From CryptoVerif Specifications to Computationally Secure Implementations of Protocols

Bruno Blanchet
joint work with David Cadé

INRIA Paris
Bruno.Blanchet@inria.fr

Year 2022-23

Protocol verification

	Symbolic	Computational
Specifications	FDR, AVISPA, ProVerif, Scyther, Tamarin, ...	CryptoVerif, CertiCrypt, EasyCrypt, ...
Implementations	FS2PV, F7, Spi2Java, C2PV, ...	FS2CV, Computational F7, C2CV, , ...

Protocol verification

	Symbolic	Computational
Specifications	FDR, AVISPA, ProVerif, Scyther, Tamarin, ...	CryptoVerif, CertiCrypt, EasyCrypt, ...
Implementations	FS2PV, F7, Spi2Java, C2PV, ...	FS2CV, Computational F7, C2CV, our work, ...

Our approach

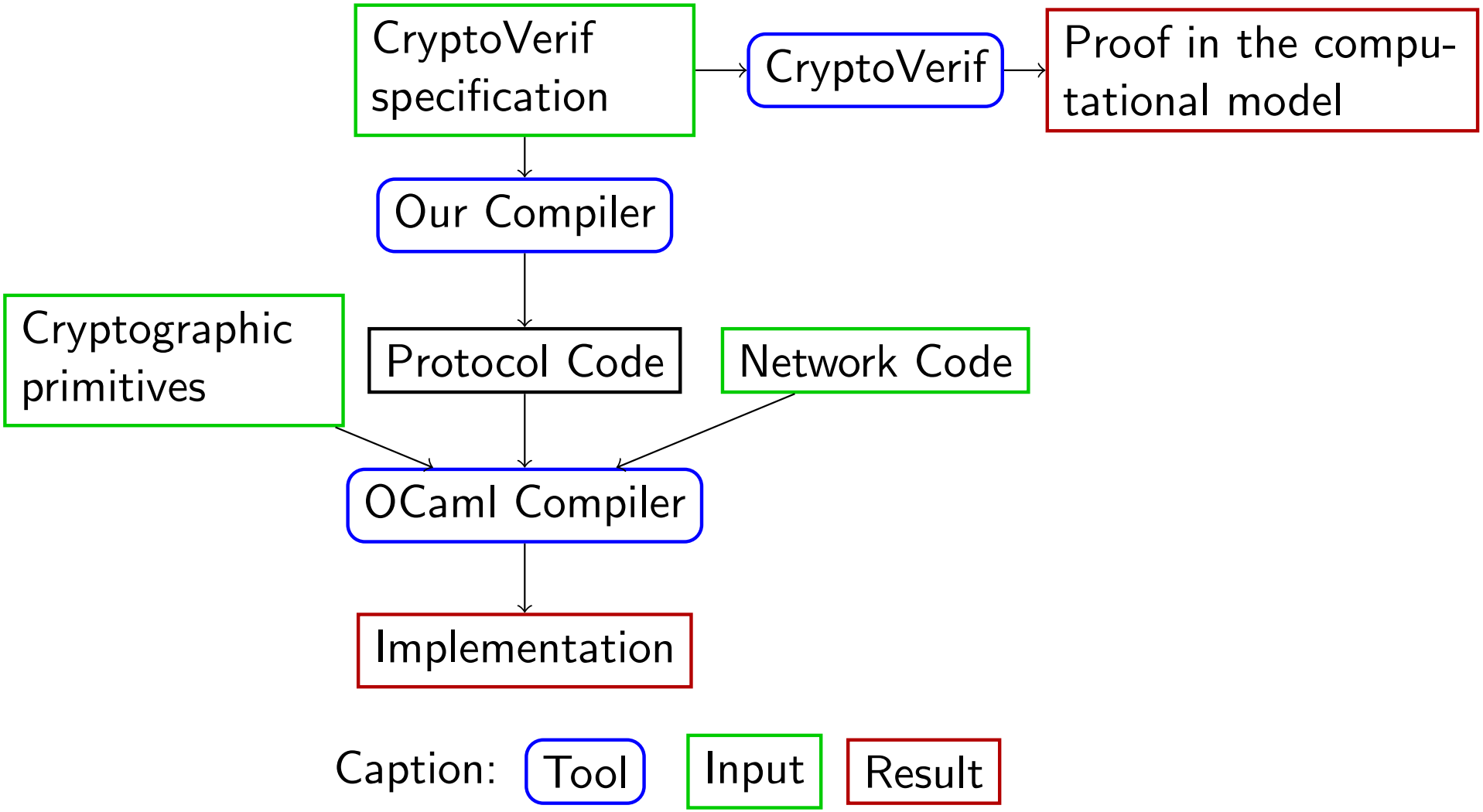
Generate protocol implementations from specifications.

