From CryptoVerif Specifications to Computationally Secure Implementations of Protocols

Bruno Blanchet and David Cadé

INRIA, École Normale Supérieure, CNRS, Paris

April 2012
## Protocol verification

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Symbolic</th>
<th>Computational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifications</td>
<td>FDR, AVISPA, ProVerif, …</td>
<td>CryptoVerif, CertiCrypt, …</td>
</tr>
<tr>
<td>Implementations</td>
<td>FS2PV, F7, Spi2Java, Andy’s talk, …</td>
<td>FS2CV, Computational F7, Andy’s talk, …</td>
</tr>
</tbody>
</table>
## Protocol verification

<table>
<thead>
<tr>
<th></th>
<th>Symbolic</th>
<th>Computational</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specifications</strong></td>
<td>FDR, AVISPA, ProVerif, ...</td>
<td>CryptoVerif, CertiCrypt, ...</td>
</tr>
<tr>
<td><strong>Implementations</strong></td>
<td>FS2PV, F7, Spi2Java, Andy’s talk, ...</td>
<td>FS2CV, Computational F7, Andy’s talk, our work, ...</td>
</tr>
</tbody>
</table>
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

CryptoVerif specification → CryptoVerif

Our Compiler

Proof in the computational model

Cryptographic primitives

Protocol Code

Network Code

OCaml Compiler

Implementation

Caption: Tool Input Result
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 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.)
CryptoVerif represents protocols and games in a \textit{process calculus}.

\[
M, N ::= \quad \text{terms} \\
\quad x \quad \text{variable} \\
\quad f(M_1, \ldots, M_m) \quad \text{function application}
\]

Function symbols $f$ correspond to functions computable by polynomial-time deterministic Turing machines.
The CryptoVerif specification language: processes

\[Q ::=\]
0 \quad \text{nil}
\quad Q | Q' \quad \text{parallel composition}
\quad \text{foreach } i \leq n \text{ do } Q \quad \text{replication } n \text{ times}
\quad O[i](x_1 : T_1, \ldots, x_k : T_k) := P \quad \text{oracle definition}

\[P ::=\]
\quad \text{return}(M_1, \ldots, M_k); Q \quad \text{return}
\quad \text{end} \quad \text{end}
\quad x \xleftarrow{R} T; P \quad \text{random number}
\quad x : T \xleftarrow{} M; P \quad \text{assignment}
\quad \text{if } M \text{ then } P \text{ else } P' \quad \text{conditional}
\quad \text{insert } Tbl(M_1, \ldots, M_k); P \quad \text{insert in table}
\quad \text{get } Tbl(x_1 : T_1, \ldots, x_k : T_k) \text{ such that } M \text{ in } P \text{ else } P' \quad \text{get from table}
Example

\[ A \rightarrow B : \text{enc}(r, Kab) \]

\begin{verbatim}
process Ostart() := \text{rKab} \leftarrow \text{keyseed}; \text{Kab} \leftarrow \text{kgen}(r\text{Kab})\; \text{return}();
    (\text{foreach } i_1 \leq N \text{ do } \text{processA} | \\
     \text{foreach } i_2 \leq N \text{ do } \text{processB})
\end{verbatim}

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

Example

\[ \text{A} \quad \rightarrow \quad \text{B} : \text{enc}(r, Kab) \]

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

let \( \text{processB} = \text{OB}(m : \text{bitstring}) := \)
\[ \text{return()}. \]

- \( \text{OA} \) sends the encryption of \( r \) under \( Kab \) (probabilistic encryption)
- \( \text{OB} \) decrypts the received message
Example — summary

\begin{verbatim}
let processA = OA() :=
    r \leftarrow nonce; s \leftarrow seed;
    return(enc(nonceToBitstring(r), Kab, s)).

let processB = OB(m : bitstring) :=
    let injbot(nonceToBitstring(r' : nonce)) = dec(m, Kab) in
    return().

process Ostart() :=
    rKab \leftarrow keyseed; Kab \leftarrow kgen(rKab); return();
    (foreach i1 \leq N do processA |
    foreach i2 \leq N do processB)
\end{verbatim}
Annotations: Separation in multiple programs

