From CryptoVerif Specifications to Computationally Secure Implementations of Protocols

Bruno Blanchet and David Cadé

INRIA, École Normale Supérieure, CNRS, Paris

April 2012

Protocol verification

	Symbolic	Computational
Specifications	FDR, AVISPA, ProVerif, . . .	CryptoVerif, CertiCrypt, . . .
Implementations	FS2PV, F7, Spi2Java, Andy's talk, . . .	FS2CV, Computational F7, Andy's talk, , . . .

Protocol verification

	Symbolic	Computational
Specifications	FDR, AVISPA, ProVerif, . . .	CryptoVerif, CertiCrypt, . . .
Implementations	FS2PV, F7, Spi2Java, Andy's talk, . . .	FS2CV, Computational F7, Andy's talk, our work, . . .

Our approach

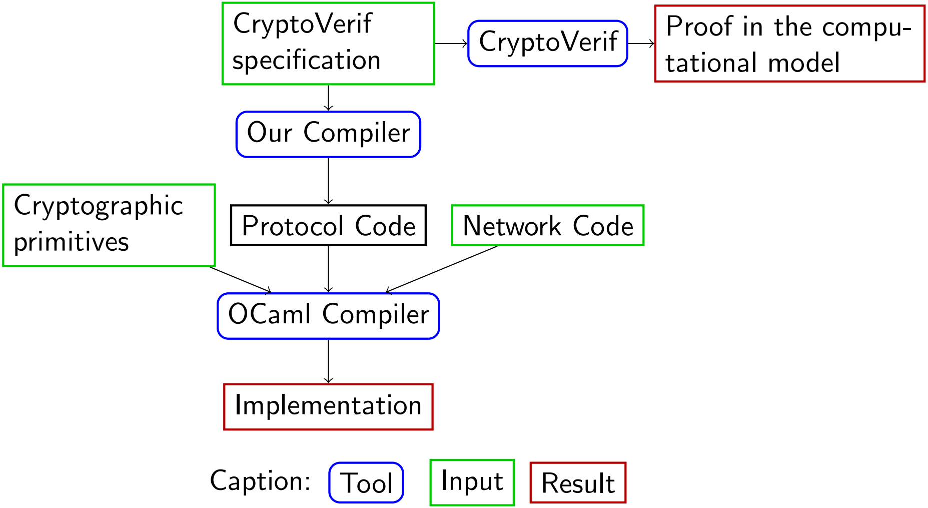
Generate protocol implementations from specifications.

- 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.

CryptoVerif

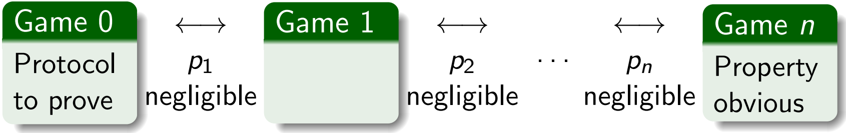
CryptoVerif is an automatic prover:

- in the computational model.
- proves secrecy and correspondence (authentication) properties.
- provides a generic method for specifying properties of cryptographic primitives.
- works for *N* sessions (polynomial in the security parameter), with an active adversary.
- gives a bound on the probability of an attack (exact security).
- possibility to guide the prover (manual mode).

Proofs by sequences of games

CryptoVerif produces proofs by sequences of games, like those of cryptographers [Shoup, Bellare&Rogaway]:

- The first game is the real protocol.
- One goes from one game to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. The difference of probability between consecutive games is negligible.
- The last game is "ideal": the security property is obvious from the form of the game.
  (The advantage of the adversary is 0 for this game.)



Game 0

Protocol to prove

$\longleftrightarrow$ p_1 negligible

Game 1

$\longleftrightarrow$ p_2 negligible

$\cdots$

$\longleftrightarrow$ p_n negligible

Game n

Property obvious

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

$Q ::=$	oracle definitions
0	nil
$Q \mid Q'$	parallel composition
$\text{foreach } i \leq n \text{ do } Q$	replication n times
$O[\widetilde{i}](x_1 : T_1, \dots, x_k : T_k) := P$	oracle definition
$P ::=$	oracle body
$\text{return}(M_1, \dots, M_k); Q$	return
end	end
$x \overset{R}{\leftarrow} T; P$	random number
$x : T \leftarrow M; P$	assignment
$\text{if } M \text{ then } P \text{ else } P'$	conditional
$\text{insert } Tbl(M_1, \dots, M_k); P$	insert in table
$\text{get } Tbl(x_1 : T_1, \dots, x_k : T_k) \text{ suchthat } M \text{ in } P \text{ else } P'$	get from table