- Specification proved secure in the computational model by CryptoVerif.
- Specification translated into an OCaml implementation by our compiler.

Goal: proved implementations of cryptographic protocols.

Remark: FS2CV does the translation in the other direction!

Overview of our approach



Choice of the target language

- Why OCaml?
  - Memory safe. Easier to show that the network code does not access the protocol memory.
  - Clean semantics.
  - Crypto library available.
- Writing a compiler into another language would not be difficult.

  Proving the security of the generated protocol may be more difficult.

The CryptoVerif specification language: terms

CryptoVerif represents protocols and games in a process calculus.

$M, N ::=$	terms
x	variable
$f(M_1, \ldots, M_m)$	function application

Function symbols f correspond to functions computable by polynomial-time deterministic Turing machines.

The CryptoVerif specification language: processes

$Q ::=$	input process
0	nil
$Q \mid Q'$	parallel composition
$!^{i \leq n}\ Q$	replication n times
$\mathsf{in}(c, x_1 : T_1, \ldots, x_k : T_k); P$	input
$P ::=$	output process
$\mathsf{out}(c, M_1, \ldots, M_k); Q$	output
yield	end
$\mathsf{new}\ x : T; P$	random number
$\mathsf{let}\ x : T = M\ \mathsf{in}\ P$	assignment
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P'$	conditional
$\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P$	insert in table
$\mathsf{get}\ Tbl(x_1 : T_1, \ldots, x_k : T_k)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$	get from table

Example

$$\mathrm{A} \longrightarrow \mathrm{B} : \mathrm{enc}(r, Kab)$$

$$\begin{array}{l} \text{process in}(c_{start}, ()); \text{new } Kab : key; \text{out}(c, ()); \\ \quad (!^{i1 \leq N} \; processA(Kab) \mid \\ \quad\;\; !^{i2 \leq N} \; processB(Kab)) \end{array}$$

- The process generates Kab.
- This symmetric key will not be known by the opponent.
- Only after the key has been generated, we can call at most N times *processA* and at most N times *processB*.

Example

$$A \longrightarrow B : \mathrm{enc}(r, Kab)$$

$$\begin{aligned}
&\mathsf{let}\ processA(Kab) = \\
&\quad \mathsf{in}(c_A, ()); \mathsf{new}\ r : nonce; \mathsf{new}\ s : seed; \\
&\quad \mathsf{out}(c_A, \mathrm{enc}(\mathrm{nonceToBitstring}(r), Kab, s)).
\end{aligned}$$

$$\begin{aligned}
&\mathsf{let}\ processB(Kab) = \\
&\quad \mathsf{in}(c_B, m : bitstring); \\
&\quad \mathsf{let}\ \mathrm{injbot}(\mathrm{nonceToBitstring}(r' : nonce)) = \mathrm{dec}(m, Kab)\ \mathsf{in} \\
&\quad \mathsf{out}(c_B, ()).
\end{aligned}$$

- *processA* sends the encryption of r under Kab (probabilistic encryption)
- *processB* decrypts the received message

