EasyCrypt - Lecture 2
An introduction to EasyCrypt

Benjamin Grégoire

Monday November 24th

Bellare and Rogaway 93

Definition (BR93 encryption scheme)

Let $\mathcal{M}$ the type of message, $\mathcal{R}$ the type of randomness. Let $(\mathcal{K}_f, f, f^{-1})$ be a family of trapdoor permutations on $\mathcal{R}$ and $G : \mathcal{R} \to \mathcal{M}$ a hash functions.
The BR93 sheme is composed of:

$$\begin{array}{lcl} \mathrm{kg}() & = & pk, sk = \$\mathcal{K}_f;\ \text{return } (pk, sk) \\ \mathrm{enc}(pk, m) & = & r = \$\mathcal{R};\ \text{return } (f\ \ pk\ \ r, m \oplus G\ \ r) \\ \mathrm{dec}(sk, c) & = & (s, t) = c;\ \ r = f^{-1}\ \ sk\ \ s;\ \text{return } t \oplus G\ \ r \end{array}$$

Questions: How to formalize types, operators, distribution, algebraic properties, scheme, parametric games, adversaries ?

Specification of basic type and operations

In EasyCrypt, one can be declare types and operators:

```
type plain.
type rand.
type cipher = rand * plain.

op zero  : plain.
op ( + ) : plain → plain → plain.

op G : rand → plain.

op xor0 (x:rand) = x + zero.
```

EasyCrypt also allows to deal with polymorphic type, data type and operators ('a list, **op** (::) : 'a → 'a list → 'a list), high order operators.

Properties of basic operations

Example the type plain is a group of order 2:

```
axiom xor0p: forall (x:plain), zero + x = x.

axiom xorC (x y:plain): x + y = y + x.

axiom xorA x y z : x + (y + z) = (x + y) + z.

axiom xorN x : x + x = zero.

lemma xorp0 x : x + zero = x by smt.

lemma foo x y : (x + y) + y = x by smt.
```

EasyCrypt allows to use smt solver but also provide a tactic language (close to Coq/Ssreflect) to perform proofs interactively.

Specification of random operators

Random operators are defined using a special polymorphic type distr and an operator mu returning the probability of an event in a given discrite sub-distribution:

---

type 'a distr.

op mu : 'a distr $\rightarrow$ ('a $\rightarrow$ bool) $\rightarrow$ real.

---

mu d E should be understood as

$$\Pr[x = \$d : E\ x]$$

Basic properties of mu

```
axiom mu_bounded (d:'a distr) (p:'a → bool):
  0 ≤ mu d p ≤ 1.

axiom mu_false (d:'a distr):
  mu d False = 0.

axiom mu_or (d:'a distr) (p q:'a → bool) :
  mu d (p ∨ q) = mu d p + mu d q − mu d (p ∧ q).
```

Specification of random operators

```
op mu_x (d:'a distr) x = mu d ((=) x).

op in_supp x (d:'a distr) = 0 < mu_x d x.

pred isuniform (d:'a distr) =
  forall (x y:'a),
    in_supp x d => in_supp y d => mu_x d x = mu_x d y.
```

Example uniform distribution over Booleans:

```
op dbool : bool distr.

axiom mu_x_def b : mu_x dbool b = 1/2.

lemma dbool_uni : isuniform dbool.
```

Specification of random operators

```
type rand.

op drand : rand distr.

axiom drand_lossless : mu drand True = 1.

axiom drand_uniform : isuniform drand.
```

Specification of random operators

One possible formalisation of f and f^{-1}:

```
type pkey, skey.
type keys = pkey * skey.

op keygen : keys distr.

op f    : pkey → rand → rand.
op finv : skey → rand → rand.

axiom finvof pk sk x: in_supp (pk,sk) keygen ⇒
   finv sk (f pk x) = x.

axiom fofinv pk sk x: in_supp (pk,sk) keygen ⇒
  f pk (finv sk x) = x.
```

BR93

```
module BR93 = {
  proc kg(): keys = { var ks; ks = $keygen; return ks; }

  proc enc(pk:pkey,m:plain) : cipher = {
    var r;
    r = $drand;
    return (f pk r, m + G r);
  }

  proc dec(sk:skey,c:cipher) : plain = {
    var s,t;
    (s,t) = c;
    return t + G (finv sk s);
  }
}
```

Modules are a keystone of EasyCrypt

Specification of schemes, oracles, adversaries, cryptographic assumptions, game-based properties are based on modules

- Manage complexity by abstraction
- Supporting high-level reasoning steps: reduction, hydrid argument, ...