Example

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

$$\begin{aligned}&\textsf{process}\ Ostart() := rKab \overset{R}{\leftarrow} keyseed; Kab \leftarrow \mathrm{kgen}(rKab); \textsf{return}();\\ &\qquad (\textsf{foreach}\ i1 \leq N\ \textsf{do}\ processA \mid\\ &\qquad\ \ \textsf{foreach}\ i2 \leq N\ \textsf{do}\ processB)\end{aligned}$$

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

Example

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

$$\text{let } processA = OA() := r \overset{R}{\leftarrow} nonce; s \overset{R}{\leftarrow} seed;$$
$$\text{return}(\mathrm{enc}(\mathrm{nonceToBitstring}(r), Kab, s)).$$

$$\text{let } processB = OB(m : bitstring) :=$$
$$\text{let } \mathrm{injbot}(\mathrm{nonceToBitstring}(r' : nonce)) = \mathrm{dec}(m, Kab) \text{ in}$$
$$\text{return}().$$

- *OA* sends the encryption of r under *Kab* (probabilistic encryption)
- *OB* decrypts the received message

Example — summary

$$\text{let } \mathit{processA} = \mathit{OA}() := r \overset{R}{\leftarrow} \mathit{nonce}; s \overset{R}{\leftarrow} \mathit{seed};$$
$$\text{return}(\text{enc}(\text{nonceToBitstring}(r), \mathit{Kab}, s)).$$

$$\text{let } \mathit{processB} = \mathit{OB}(m : \mathit{bitstring}) :=$$
$$\text{let injbot}(\text{nonceToBitstring}(r' : \mathit{nonce})) = \text{dec}(m, \mathit{Kab}) \text{ in}$$
$$\text{return}().$$

$$\text{process } \mathit{Ostart}() := \mathit{rKab} \overset{R}{\leftarrow} \mathit{keyseed}; \mathit{Kab} \leftarrow \text{kgen}(\mathit{rKab}); \text{return}();$$
$$(\text{foreach } i1 \leq N \text{ do } \mathit{processA} \mid$$
$$\text{foreach } i2 \leq N \text{ do } \mathit{processB})$$

Annotations: Separation in multiple programs

$$\begin{array}{l} \text{let } \mathit{processA} = \mathit{pA}\{\mathit{OA}() := r \xleftarrow{R} \mathit{nonce}; s \xleftarrow{R} \mathit{seed}; \\ \qquad \text{return}(\text{enc}(\text{nonceToBitstring}(r), \mathit{Kab}, s))\}. \end{array}$$

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

$$\begin{array}{l} \text{process } \mathit{keygen}\ [\mathit{Kab} > \mathit{fileKab}]\ \{\mathit{Ostart}() := \\ \qquad \mathit{rKab} \xleftarrow{R} \mathit{keyseed}; \mathit{Kab} : \mathit{key} \leftarrow \text{kgen}(\mathit{rKab}); \text{return}()\}; \\ \quad (\text{foreach } i1 \leq N \text{ do } \mathit{processA} \mid \\ \quad\ \text{foreach } i2 \leq N \text{ do } \mathit{processB}) \end{array}$$

Annotations: External data files

$$\begin{aligned}
&\mathsf{let}\ \mathit{processA} = \mathit{pA}\{\mathit{OA}() := r \overset{R}{\leftarrow} \mathit{nonce};\ s \overset{R}{\leftarrow} \mathit{seed};\\
&\qquad \mathsf{return}(\mathsf{enc}(\mathsf{nonceToBitstring}(r), \mathit{Kab}, s))\}.
\end{aligned}$$

$$\begin{aligned}
&\mathsf{let}\ \mathit{processB} = \mathit{pB}\{\mathit{OB}(m : \mathit{bitstring}) :=\\
&\qquad \mathsf{let}\ \mathsf{injbot}(\mathsf{nonceToBitstring}(r' : \mathit{nonce})) = \mathsf{dec}(m, \mathit{Kab})\ \mathsf{in}\\
&\qquad \mathsf{return}()\}.
\end{aligned}$$

$$\begin{aligned}
&\mathsf{process}\ \mathit{keygen}\ [\mathit{Kab} > \mathit{fileKab}]\ \{\mathit{Ostart}() :=\\
&\qquad\quad \mathit{rKab} \overset{R}{\leftarrow} \mathit{keyseed};\ \mathit{Kab} : \mathit{key} \leftarrow \mathsf{kgen}(\mathit{rKab});\ \mathsf{return}()\};\\
&\qquad (\mathsf{foreach}\ i1 \leq N\ \mathsf{do}\ \mathit{processA}\ |\\
&\qquad\ \ \mathsf{foreach}\ i2 \leq N\ \mathsf{do}\ \mathit{processB})
\end{aligned}$$

