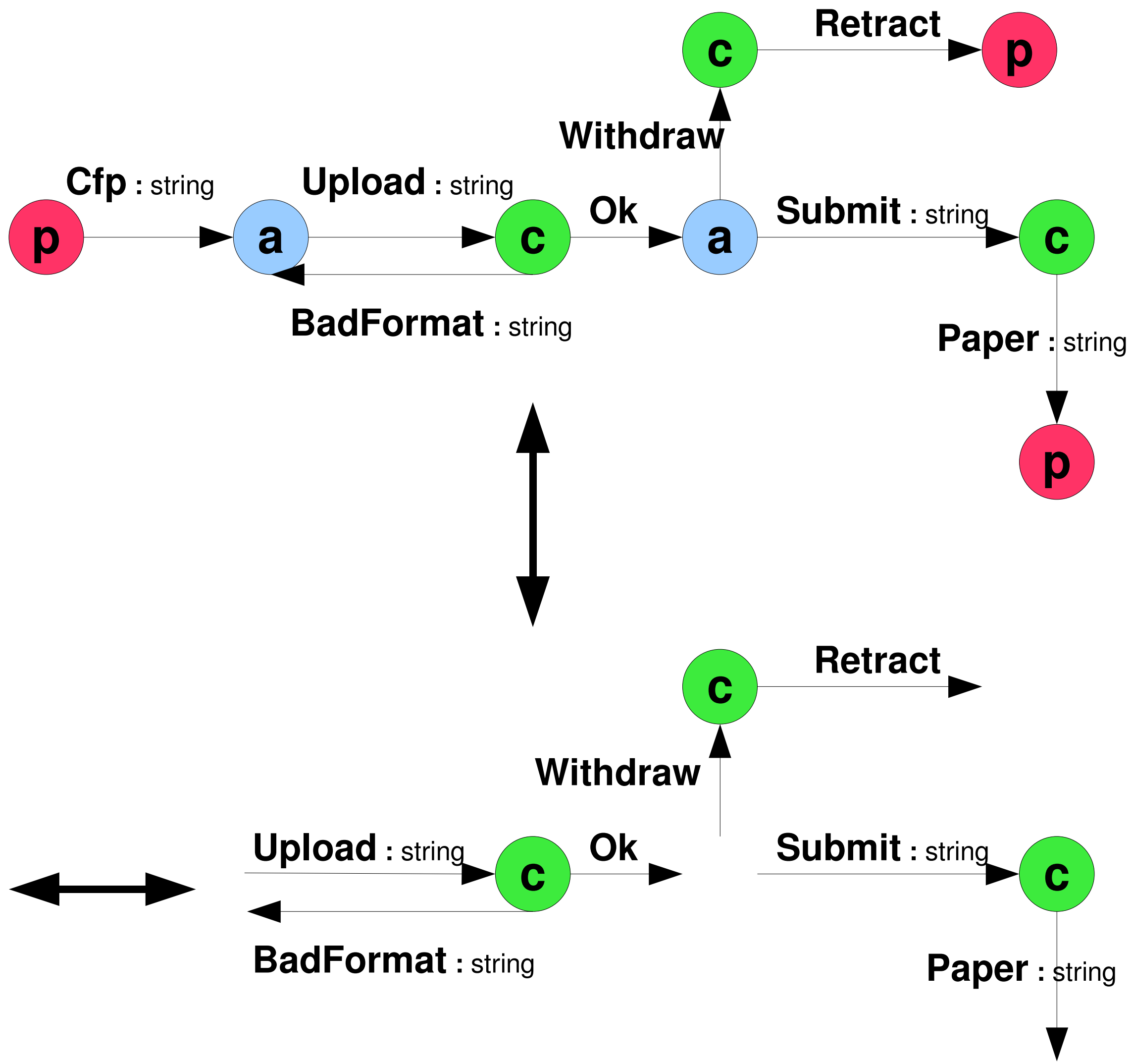


1. Call for paper
2. Upload sequence
3. Revision loop
4. Decision & Rebuttal Loop

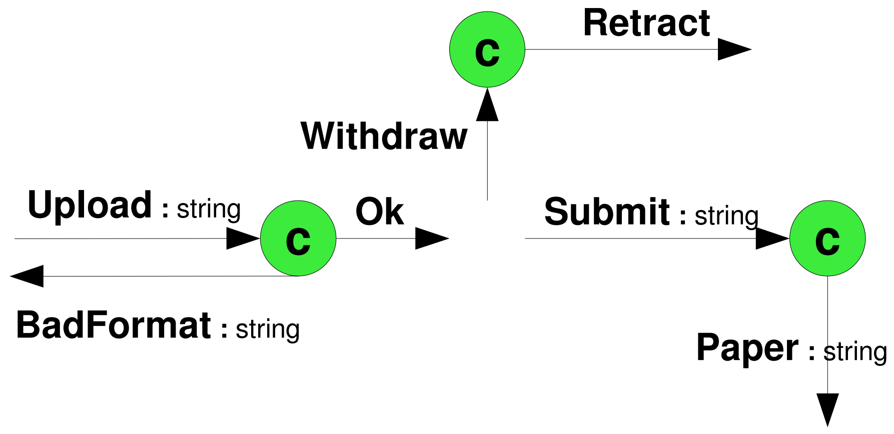
Global and Local sessions

```
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`

