CryptoVerif

Bruno Blanchet

INRIA Paris-Rocquencourt
bruno.blanchet@inria.fr

November 2014

CryptoVerif, http://cryptoverif.inria.fr/

CryptoVerif is an automatic prover that:

- generates proofs by sequences of games.
- proves secrecy and correspondence properties.
- provides a generic method for specifying properties of cryptographic primitives which handles MACs (message authentication codes), symmetric encryption, public-key encryption, signatures, hash functions, Diffie-Hellman key agreements, Xor, ...
- works for N sessions (polynomial in the security parameter), with an active adversary.
- gives a bound on the probability of an attack (exact security).
- has automatic and manual modes.

Proofs by sequences of games

Proofs in the computational model are typically proofs by sequences of games [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	$\longleftrightarrow$	Game 1	$\longleftrightarrow$		$\longleftrightarrow$	Game n
Protocol to prove	p_1 negligible		p_2 negligible	$\cdots$	p_n negligible	Property obvious

Input and output of the tool

1. Prepare the input file containing
   - the specification of the protocol to study (initial game),
   - the security assumptions on the cryptographic primitives,
   - the security properties to prove.
2. Run CryptoVerif
3. CryptoVerif outputs
   - the sequence of games that leads to the proof,
   - a succinct explanation of the transformations performed between games,
   - an upper bound of the probability of success of an attack.

Process calculus for games

Games are formalized in a process calculus:

- It is adapted from the pi calculus.
- The semantics is purely probabilistic (no non-determinism).
- The runtime of processes is bounded:
  - bounded number of copies of processes,
  - bounded length of messages on channels.

Indistinguishability

$$G \approx_p G'$$

means that an adversary has probability at most p of distinguishing G from G'.

Indistinguishability

$$G \approx_p G'$$

means that an adversary has probability at most p of distinguishing G from G'.

The probability p may depend on the runtime of the adversary, the number of calls to the oracles in the games, etc.

Proof technique

1. Start from the provided initial game
2. Transform it step by step using a collection of generic game transformations, such that each game is indistinguishable from the next one (up to probability p_i).
   We obtain a sequence of games $G_0 \approx_{p_1} G_1 \approx \ldots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1+\cdots+p_m} G_m$.
3. On the last game G_m, use a syntactic criterion to prove the desired security property (up to probability p).
   If some trace property holds up to probability p in G_m, then it holds up to probability $p + p_1 + \cdots + p_m$ in G_0.

Game transformations

We transform a game G_0 into an indistinguishable one using:

- indistinguishability properties $L \approx_p R$ given as axioms and that come from security assumptions on primitives (e.g. encryption is IND-CPA).
  These properties are used inside a context:

$$G_1 \approx_0 C[L] \approx_{p'} C[R] \approx_0 G_2$$

- syntactic transformations: simplification, expansion of assignments, ...

Proof strategy: advice

- One tries to execute each transformation given by the definition of a cryptographic primitive.
- When it fails, it tries to analyze why the transformation failed, and suggests syntactic transformations that could make it work.
- One tries to execute these syntactic transformations.
  (If they fail, they may also suggest other syntactic transformations, which are then executed.)
- We retry the cryptographic transformation, and so on.

Proof of security properties: one-session secrecy

One-session secrecy: the adversary cannot distinguish any of the secrets from a random number with one test query.

Proof of security properties: one-session secrecy

One-session secrecy: the adversary cannot distinguish any of the secrets from a random number with one test query.

Criterion for proving one-session secrecy of x:
x is defined by **new** $x[i] : T$ and there is a set of variables S such that only variables in S depend on x.
The output messages and the control-flow do not depend on x.

Proving protocol implementations (joint work with David Cadé)

- The previous approach proves protocol specifications.
- However, one does not run the specification, one runs an implementation.
- $\Rightarrow$ We need to prove protocol implementations secure!

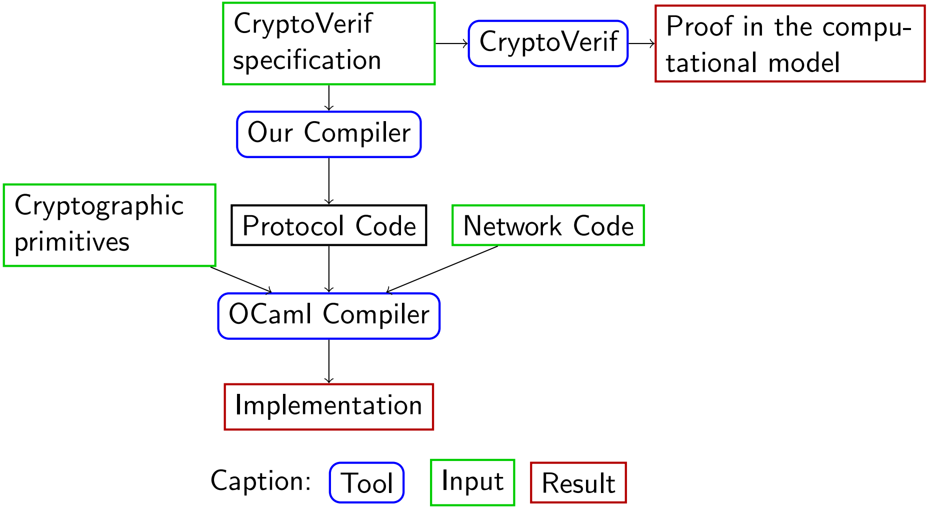
Our approach

Generate protocol implementations from specifications.

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

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.

Annotations

The specification is enriched with annotations that give implementation details:

- Separation in multiple programs, e.g. key generation, client, server.
- External data files, to store long-term keys and tables of keys.
- Correspondence between CryptoVerif and OCaml types and functions.

Compiler

- Our compiler translates annotated CryptoVerif specifications into an OCaml implementation.
- The compiler is proved secure

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 Transport Layer

Key exchange

---

enc&MAC tunnel

$\rightarrow$
$\leftarrow$

Authentication

of the client

$\rightarrow$
$\leftarrow$

Connection

various applications

SSH v. 2.0

SSH Transport Layer Protocol: key exchange

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

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

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

$$\text{Client } C \xrightarrow{enc(K_{enc,C}, packet, IV_C), MAC(K_{MAC,C}, sequence\_number_C || packet)} \text{Server } S$$

$$\xleftarrow{enc(K_{enc,S}, packet, IV_S), MAC(K_{MAC,S}, sequence\_number_S || packet)}$$

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 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

More details on Friday

Focusing on how to prove protocols using CryptoVerif.

- Morning: course. Details on how CryptoVerif works, with demos and examples
- Afternoon: tutorial.