Example — summary

$$\begin{array}{l}
\text{let } \mathit{processA}(\mathit{Kab}) = \\
\quad \text{in}(c_A, ()); \text{new } r : \mathit{nonce}; \text{new } s : \mathit{seed}; \\
\quad \text{out}(c_A, \text{enc}(\text{nonceToBitstring}(r), \mathit{Kab}, s)). \\
\\
\text{let } \mathit{processB}(\mathit{Kab}) = \\
\quad \text{in}(c_B, m : \mathit{bitstring}); \\
\quad \text{let injbot}(\text{nonceToBitstring}(r' : \mathit{nonce})) = \text{dec}(m, \mathit{Kab}) \text{ in} \\
\quad \text{out}(c_B, ()). \\
\\
\text{process in}(c_{start}, ()); \text{new } \mathit{Kab} : \mathit{key}; \text{out}(c, ()); \\
\quad (!^{i1 \leq N} \ \mathit{processA}(\mathit{Kab}) \mid \\
\quad\ !^{i2 \leq N} \ \mathit{processB}(\mathit{Kab}))
\end{array}$$

Annotations: Separation in multiple programs

$$\begin{array}{l}
\text{let } \mathit{processA}(\mathit{Kab}) = \\
\quad \mathit{pA}\{\text{in}(c_A, ()); \text{new } r : \mathit{nonce}; \text{new } s : \mathit{seed}; \\
\quad \text{out}(c_A, \text{enc}(\text{nonceToBitstring}(r), \mathit{Kab}, s))\}. \\
\\
\text{let } \mathit{processB}(\mathit{Kab}) = \\
\quad \mathit{pB}\{\text{in}(c_B, m : \mathit{bitstring}); \\
\quad \text{let injbot}(\text{nonceToBitstring}(r' : \mathit{nonce})) = \text{dec}(m, \mathit{Kab}) \text{ in} \\
\quad \text{out}(c_B, ())\}. \\
\\
\text{process } \mathit{keygen}\ [\mathit{Kab} > \mathit{fileKab}]\ \{\text{in}(c_{\mathit{start}}, ()); \text{new } \mathit{Kab} : \mathit{key}; \text{out}(c, ())\}; \\
\quad (!^{i1 \leq N}\ \mathit{processA}(\mathit{Kab}) \mid \\
\quad\ !^{i2 \leq N}\ \mathit{processB}(\mathit{Kab}))
\end{array}$$

Annotations: External data files

$$\begin{array}{l}
\text{let } \mathit{processA}(\mathit{Kab}) = \\
\quad \mathit{pA}\{\text{in}(c_A, ()); \text{new } r : \mathit{nonce}; \text{new } s : \mathit{seed}; \\
\quad \text{out}(c_A, \text{enc}(\text{nonceToBitstring}(r), \mathit{Kab}, s))\}.
\end{array}$$

$$\begin{array}{l}
\text{let } \mathit{processB}(\mathit{Kab}) = \\
\quad \mathit{pB}\{\text{in}(c_B, m : \mathit{bitstring}); \\
\quad \text{let injbot}(\text{nonceToBitstring}(r' : \mathit{nonce})) = \text{dec}(m, \mathit{Kab}) \text{ in} \\
\quad \text{out}(c_B, ())\}.
\end{array}$$

$$\begin{array}{l}
\text{process } \mathit{keygen}\ [\mathit{Kab} > \mathit{fileKab}]\ \{\text{in}(c_{start}, ()); \text{new } \mathit{Kab} : \mathit{key}; \text{out}(c, ())\}; \\
\quad (!^{i1 \leq N}\ \mathit{processA}(\mathit{Kab}) \mid \\
\quad\ !^{i2 \leq N}\ \mathit{processB}(\mathit{Kab}))
\end{array}$$

Annotations: types and functions

- OCaml type representing a CryptoVerif type:
  implementation type *keyseed* = 128. (bitstring of 128 bits)
  implementation type *host* = "*string*" [serial = "*id*", "*id*"].
- OCaml function representing a function in the protocol specification:
  implementation fun enc = "*sym_enc*".
  implementation fun injbot = "*injbot*" [inverse = "*injbot_inv*"].
  - In the CryptoVerif specification, there are assumptions about these functions.
    - Functional assumptions: $\mathrm{dec}(\mathrm{enc}(m, k, s), k) = \mathrm{injbot}(m)$.
    - Security assumptions: encryption is IND-CPA and INT-CTXT.
  - These assumptions must be manually verified.