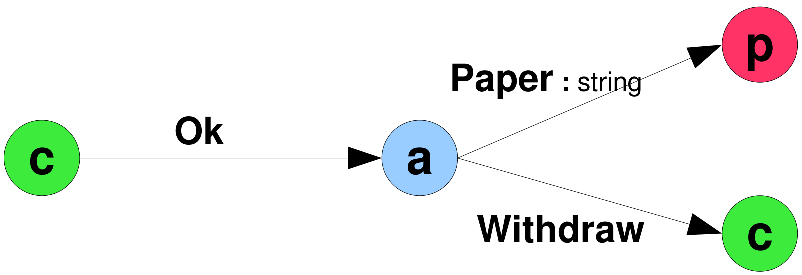
```
open CMS

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`

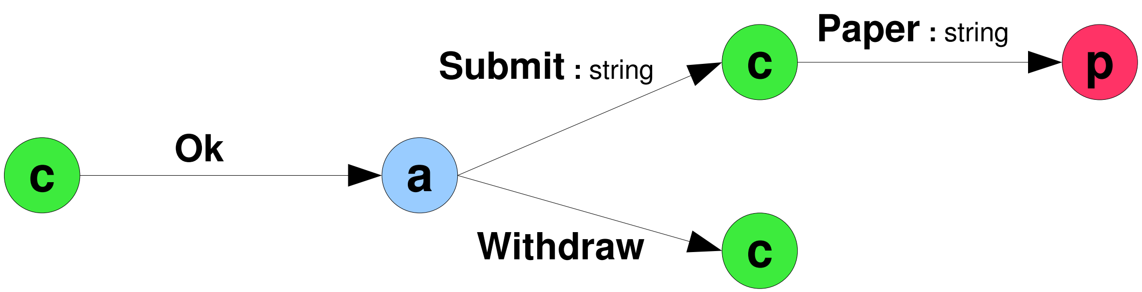
Ordinary ML type-checking provides functional guarantees!

Implementability conditions

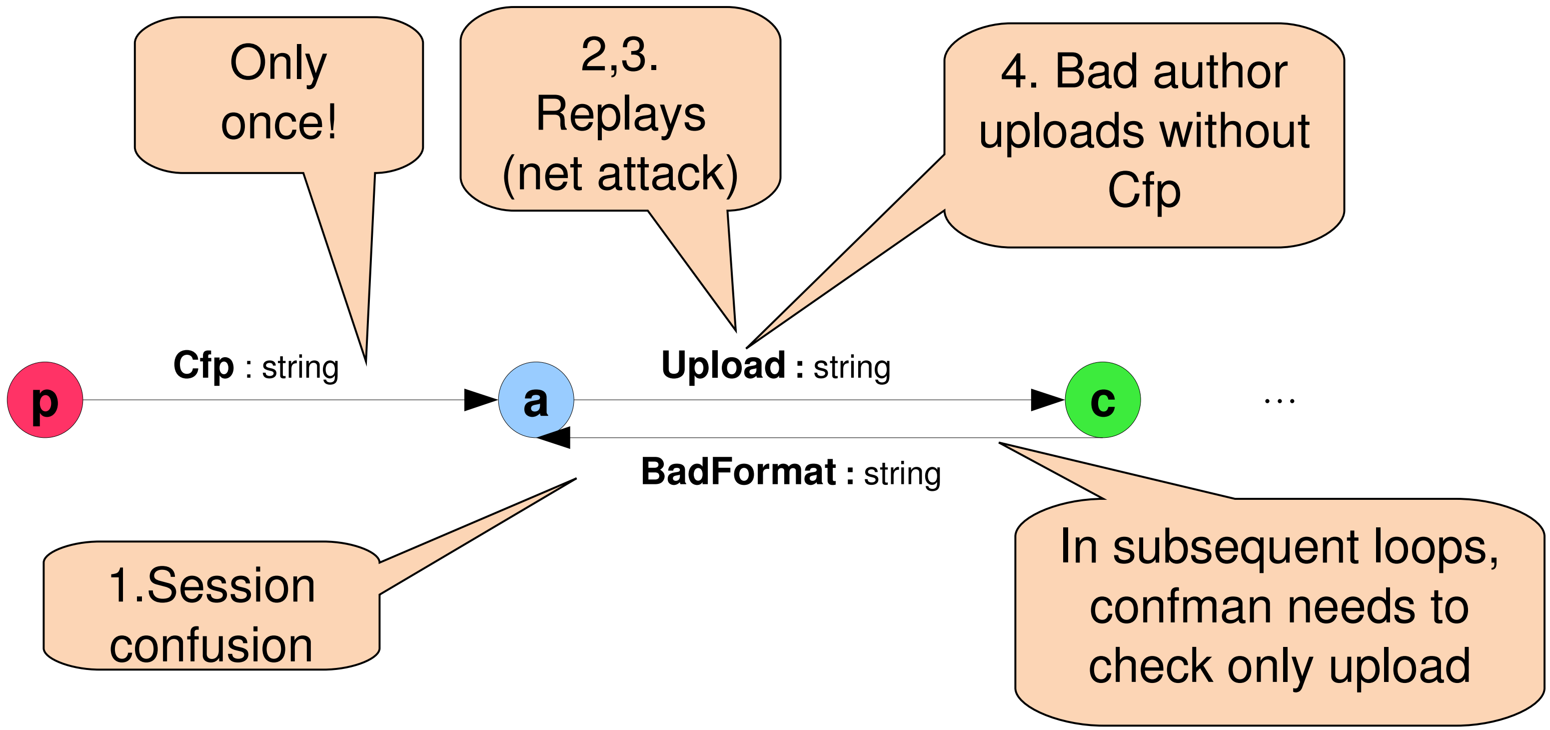
- We want session integrity.
- Some sessions are always vulnerable:



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



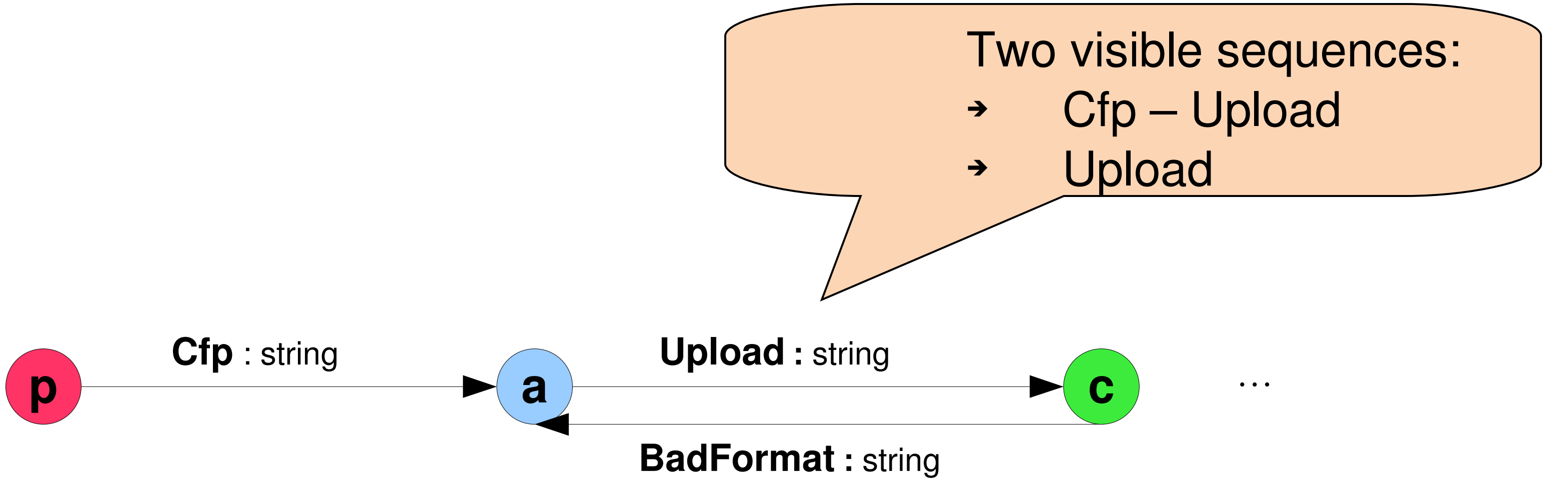
Protocol outline & (Potential) attacks



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

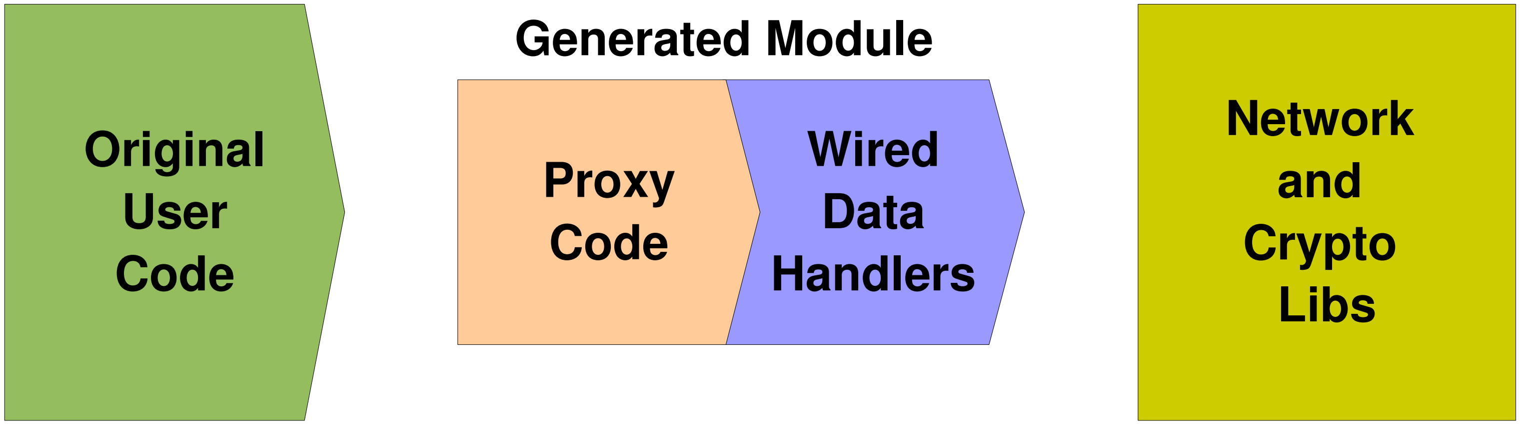
- For any run of any choice of honest principals running roles of compiled session declarations plus any coalition of dishonest principals + network attacker