Content of a module

```
module M = {            (* name of the module *)
  var m : t             (* global variable declarations *)
  var m1, m2 : t
  proc h(x: int) : int = { (* procedure definitions *) }

  module N =            (* sub module definitions *)
}.
```

Some restrictions:

- Types, operators and predicates cannot be declared/defined inside a module
- No polymorphism : variables and procedures are monomorphic

Remark:

Polymorphism can be recovered using theory and cloning (tomorrow)

Example: Random Oracle

```
module G = {
  var m : (rand, plain) map

  proc init () : unit = { m = empty; }

  proc o (x:rand) : plain = {
      var r : int;
      r = $dplain;
      if (!mem x (dom m)) m.[x] = r;
      return m.[x];
    }
  }.
```

Declare a module G with a global variable m and a procedure o. Outside of the module the variable is denoted G.m and the function G.o.

Modules can use external modules

```
module BR93 = {
  ...
  proc enc(pk:pkey, m:plain): cipher = {
    var h, r;

    r = $drand;
    h = G.o(r);
    return (f pk r, m + h);
  }
  ...
}.
```

Modules allows to encode parametric games

Example:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{Game CPA} =} \\
\quad (pk, sk) & = & S.\mathrm{kg}(); \\
\quad (m_0, m_1) & = & A.choose(pk); \\
\quad b & =\$ & \{0, 1\}; \\
\quad c & = & S.\mathrm{enc}(pk, m_b); \\
\quad b' & = & A.guess(c); \\
\multicolumn{3}{l}{\quad \text{return } b' = b;}
\end{array}
$$

The game is paremetrized by two other modules: S and A.

- S provide at least the procedure $S.\mathrm{kg}$ and $S.\mathrm{enc}$
- A provide at least the procedure $S.choose$ and $S.guess$

Modules can be parameterized by other modules: Functors

A module type is an abstraction of a module

```
module type Adv = {                       (* name of the module type *)
  proc choose (_:pkey) : plain * plain  (* procedure declarations   *)
  proc guess  (c:cipher) : bool
}.
```

Remarks:

- A procedure declaration contains the type of its parameters and possibly their names (used during specification and proof)
- Module types cannot contain variable or module declarations

Modules can be parameterized by other modules: Functors

```
module CPA (S:Scheme, A:Adv) = {
  proc main () : bool = {
    var pk,sk,m0,m1,b,c,b';
    (pk,sk)   = S.kg();
    (m0,m1) = A.choose(pk);
    b         = ${0,1};
    c*        = S.enc(pk, b?m0:m1);
    b'        = A.guess(c*);
    return b' = b;
}.
```

Remark:

The procedures A.choose and A.guess can share procedures and memory (active adversary)

Functors can be applied to other modules

```
module CPA_BR  = CPA(BR93).        (* Partial application *)

module CPA_BRA = CPA(BR93, A).  (* Full application *)
```

Remark:

EasyCrypt ensure that the application is well formed.

Higher order modules

Example CPA in the random oracle model:

```
module type Ro = { proc o(_:rand) : plain }.

module type RoSch (O:Ro) = { proc enc ... }.

module type RoAdv (O:Ro) = { proc a1 ... }.

module G : Ro = { ... }.

module RoCPA (S:RoSch, A:RoAdv) = {
  module S = S(G)
  module A = A(G)
  proc main () : bool = {
     ...
  }
}
```

Adversaries are represented using “abstract module”

EasyCrypt allows quantification over modules:

```
lemma security:
  forall (A <: Adv),
    Pr[CPA(A).main : res] − 1/2 ≤  Pr[OW(I(A)).main : res].
```

Adversaries can be declared localy inside section:

```
section.
  declare module A : Adv.
    ...
 lemma security:
  Pr[CPA(A).main : res] − 1/2 ≤  Pr[OW(I(A)).main : res].
end section.
```

Back to the CPA game

```
module CPA (S:Scheme, A:Adv) = {
  proc main () : bool = {
    var pk,sk,m0,m1,b,c,b';
    (pk,sk)   = S.kg();
    (m0,m1) = A.choose(pk);
    b         = ${0,1};
    c*        = S.enc(pk, b?m0:m1);
    b'        = A.guess(c*);
    return b' = b;
}.
```

Remark:

In the literature, the IND-CPA, IND-CCA1, IND-CCA properties are all defined using the same basic game. Only the capabilities of the adversary change.