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

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

process keygen [Kab > fileKab] {Ostart() := 
  rKab \xleftarrow{R}\ keyseed; Kab : key \xleftarrow{}\ kgen(rKab); return()};

(foreach i1 \leq N do processA | 
  foreach i2 \leq N do processB)
Annotations: External data files

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

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

process keygen [Kab > fileKab] \{Ostart() :=
    rKab \leftarrow \text{keyseed}; Kab : key \leftarrow \text{kgen}(rKab); return()\};
    (\text{foreach } i1 \leq N \text{ do processA} |
    \text{foreach } i2 \leq N \text{ do processB})
Annotations: types and functions

- OCaml type representing a CryptoVerif type:
  
  \[
  \text{implementation type } \text{keyseed} = 128. \quad \text{(bitstring of 128 bits)} \\
  \text{implementation type } \text{host} = "\text{string}" \quad [\text{serial} = "\text{id}", "\text{id}"].
  \]

- OCaml function representing a function in the protocol specification:

  \[
  \text{implementation fun kgen} = "\text{sym\_kgen}" . \\
  \text{implementation fun injbot} = "\text{injbot}" \quad [\text{inverse} = "\text{injbot\_inv}""].
  \]

  - In the CryptoVerif specification, there are assumptions about these functions.
    - Functional assumptions: \(\text{dec(\text{enc}(m, k, s), k) = injbot(m)}\).
    - 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')` such that `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"`. 
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.
- \( \text{foreach } i \leq n \text{ do } \ldots \text{ insert } \text{keytbl}(h, k) \)
  becomes
- \( \text{foreach } i \leq n \text{ do } \ldots \text{keytbl}_1[i] \leftarrow h; \text{keytbl}_2[i] \leftarrow k \)

- \( \text{get } \text{keytbl}(h', k') \text{ suchthat } h' = h \text{ in } P \text{ else } P' \)
  becomes
- \( \text{find } u \leq n \text{ suchthat } \text{defined}(\text{keytbl}_1[u], \text{keytbl}_2[u]) \land \text{keytbl}_1[u] = h \)
  then \( h' \leftarrow \text{keytbl}_1[u]; k' \leftarrow \text{keytbl}_2[u]; P \text{ else } P' \)
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.
- `foreach i ≤ n do ... insert keytbl(h, k)`
  becomes
  `foreach i ≤ n do ... keytbl1[i] ← h; keytbl2[i] ← k`
- `get keytbl(h′, k′) suchthat h′ = h in P else P′`
  becomes
  `find u ≤ n suchthat defined(keytbl1[u], keytbl2[u]) ∧ keytbl1[u] = h`
  `then h′ ← keytbl1[u]; k′ ← keytbl2[u]; P else P′`
- Generalized to several insertions by looking up in the variables defined at each insertion.
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 ≤ n do ... insert keytbl(h, k)` becomes
  `foreach i ≤ n do ... keytbl1[i] ← h; keytbl2[i] ← k`
- `get keytbl(h', k') suchthat h' = h in P else P'` becomes
  `find u ≤ n suchthat defined(keytbl1[u], keytbl2[u]) ∧ keytbl1[u] = h` then
  `h' ← keytbl1[u]; k' ← keytbl2[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 \( \text{init} : \text{unit} \rightarrow \tau \) returns the tuple of functions representing the oracles available at the beginning of the program.
  - \( \text{init} \) 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