  ➔ 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

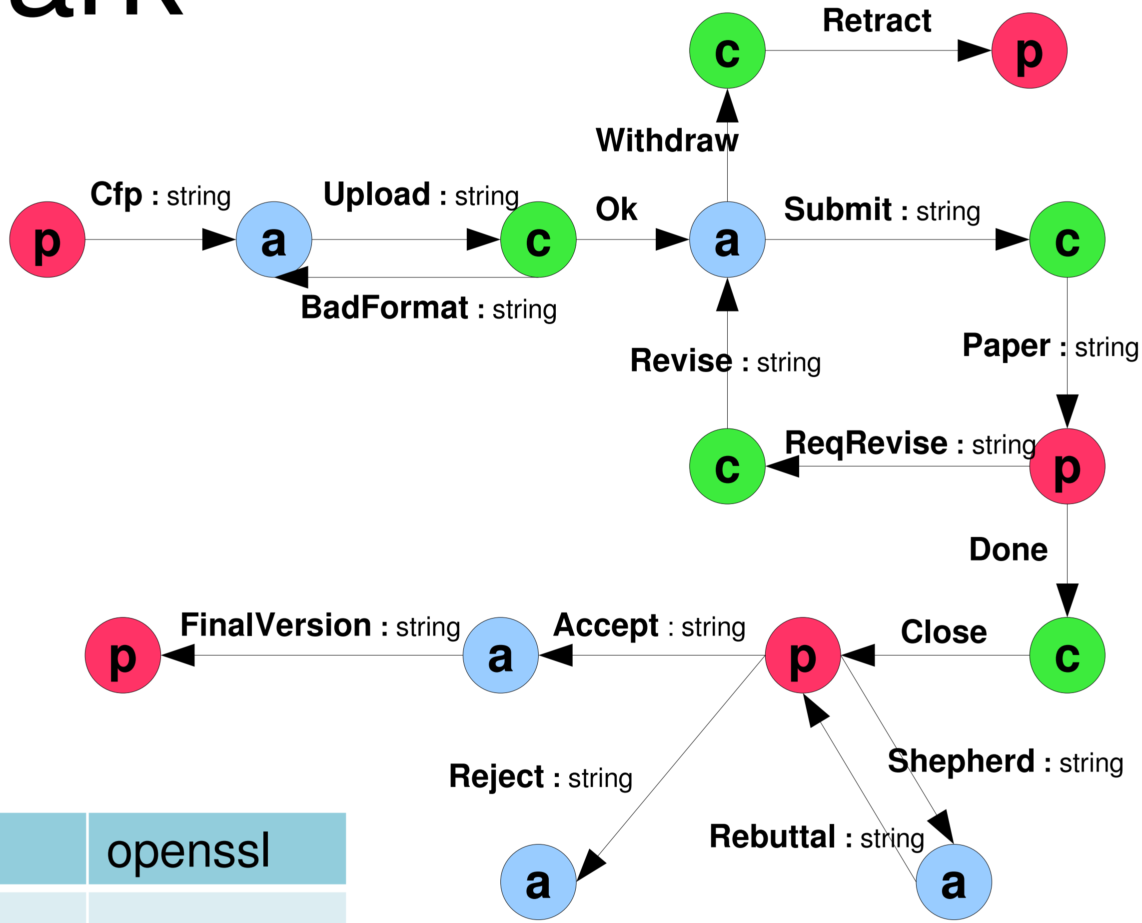
```
type msg11 = {
  hUpload : (principals -> string -> msg12)}
and msg12 =
  | BadFormat of string * msg11
  | Ok of unit * msg13
and msg13 = {
  hSubmit : (principals -> string -> msg14);
  hWithdraw : (principals -> unit -> msg15)}
and msg14 = Paper of string * unit
and msg15 = Retract of string *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