Annotations: tables

- get/insert handle tables of keys:
  - insert $keytbl(h, k)$
    inserts element h, k in the table $keytbl$.
  - get $keytbl(h', k')$ suchthat $h' = h$ in P else P'
    stores in h', k' an element of table $keytbl$ such that $h' = h$, i.e., stores in k' the key of h, and runs P.
    Runs P' when no such element exists.
- Tables are stored in files:
  implementation table $keytbl = "filekeytbl"$.

Treatment of tables in CryptoVerif

For proving the protocol, CryptoVerif encodes tables as arrays:

- The variables are considered as arrays with one cell for each copy of the definition.
  - Useful for remembering all values taken by the variable.
- $!^{i \leq n} \ldots \text{insert } keytbl(h, k)$
  becomes
  $!^{i \leq n} \ldots \text{let } keytbl_1[i] = h \text{ in let } keytbl_2[i] = k \text{ in}$
- $\text{get } keytbl(h', k') \text{ suchthat } h' = h \text{ in } P \text{ else } P'$
  becomes
  $\text{find } u \leq n \text{ suchthat defined}(keytbl_1[u], keytbl_2[u]) \wedge keytbl_1[u] = h$
  $\text{then let } h' = keytbl_1[u] \text{ in let } k' = keytbl_2[u] \text{ in } P \text{ else } P'$

Treatment of tables in CryptoVerif

For proving the protocol, CryptoVerif encodes tables as arrays:

- The variables are considered as arrays with one cell for each copy of the definition.
  - Useful for remembering all values taken by the variable.
- $!^{i \leq n} \ldots \text{insert } keytbl(h, k)$
  becomes
  $!^{i \leq n} \ldots \text{let } keytbl_1[i] = h \text{ in let } keytbl_2[i] = k \text{ in}$
- $\text{get } keytbl(h', k') \text{ suchthat } h' = h \text{ in } P \text{ else } P'$
  becomes
  $\text{find } u \leq n \text{ suchthat defined}(keytbl_1[u], keytbl_2[u]) \wedge keytbl_1[u] = h$
  $\text{then let } h' = keytbl_1[u] \text{ in let } k' = keytbl_2[u] \text{ in } P \text{ else } P'$
- Generalized to several insertions by looking up in the variables defined at each insertion.

Treatment of tables in CryptoVerif

For proving the protocol, CryptoVerif encodes tables as arrays:

- The variables are considered as arrays with one cell for each copy of the definition.
  - Useful for remembering all values taken by the variable.
- $!^{i \leq n} \ldots \text{insert } keytbl(h, k)$
  becomes
  $!^{i \leq n} \ldots \text{let } keytbl_1[i] = h \text{ in let } keytbl_2[i] = k \text{ in}$
- $\text{get } keytbl(h', k') \text{ suchthat } h' = h \text{ in } P \text{ else } P'$
  becomes
  $\text{find } u \leq n \text{ suchthat defined}(keytbl_1[u], keytbl_2[u]) \wedge keytbl_1[u] = h$
  $\text{then let } h' = keytbl_1[u] \text{ in let } k' = keytbl_2[u] \text{ in } P \text{ else } P'$
- Generalized to several insertions by looking up in the variables defined at each insertion.

Avoiding arrays is more intuitive and simplifies the compilation.

Compilation to OCaml

For each program, the compiler generates an OCaml module where it defines a function for each input.