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 kgen = "*sym_kgen*".
  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.
- foreach $i \leq n$ do $\ldots$ insert $keytbl(h, k)$

  becomes

  foreach $i \leq n$ do $\ldots keytbl_1[i] \leftarrow h; keytbl_2[i] \leftarrow k$
- get $keytbl(h', k')$ suchthat $h' = h$ in P else P'

  becomes

  find $u \leq n$ suchthat defined$(keytbl_1[u], keytbl_2[u]) \wedge keytbl_1[u] = h$
  then $h' \leftarrow keytbl_1[u]; k' \leftarrow keytbl_2[u]; P$ 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.
- foreach $i \leq n$ do $\dots$ insert $keytbl(h, k)$

  becomes

  foreach $i \leq n$ do $\dots keytbl_1[i] \leftarrow h; keytbl_2[i] \leftarrow k$
- get $keytbl(h', k')$ suchthat $h' = h$ in P else P'

  becomes

  find $u \leq n$ suchthat defined$(keytbl_1[u], keytbl_2[u]) \wedge keytbl_1[u] = h$
  then $h' \leftarrow keytbl_1[u]; k' \leftarrow keytbl_2[u]; P$ 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.
- foreach $i \leq n$ do $\ldots$ insert $keytbl(h, k)$

  becomes

  foreach $i \leq n$ do $\ldots keytbl_1[i] \leftarrow h; keytbl_2[i] \leftarrow k$
- get $keytbl(h', k')$ suchthat $h' = h$ in P else P'

  becomes

  find $u \leq n$ suchthat defined$(keytbl_1[u], keytbl_2[u]) \wedge keytbl_1[u] = h$ then $h' \leftarrow keytbl_1[u]; k' \leftarrow keytbl_2[u]; P$ 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 oracle.

- A function $\texttt{init} : \texttt{unit} \to \tau$ returns the tuple of functions representing the oracles available at the beginning of the program.
  - `init` may also read variables from files when needed.
- Each oracle O is represented by a function that
  - takes as argument the arguments of O
  - and returns
    - the tuple of functions representing oracles that follow O,
    - the result of O.

Compilation to OCaml: example

$$\text{let } processA = pA\{OA() := r \xleftarrow{R} nonce; s \xleftarrow{R} seed; \\ \text{return}(\text{enc}(\text{nonceToBitstring}(r), Kab, s))\}.$$

