Modules

Modules

- Manage complexity by abstraction
- Instantiating generic transformations (simplified syntax)

---

forall &m (A <: AdvCCA), *exists* (B <: AdvCPA),
    Pr[CCA(FO(S),A) @ &m : b’ = b] <=
    Pr[CPA(S,B) @ &m : b’ = b] + ...

- Supporting high-level reasoning steps

Modules are a keystone of EasyCrypt

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

Remark: There are some major differences with the Ocaml notion of module

Content of a module

```
module M = {                 (* name of the module *)
   var m : t                 (* global variable declarations *)
   var m1, m2 : t

   fun 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 sections and cloning (next talk)

My first module

```
module RO = {
  var m : (int, int) map

  fun h (x:int) : int = {
    var r : int;
    r = $[0..10];
    if (!in_dom x m) m.[x] = r;
    return proj (m.[x]);
  }
}.
```

Declare a module “RO” with a global variable “m” and a function “f”.
Outside of the module the variable is denoted “RO.m” and the function “RO.f”

Modules can use external modules

```
module M1 = {
   var x : int
   fun f (i:int) : int = { ... }
}.

module M2 = {
   fun g (i:int) : int = {
      M1.x = M1.x + 1;
      i = M1.f(i);
      return i;
   }
}.
```

Module types

A module type is an abstraction of a module

```
module type ADV = {                          (* name of the module type *)
  fun choose (pk:pkey) : msg * msg           (* procedure declarations *)
  fun guess (c:cipher) : bool
}.
```

Remarks:

- A procedure declaration contains the names of its parameters (it is used during specification and proof)
- Module types cannot contain variable and module declarations
- A module M has type I if it contains at least the procedures declared in I (with the correct types)

Modules can be parameterized by other modules: Functors

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

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

Restriction: A sub-module cannot be a functor

Functor application

It is possible to define a module by (partially) applying a functor to other modules.

```
module A : ADV = { .... }.          (* Structure *)

module CPA' = CPA.                  (* Alias *)

module CPAS = CPA(S).               (* Partial application *)

module CPASA = CPA(S,A).            (* Full application *)
```

Restrictions:

- Functor arguments should have the expected type
- Partial application of a functor is not allowed for sub-modules.

The memory model

```
module M1 = { var x : int }.

module M2 = M1.

module T = {
  fun f () : unit = { M1.x = 1; M2.x = 2 }
}.
```

Questions: After the execution of T.f

- what is the value of M2.x?
- what is the value of M1.x?

The memory model

```
module type Empty = {}.
module E : Empty = {}.

module F(I:Empty) = { var x : int }.

module M1 = F(E).
module M2 = F(E).

module T = {
  fun f () : unit = { M1.x = 1; M2.x = 2 }
}.
```

Questions: After the execution of T.f

- what is the value of M2.x and M1.x in Ocaml?
- what is the value of M2.x and M1.x in EasyCrypt?

Functor application is not generative

A functor should be understood as a pair of:

- A memory space (the global variables declared in the module)
- A set of procedures parameterized by the procedures provided by the module parameters (higher order)

Remarks:

- Functor application does not generate a new memory space.
- