From CryptoVerif Specifications to Computationally Secure Implementations of Protocols

(Work in Progress)

David Cadé
École Normale Supérieure, CNRS, INRIA

19 June 2009

Motivation

Get automatically a proved implementation in ML of a cryptographic protocol, by:

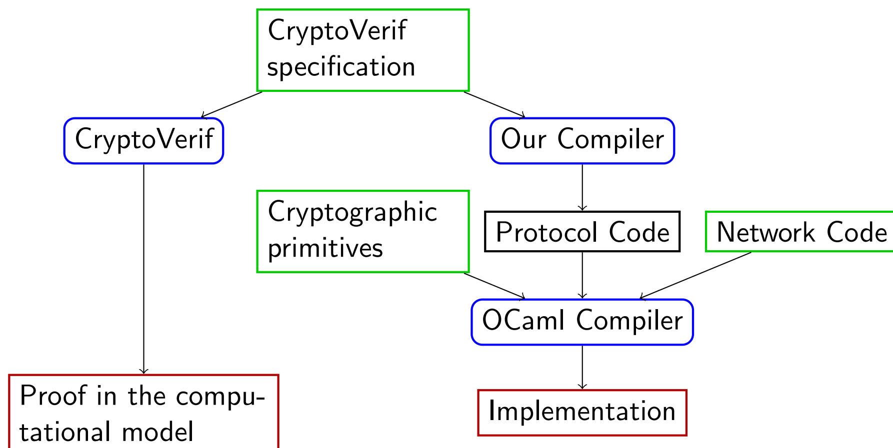
- Annotating a CryptoVerif specification,
- Compiling it into protocol code.

Related work

Bhargavan et al.: From protocol implementation in ML, generate a CryptoVerif specification and then prove it.

- We are doing it the other way around !
- Our goal is the same: we want at the end a proved implementation of cryptographic protocols.

Overview of the Tool



Outline

Motivation

The CryptoVerif input language

Language annotations

Code generation

Description of CryptoVerif

CryptoVerif is a verifier of cryptographic protocols in the computational model written by Bruno Blanchet.

- Generates proofs by sequences of games,
- Uses a probabilistic polynomial-time process calculus.

Example

$$A \longrightarrow B : \underbrace{\{k\}_{Kab}}_{e}, mac(e, mKab)$$

```
process Ostart () :=
        rKab <-R- keyseed; Kab <- kgen(rKab);
        rmKab <-R- mkeyseed; mKab <- mkgen(rmKab);
        return ();
        ( foreach i1 ≤ N do processA |
          foreach i2 ≤ N do processB)
```

- The Ostart oracle generates Kab and mKab.
- These symmetric keys 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

$$A \longrightarrow B : \underbrace{\{k\}_{Kab}}_{e}, mac(e, mKab)$$

```
let processA = OA() :=
               k2 <-R key; s1 <-R seed;
               ea1 <- enc(keyToBitstring(k2), Kab, s1);
               return (ea1,mac(ea1, mKab)).

let processB = OB(ea:bitstring, ma:macs) :=
               if check(ea, mKab, ma) then
               let injbot(keyToBitstring(k3:key)) = dec(ea, Kab) in
               return ().
```

- OB checks that the received message is actually correct, verifying the MAC and the ciphertext.

Example — summary

let processA = OA() :=
$\quad$ k2 $\xleftarrow{R}$ key; s1 $\xleftarrow{R}$ seed;
$\quad$ ea1 ← enc(keyToBitstring(k2), Kab, s1);
$\quad$ return (ea1,mac(ea1, mKab)).

let processB = OB(ea:bitstring, ma:macs) :=
$\quad$ if check(ea, mKab, ma) then
$\quad$ let injbot(keyToBitstring(k3:key)) = dec(ea, Kab) in
$\quad$ return ().

process Ostart () :=
$\quad$ rKab $\xleftarrow{R}$ keyseed; Kab ← kgen(rKab);
$\quad$ rmKab $\xleftarrow{R}$ mkeyseed; mKab ← mkgen(rmKab);
$\quad$ return ();
$\quad$ (foreach i1 ≤ N do processA |
$\quad$ foreach i2 ≤ N do processB)

Separation in multiple programs

```
let processA = pA { OA() :=
                    k2 <-R key; s1 <-R seed;
                    ea1 <- enc(keyToBitstring(k2), Kab, s1);
                    return (ea1,mac(ea1, mKab))}.

let processB = pB { OB(ea:bitstring, ma:macs) :=
                    if check(ea, mKab, ma) then
                    let injbot(keyToBitstring(k3:key)) = dec(ea, Kab) in
                    return ()}.

process keygen [ Kab >Kab, mKab >mKab ] { Ostart () :=
        rKab <-R keyseed; Kab <- kgen(rKab);
        rmKab <-R mkeyseed; mKab <- mkgen(rmKab);
        return ()};
        ( foreach i1 <= N do processA |
          foreach i2 <= N do processB)
```

External data files

```
let processA = pA { OA() :=
                    k2 <-R key; s1 <-R seed;
                    ea1 <- enc(keyToBitstring(k2), Kab, s1);
                    return (ea1,mac(ea1, mKab))}.

let processB = pB { OB(ea:bitstring, ma:macs) :=
                    if check(ea, mKab, ma) then
                    let injbot(keyToBitstring(k3:key)) = dec(ea, Kab) in
                    return ()}.

process keygen [ Kab >Kab, mKab >mKab ] { Ostart () :=
        rKab <-R keyseed; Kab <- kgen(rKab);
        rmKab <-R mkeyseed; mKab <- mkgen(rmKab);
        return ()};
        ( foreach i1 <= N do processA |
          foreach i2 <= N do processB)
```

Caveats of this approach

- It is necessary to separate the process into a key generator (keygen), the A part of the protocol (pA), and the B part (pB).
- We need to assume that the opponent will not be able to read the files.
- In general, we may need to assume that the opponent will launch the programs in the right order.

Implementation specific details

- Size of types used to generate random numbers :
  `implementation typesize keyseed 256.`
- OCaml function representing a function in the protocol specification :
  `implementation function check "check".`
  - In the CryptoVerif specification, there are assumptions about these functions.
  - Verifying that the OCaml function satisfies them is out of the scope of this project.

Definition of atomic execution units

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

```
let processA = pA { OA() :=
                 k2 <-R key; s1 <-R seed;
                 ea1 <- enc(keyToBitstring(k2), Kab, s1);
                 return (ea1,mac(ea1, mKab))}.
```

The module PA generated has the following interface :

```
open Crypto

val init :  unit -> unit
val oracle_OA : unit -> bitstring
```

Control Flow

```
pA {
( Oa () :=
  return ();
  foreach i ≤ N do
        Ob () :=
        return ();
        ( ( Oc () :=
            return ())
          |((foreach j ≤ N' do
                Of () :=
                return ();
                Og () :=
                return ())
        )))
```

```
open Crypto

val init : unit → unit
val oracle_Oa :
    unit → bitstring
val oracle_Ob :
    unit → (bitstring*int)
val oracle_Oc :
    unit → int → bitstring
val oracle_Of :
    unit → int → (bitstring*int)
val oracle_Og :
    unit → int → int → bitstring
```

After a replication, the oracles need the replication indices (i,j).

- The first oracle after a replication generates the index.
- The following ones takes it into argument.

Control Flow

```
pA {
( Oa () :=
  return ();
  foreach i ≤ N do
          Ob[i] () :=
          return ();
          ( ( Oc[i] () :=
            return ())
           |((foreach j ≤ N' do
                  Of[j,i] () :=
                  return ();
                  Og[j,i] () :=
                  return ())
          )))
```

```
open Crypto

val init : unit → unit
val oracle_Oa :
    unit → bitstring
val oracle_Ob :
    unit → (bitstring*int)
val oracle_Oc :
    unit → int → bitstring
val oracle_Of :
    unit → int → (bitstring*int)
val oracle_Og :
    unit → int → int → bitstring
```

After a replication, the oracles need the replication indices (i,j).

- The first oracle after a replication generates the index.
- The following ones takes it into argument.

Control Flow

- We keep an environment variable `envx` in the generated code which knows which oracle can be launched at the point of execution.
- When you launch an oracle you should not have launched, then the code raises an exception.
- On quitting the oracle function, this variable is updated.

How the variables are handled

- There is an environment variable `env` that stores every variable that is needed over two oracles, or read from file.
- The `init` function reads from the files the variables needed and stores them in the environment.
- At the beginning of a function, we load the needed variables from environment.

Translation of an Oracle

1. We first check if the oracle can be launched.
2. Then we get all the variables required.
3. We then match the arguments of the oracle.
4. For each instruction of the oracle, we translate it in OCaml:
   - `x` $\xleftarrow{R}$ `t; P'` becomes `let` [[x]] `= random [sizeof t] in` [[P']] with `[sizeof t]` replaced by the actual declared size of `t`.
   - `x` $\leftarrow$ `M; P'` becomes `let` [[x]] `=` [[M]] `in` [[P']].
   - `if (test) then P else P'` becomes `if (get_bool (`[[test]]`)) then` [[P]] `else` [[P']]
   - In `return x` we update `envx`, and then return [[x]].
   - the `find` construct is a non trivial element to translate.
   - ...

Future work

- The proof of correctness. Correspondence between traces
- Test on real-world examples. For example, for a given protocol, test if our implementation can talk to an already implemented solution.

Conclusion

- We have built a compiler that takes a CryptoVerif specification as input and generates an OCaml implementation.
- Our approach favors the methodology of:
  1. Writing a specification,
  2. Proving it,
  3. Then, building an implementation.