From CryptoVerif Specifications to Computationally Secure Implementations of Protocols

Bruno Blanchet and David Cadé

INRIA Paris-Rocquencourt

December 2014

Protocol verification

	Symbolic	Computational
Specifications	FDR, AVISPA, ProVerif, ...	CryptoVerif, CertiCrypt, EasyCrypt, ...
Implementations	FS2PV, F7, Spi2Java, C2ProVerif (Aizatulin et al, CCS'11), ...	FS2CV, Computational F7, F*, C2CryptoVerif (Aizatulin et al, CCS'12), ...

Protocol verification

	Symbolic	Computational
Specifications	FDR, AVISPA, ProVerif, ...	CryptoVerif, CertiCrypt, EasyCrypt, ...
Implementations	FS2PV, F7, Spi2Java, C2ProVerif (Aizatulin et al, CCS'11), ...	FS2CV, Computational F7, F*, C2CryptoVerif (Aizatulin et al, CCS'12), our work, ...

Our approach

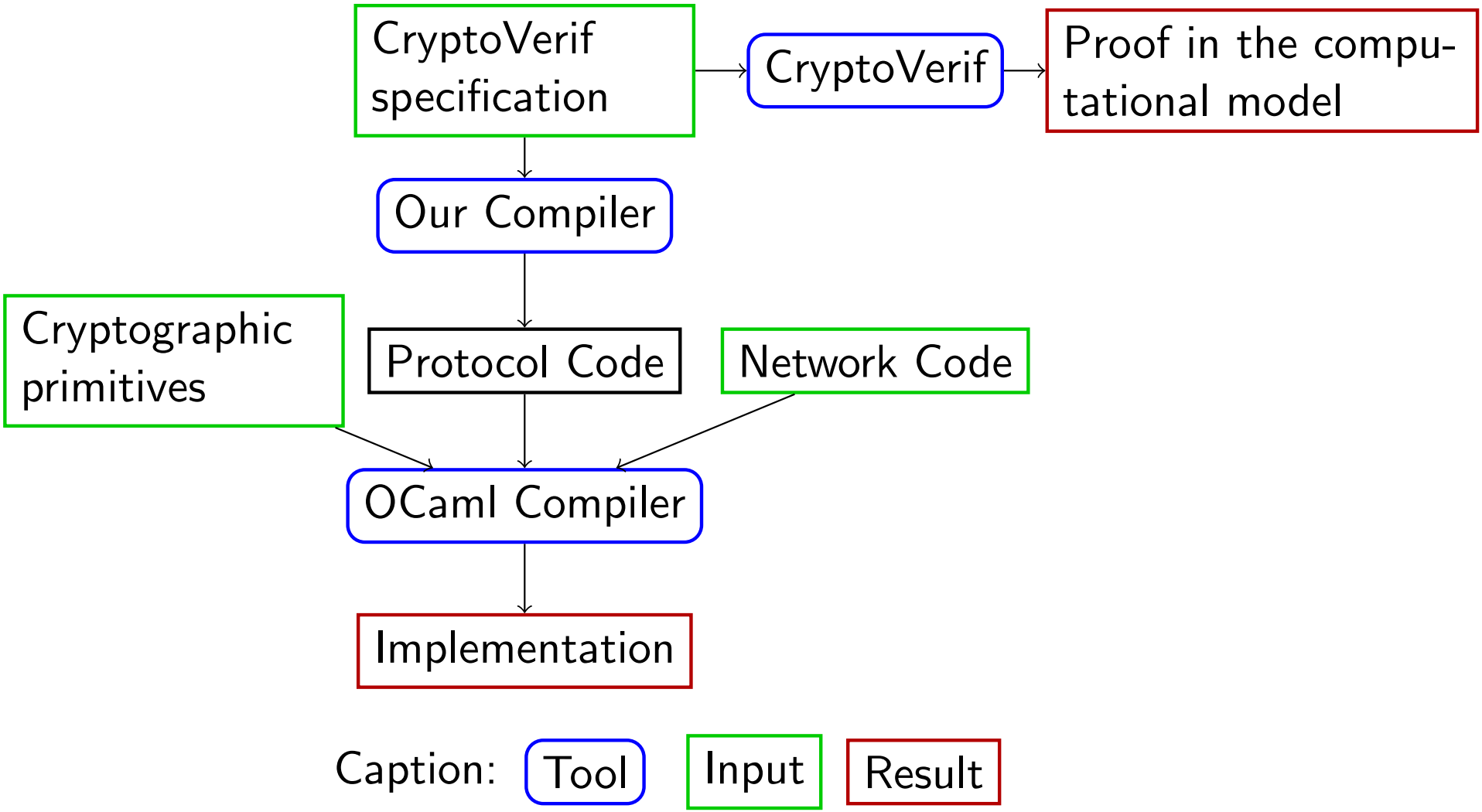
Generate protocol implementations from specifications.

- 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.

Example

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

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

- 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

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

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

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

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

Example — summary

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

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

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

Annotations: Separation in multiple programs

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

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

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

Annotations: External data files

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

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

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

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.

Compilation to OCaml

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

- A function $\texttt{init} : \texttt{unit} \rightarrow \tau$ 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.

Proof of security

The compiler is proved secure (POST'13)

Theorem (informal)

If a security property holds (up to probability p) on the CryptoVerif specification, then it also holds (up to the same probability) on the generated implementation.

This theorem requires some assumptions (next slide).

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

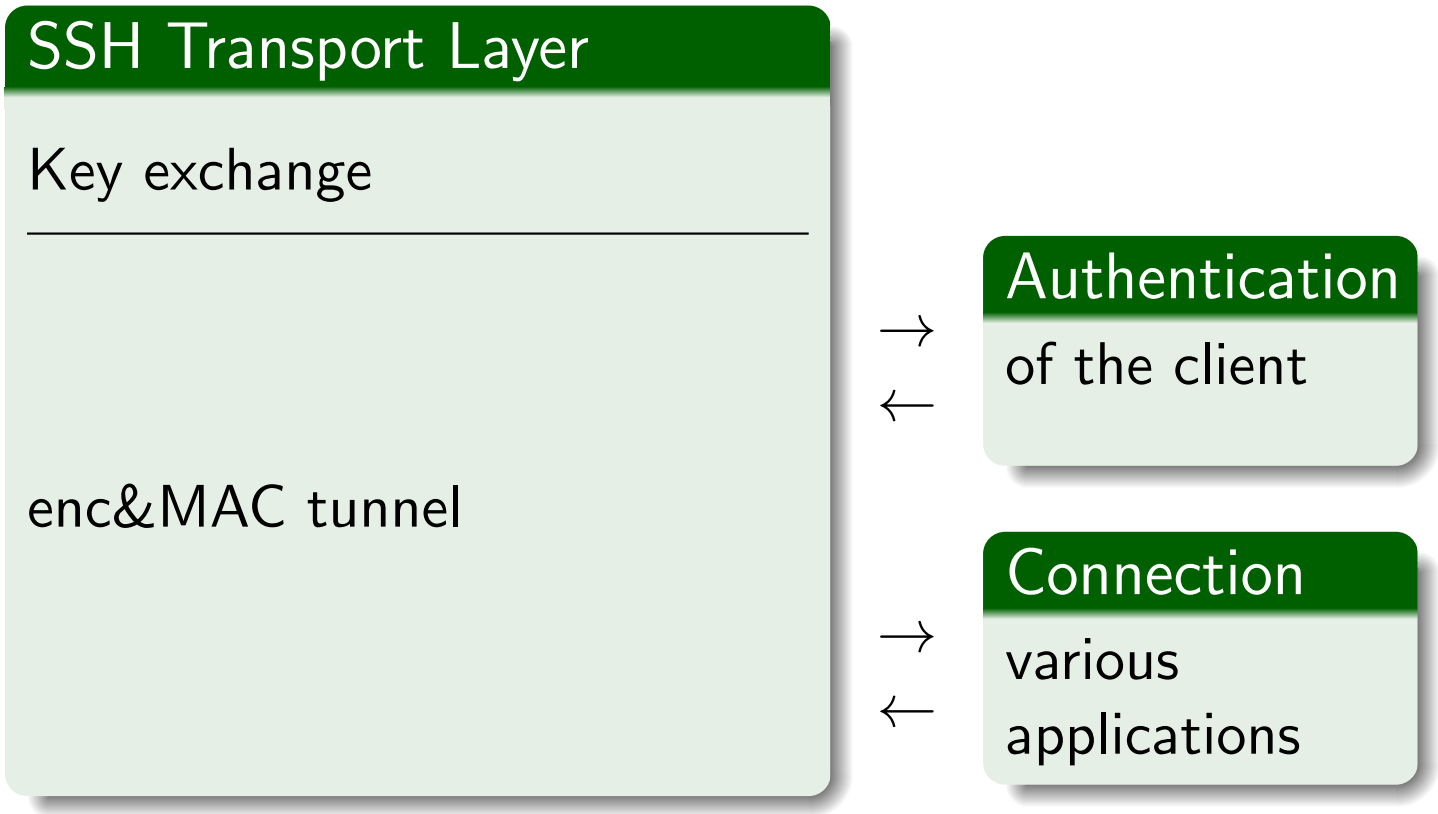
- Assumptions on the network code:
  - No unsafe OCaml functions (such as `Obj.magic`).
  - No mutation of values received from or passed to generated functions.
  - No fork after obtaining and before calling an oracle that can be called only once.
- Assumptions on program execution:
  - Programs are 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 v. 2.0

CryptoVerif proof

- Modeled the SSH Transport Layer Protocol in CryptoVerif.
- Proved
  - authentication of the server to the client (automatically)
  - secrecy of the session key (with user guidance)
- The authentication of the client to the server requires the authentication protocol.
- 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

Conclusion

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