- A function `init` : `unit` $\rightarrow \tau$ returns the tuple of functions representing the input processes available at the beginning of the program.
  - `init` may also read variables from files when needed.
- Each input Q is represented by a function that
  - takes as argument the arguments of the input Q
  - and returns
    - the tuple of functions representing input processes that follow Q,
    - the result output at the end of Q.

Compilation to OCaml: example

$$\text{let } processA(Kab) = pA\{\text{in}(c_A, ()); \text{new } r : nonce; \text{new } s : seed; \\ \text{out}(c_A, \text{enc}(\text{nonceToBitstring}(r), Kab, s))\}.$$

The generated module PA has the following interface :

```
open Base
open Crypto

type type_processA = unit -> (unit * string)
val init :  unit -> type_processA
```

Compilation to OCaml: replication

- When a process is under replication, it is compiled into an ordinary function:

  fun ⟦*args*⟧ -> ⟦*body*⟧

- When a process is not under replication, it is compiled into a function that can be called only once:

```
let token = ref true in
fun ⟦args⟧ ->
  if (!token) then
    begin
      token := false;
      ⟦body⟧
    end
  else raise Bad_call
```

Compilation to OCaml: terms and body (1)

CryptoVerif	OCaml
M	$[\![M]\!]$
x	$[\![x]\!]$
$f(M_1, \ldots, M_n)$	$[\![f]\!]\ [\![M_1]\!]\ \ldots\ [\![M_n]\!]$
P	$[\![P]\!]$
new $x : T; P$	`let` $[\![x]\!]$ `=` $[\![rand_T]\!]$`()` `in` $[\![P]\!]$
let $x = M$ in P	`let` $[\![x]\!]$ `=` $[\![M]\!]$ `in` $[\![P]\!]$
if M then P else P'	`if` $[\![M]\!]$ `then` $[\![P]\!]$ `else` $[\![P']\!]$
end	`raise Match_fail`
out$(c, M); Q$	`(`$[\![Q]\!]$`,` $[\![M]\!]$`)`

When a variable needs to be written to a file, it is written just after its definition.

Compilation to OCaml: terms and body (2)

insert $Tbl(M_1, \ldots, M_n); P$
compiled into
`insert_in_table` $[\![Tbl]\!]$ `[`$[\![serial_{T_1}]\!]$ $[\![M_1]\!]$`;...;`$[\![serial_{T_n}]\!]$ $[\![M_n]\!]$`];` $[\![P]\!]$

get $Tbl(x_1 : T_1, \ldots, x_n : T_n)$ suchthat M in P else P'
compiled into

```
let l = get_from_table [[Tbl]]
  (function [[x1]]'; ...; [[xn]]'] ->
    let [[x1]] = exc_bad_file [[Tbl]] ([[deserial_T1]] [[x1]]') in ...
    let [[xn]] = exc_bad_file [[Tbl]] ([[deserial_Tn]] [[xn]]') in
    if [[M]] then ([[x1]], ..., [[xn]]) else raise Match_fail
  | _ -> raise (Bad_file [[Tbl]]))
in
if l = [] then [[P']] else
let ([[x1]], ..., [[xn]]) = rand_list l in [[P]]
```

Assumptions

- Assumptions on the network code:
  - No unsafe OCaml functions (such as `Obj.magic`).
  - No mutation of values received from or passed to generated functions.
  - No fork after obtaining and before calling a process that can be called only once.