The generated module PA has the following interface :

```
open Base
open Crypto

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

Compilation to OCaml: replication

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

  `fun` $[\![args]\!]$ `->` $[\![body]\!]$

- When an oracle 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	$\llbracket M \rrbracket$
x	$\llbracket x \rrbracket$
$f(M_1, \dots, M_n)$	$\llbracket f \rrbracket\ \llbracket M_1 \rrbracket\ \dots\ \llbracket M_n \rrbracket$
P	$\llbracket P \rrbracket$
$x \xleftarrow{R} T; P$	$\texttt{let}\ \llbracket x \rrbracket\ \texttt{=}\ \llbracket rand_T \rrbracket\texttt{()}\ \texttt{in}\ \llbracket P \rrbracket$
$x \leftarrow M; P$	$\texttt{let}\ \llbracket x \rrbracket\ \texttt{=}\ \llbracket M \rrbracket\ \texttt{in}\ \llbracket P \rrbracket$
if M then P else P'	$\texttt{if}\ \llbracket M \rrbracket\ \texttt{then}\ \llbracket P \rrbracket\ \texttt{else}\ \llbracket P' \rrbracket$
end	$\texttt{raise Match\_fail}$
return$(M_1, \dots, M_n); Q$	$\texttt{(}\llbracket Q \rrbracket\texttt{, (}\llbracket M_1 \rrbracket\texttt{, ..., }\llbracket M_n \rrbracket\texttt{))}$

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 $Tbl(M_1, \dots, M_n); P$

compiled into

$\texttt{insert\_in\_table}\ [\![Tbl]\!]\ \texttt{[}[\![serial_{T_1}]\!]\ [\![M_1]\!]\texttt{;}\dots\texttt{;}[\![serial_{T_n}]\!]\ [\![M_n]\!]\texttt{]}\texttt{;}\ [\![P]\!]$

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

compiled into

$$\begin{array}{l}
\texttt{let l = get\_from\_table}\ [\![Tbl]\!] \\
\quad \texttt{(function [}[\![x_1]\!]\texttt{'; ...; }[\![x_n]\!]\texttt{'] ->} \\
\qquad \texttt{let }[\![x_1]\!]\texttt{ = exc\_bad\_file }[\![Tbl]\!]\texttt{ (}[\![deserial_{T_1}]\!]\ [\![x_1]\!]\texttt{') in }\dots \\
\qquad \texttt{let }[\![x_n]\!]\texttt{ = exc\_bad\_file }[\![Tbl]\!]\texttt{ (}[\![deserial_{T_n}]\!]\ [\![x_n]\!]\texttt{') in} \\
\qquad \texttt{if }[\![M]\!]\texttt{ then (}[\![x_1]\!]\texttt{, ..., }[\![x_n]\!]\texttt{) else raise Match\_fail} \\
\quad \texttt{| \_ -> raise (Bad\_file }[\![Tbl]\!]\texttt{))} \\
\texttt{in} \\
\texttt{if l = [] then }[\![P']\!]\texttt{ else} \\
\texttt{let (}[\![x_1]\!]\texttt{, ..., }[\![x_n]\!]\texttt{) = rand\_list l in }[\![P]\!]
\end{array}$$

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 an oracle that can be called only once.

Assumptions

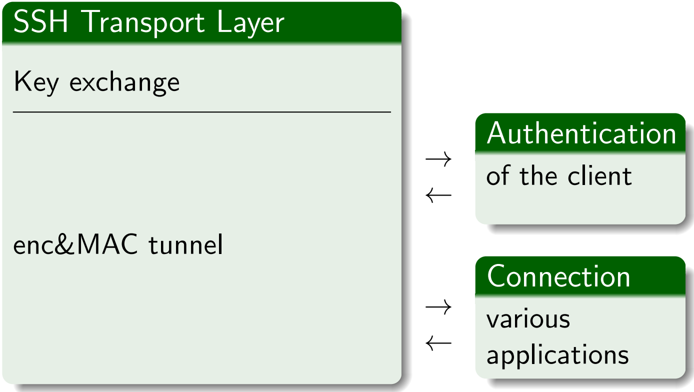
- Assumptions on the network code:
  - No unsafe OCaml functions (such as `Obj.magic`).
  - No mutation of values received from or passed to generated functions.
  - No fork after obtaining and before calling an oracle that can be called only once.
- Assumptions on program execution:
  - Programs are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

Key exchange

---

enc&MAC tunnel

→
←

Authentication

of the client

→
←

Connection

various applications

SSH v. 2.0

SSH Transport Layer Protocol: key exchange

Client C Server S

$$\xrightarrow{id_C=\text{SSH-2.0-version}_C}$$

$$\xleftarrow{id_S=\text{SSH-2.0-version}_S}$$

$$\xrightarrow{\text{KEXINIT}, cookie_C, algos_C}$$

$$\xleftarrow{\text{KEXINIT}, cookie_S, algos_S}$$

$$x \overset{R}{\leftarrow} [2, q-1], e = g^x \quad \xrightarrow{\text{KEYDH\_INIT}, e} \quad y \overset{R}{\leftarrow} [1, q-1], f = g^y$$

$$K = f^x \quad \xleftarrow{\text{KEYDH\_REPLY}, pk_S, f, sign(H, sk_S)} \quad K = e^y$$

$$pk_S,\ sign(H, sk_S)\ \text{ok?} \quad \xrightarrow{\text{NEWKEYS}}$$

$$\xleftarrow{\text{NEWKEYS}}$$

$algos =$ 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

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

$$packet = packet\_length || padding\_length || payload || padding$$

$$\text{Client } C \quad \begin{array}{c} \xrightarrow{enc(K_{enc,C}, packet, IV_C), MAC(K_{MAC,C}, sequence\_number_C || packet)} \\ \xleftarrow{enc(K_{enc,S}, packet, IV_S), MAC(K_{MAC,S}, sequence\_number_S || packet)} \end{array} \quad \text{Server } S$$

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.
- In progress: prove the soundness of the compiler.
  - specification secure ⇒ implementation secure
- Future work: extend the specification language, with loops, mutable variables, ...
  - extensions of CryptoVerif and of the compiler