Capabilities of adversary

	IND-CPA	IND-CCA1	IND-CCA
A.choose	_	S.dec(sk,$\cdot$)	S.dec(sk,$\cdot$)
A.guess	_	_	S.dec(sk,$\cdot$) $\setminus\{c^*\}$

Sometimes the number of queries allowed to S.dec(sk) is also limited

The module system can help to capture those different notions

A first try, declaration of the IND-CCA adversary

```
module type DEC = {
  proc dec(c:cipher) : plain option
}.

module type ADV(D:Dec) = {
  proc choose (pk:pkey) : plain * plain
  proc guess  (c:cipher) : bool
}.
```

Remark:

This does not capture the notion of IND-CCA1 adversary, since the guess function can call the decryption oracle

A more restrictive module type system

For each procedure of a module type it is possible to select which procedures provided by the module parameters can be called

```
module type DEC = { proc dec(c:cipher) : plain }
module type ADVCCA1(D:DEC) = {
  proc choose (pk:pkey) : plain * plain  { D.dec }
  proc guess  (c:cipher) : bool          { }
}.
```

Here *choose* can call D.dec whereas *guess* cannot.

The notation

proc choose (pk:pkey) : plain $*$ plain

is a shortcut for

proc choose (pk:pkey) : plain $*$ plain { *all procedures* }

IND-CCA : using the type module system

We can split the decryption oracle in two (one for choose and one for guess)

```
module type DEC2 = {
  proc dec_c(c:cipher) : plain
  proc dec_g(c:cipher) : plain
}.

module type ADVCCA(D:DEC2) = {
  proc choose (pk:pkey) : plain * plain { D.dec_c }
  proc guess   (c:cipher) : bool         { D.dec_g }
}.
```

IND-CCA : the decryption oracle

The decryption oracle in the guess stage can now log the adversary queries:

```
module D : DEC2 = {
  var sk : skey
  var log : cipher list

  proc dec_c (c:cipher) : plain = {
    var r; r = S.dec(sk, c); return r;
  }

  proc dec_g (c:cipher) : plain = {
    var r;
    log = c :: log; r = S.dec(sk, c);
    return r;
  }
}.
```

Three possibility to restrict capacities of adversaries

- Penalty style: the adversary is being penalized, a posteriori, for its actions

$$\Pr[CCA(A) : win \wedge c^* \notin log]$$

- Exclusion style: certain adversaries are a priori excluded from consideration

$$\Pr[A.guess(c^*) : c^* \in log] = 0$$

- Enforcement style: no restriction in the adversary, the policy is enforced by oracle

```
proc dec_g (c:cipher) : plain = {
  var r = default;
  if (c ≠ c*) then r = S.dec(sk, c);
  return r;
}
```

Three possibility to restrict capacities of adversaries

It is possible to use the three kinds of restriction in EasyCrypt

The enforcement style is generaly simpler to use

This kind of techniques can be use to bound the number of oracle calls

There is a last kind of constraint which is generaly “ignored” by cryptographers

Negative constraints

```
module B = { var b:int }.

module G(A:Adv) = {
  var x' : int;
  proc main () : unit = {
     B.b = ${0,1};
     b'  = A.f();
  }
}

lemma F : forall (A<:Adv), Pr[G(A).main : B.b = G.b'] = 1/2.
```

Can we prove such a lemma ?

Negative constraints

The answer is “no”: take the following module A1

```
module G(A:Adv) = {
  var x' : int;
  proc main () : unit = {
     B.b = ${0,1};
     b'  = A.f();
  }
}

module A1 = {
  proc f () : bool = { return B.b; }
}
```

we have Pr[G(A1).main : B.b = G.b'] = 1.

Negative constraints

EasyCrypt allows to restrict the quantification over adversary using negative constraints:

---

lemma T :
  forall (A<:Adv{B}),
    Pr[G(A).main : B.b = G.b'] = 1/2.

The "(A <:Adv{B})" should be understand as

for all "adversary" A whose implementation does not use the "memory space" of B

Conclusion

- EasyCrypt allows to specify types and operators
- Properties on operators can be proved using smt solver or interactively
- Schemes and security definitions can be defined using modules
- Adversaries are represented using universal quantification over module
- How to prove properties of modules?