Assumptions

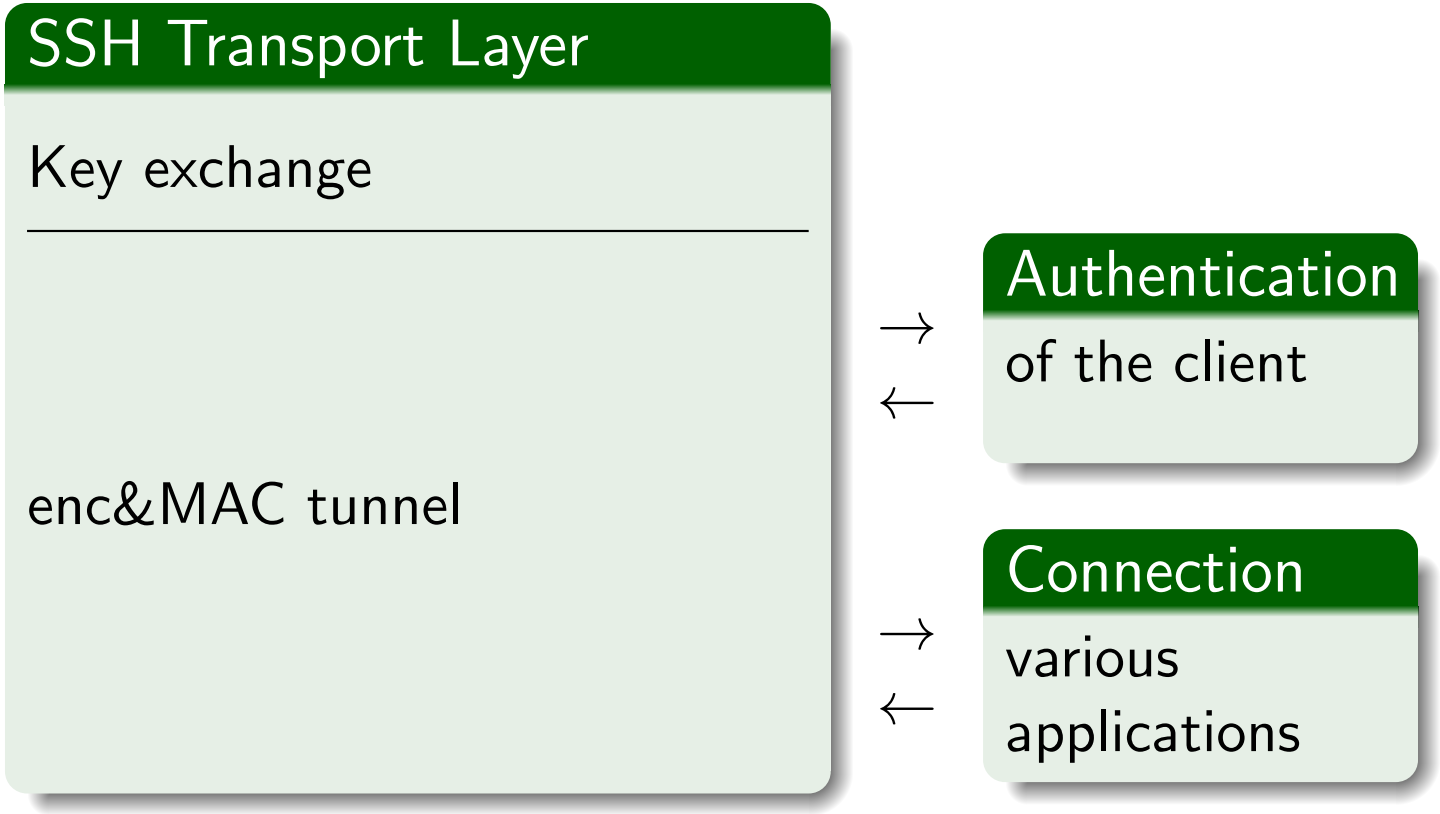
- Assumptions on the network code:
  - No unsafe OCaml functions (such as `Obj.magic`).
  - No mutation of values received from or passed to generated functions.
  - No fork after obtaining and before calling a process that can be called only once.
- Assumptions on program execution:
  - Programs are executed in the order specified in the CryptoVerif process.
  - Several programs that insert data in the same table are not run concurrently.

Assumptions

- Assumptions on the network code:
  - No unsafe OCaml functions (such as `Obj.magic`).
  - No mutation of values received from or passed to generated functions.
  - No fork after obtaining and before calling a process that can be called only once.
- Assumptions on program execution:
  - Programs are executed in the order specified in the CryptoVerif process.
  - Several programs that insert data in the same table are not run concurrently.
- Other:
  - Types that represent CryptoVerif data are not recursive.
  - The files used by generated code are not read/written by other code.