let processA = pA\{ OA() := r \xleftarrow{R} nonce; s \xleftarrow{R} seed; \
    return(enc(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:
  ```ocaml```
  ```
  fun [args] -> [body]
  ```
- When an oracle is **not under replication**, it is compiled into a function that **can be called only once**:
  ```ocaml```
  ```
  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)

<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>$[M]$</td>
</tr>
<tr>
<td>$x$</td>
<td>$[x]$</td>
</tr>
<tr>
<td>$f(M_1, \ldots, M_n)$</td>
<td>$[f] [M_1] \ldots [M_n]$</td>
</tr>
<tr>
<td>$P$</td>
<td>$[P]$</td>
</tr>
</tbody>
</table>

When a variable needs to be written to a file, it is written just after its definition.
Compilation to OCaml: terms and body (2)

\[ \text{insert } Tbl(M_1, \ldots, M_n); P \]

compiled into

\[ \text{insert_in_table } [Tbl] [[\text{serial}_{T_1}] [M_1]; \ldots; [[\text{serial}_{T_n}] [M_n]]; [P] \]

\[ \text{get } Tbl(x_1 : T_1, \ldots, x_n : T_n) \text{ such that } M \text{ in } P \text{ else } P' \]

compiled into

\[
\begin{align*}
\text{let } l &= \text{get_from_table } [Tbl] \\
&\quad \text{(function } [[x_1]'; \ldots; [x_n]'] \rightarrow \\
&\quad \quad \text{let } [x_1] = \text{exc_bad_file } [Tbl] ([\text{deserial}_{T_1}] [x_1]') \text{ in } \ldots \\
&\quad \quad \text{let } [x_n] = \text{exc_bad_file } [Tbl] ([\text{deserial}_{T_n}] [x_n]') \text{ in } \\
&\quad \quad \text{if } [M] \text{ then } ([x_1], \ldots, [x_n]) \text{ else raise Match_fail} \\
&\quad \quad \text{in } \\
&\text{if } l = [] \text{ then } [P'] \text{ else } \\
&\text{let } ([x_1], \ldots, [x_n]) = \text{rand_list } l \text{ in } [P] \\
\end{align*}
\]
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.
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

### Key Exchange Protocol

<table>
<thead>
<tr>
<th>Client C</th>
<th>Server S</th>
</tr>
</thead>
<tbody>
<tr>
<td>id(_C) = SSH-2.0-version(_C)</td>
<td>id(_S) = SSH-2.0-version(_S)</td>
</tr>
<tr>
<td>KEXINIT, cookie(_C), algos(_C)</td>
<td>KEXINIT, cookie(_S), algos(_S)</td>
</tr>
<tr>
<td>x (\xleftarrow{$} [2, q - 1]), e = g^x</td>
<td>KEYDH_INIT, e</td>
</tr>
<tr>
<td>K = f^x</td>
<td>KEYDH_REPLY, pk(_S), f, sign(H, sk(_S))</td>
</tr>
<tr>
<td>pk(_S), sign(H, sk(_S)) ok?</td>
<td>K = e^y</td>
</tr>
</tbody>
</table>

### Key Generation

\[ K = f^x \quad y \xleftarrow{\$} [1, q - 1], f = g^y \]

### Final Exchange

\[ algos = \text{diffie-hellman-group14-sha1, ssh-rsa, aes128-cbc, hmac-sha1} \]

\[ 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

\[
\text{\textit{sessionid}} = H
\]

\[
IV_C = \text{SHA1}(K, H, "A", \text{sessionid})
\]

\[
IV_S = \text{SHA1}(K, H, "B", \text{sessionid})
\]

\[
K_{enc,C} = \text{SHA1}(K, H, "C", \text{sessionid})
\]

\[
K_{enc,S} = \text{SHA1}(K, H, "D", \text{sessionid})
\]

\[
K_{MAC,C} = \text{SHA1}(K, H, "E", \text{sessionid})
\]

\[
K_{MAC,S} = \text{SHA1}(K, H, "F", \text{sessionid})
\]

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

Client C \[
\begin{align*}
&\text{enc}(K_{enc,C}, \text{packet}, IV_C), \text{MAC}(K_{MAC,C}, \text{sequence_number}_C || \text{packet}) \\
\end{align*}
\]

\[
\begin{align*}
&\text{MAC}(K_{MAC,S}, \text{sequence_number}_S || \text{packet})
\end{align*}
\]

\[
\text{Server S}
\]

Bruno Blanchet and David Cadé (INRIA) CryptoVerif April 2012 26 / 30
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
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 $\Rightarrow$ implementation secure

Future work: extend the specification language, with loops, mutable variables, . . . .
- extensions of CryptoVerif and of the compiler