Application: SSH

- Secure SHell: an important protocol



SSH v. 2.0

SSH Transport Layer Protocol: key exchange

Client C		Server S
	$\xrightarrow{id_C = \text{SSH-2.0-version}_C}$	
	$\xleftarrow{id_S = \text{SSH-2.0-version}_S}$	
	$\xrightarrow{\text{KEXINIT}, cookie_C, algos_C}$	
	$\xleftarrow{\text{KEXINIT}, cookie_S, algos_S}$	
$x \xleftarrow{R} [2, q-1], e = g^x$	$\xrightarrow{\text{KEYDH_INIT}, e}$	$y \xleftarrow{R} [1, q-1], f = g^y$
$K = f^x$	$\xleftarrow{\text{KEYDH_REPLY}, pk_S, f, sign(H, sk_S)}$	$K = e^y$
pk_S, $sign(H, sk_S)$ ok?	$\xrightarrow{\text{NEWKEYS}}$	
	$\xleftarrow{\text{NEWKEYS}}$	

$algos =$ diffie-hellman-group14-sha1, ssh-rsa, aes128-cbc, hmac-sha1

$H = \text{SHA1}(id_C, id_S, cookie_C, algos_C, cookie_S, algos_S, pk_S, e, f, K)$

SSH Transport Layer Protocol: packet protocol

$$
\begin{aligned}
sessionid &= H \\
IV_C &= \mathrm{SHA1}(K, H, \text{“}A'', sessionid) \\
IV_S &= \mathrm{SHA1}(K, H, \text{“}B'', sessionid) \\
K_{enc,C} &= \mathrm{SHA1}(K, H, \text{“}C'', sessionid) \\
K_{enc,S} &= \mathrm{SHA1}(K, H, \text{“}D'', sessionid) \\
K_{MAC,C} &= \mathrm{SHA1}(K, H, \text{“}E'', sessionid) \\
K_{MAC,S} &= \mathrm{SHA1}(K, H, \text{“}F'', sessionid)
\end{aligned}
$$

$$packet = packet_length || padding_length || payload || padding$$

$$
\begin{array}{lcl}
\text{Client } C & \xrightarrow{enc(K_{enc,C}, packet, IV_C), MAC(K_{MAC,C}, sequence_number_C || packet)} & \text{Server } S \\
 & \xleftarrow{enc(K_{enc,S}, packet, IV_S), MAC(K_{MAC,S}, sequence_number_S || packet)} &
\end{array}
$$

CryptoVerif proof

- Modeled the SSH Transport Layer Protocol in CryptoVerif.
- Proved the authentication of the server to the client
  - Automatic by CryptoVerif
- The authentication of the client to the server requires the authentication protocol.
- Secrecy of the key requires extensions of CryptoVerif.
- Secrecy of messages sent over the tunnel cannot be proved:
  - Length of the packet leaked,
  - CBC mode with chained IVs.

Generated implementation

- Manually written cryptographic primitives.
  - based on CryptoKit.
- Manually written network code:
  - Key generators,
  - Client,
  - Server.

  They call the code generated from the CryptoVerif model.
- Format respected at the bit level.
  - Interact with other SSH implementations (OpenSSH).
- Some features omitted:
  - Key re-exchange
  - IGNORE, DISCONNECT messages

Demo

- ssh.ocv
- Prove by CryptoVerif
- Compile: key generation, client, server
- Run

Conclusion

- CryptoVerif specifications
  - proved secure in the computational model by CryptoVerif,
  - translated into OCaml implementations.
- Our approach favors the methodology:
  1. Write a formal specification;
  2. Prove it;
  3. Then, build an implementation.
- We proved the soundness of the compiler.
  - specification secure $\Rightarrow$ implementation secure
- Future work: extend the specification language, with loops, mutable variables, ...
  - extensions of CryptoVerif and of the compiler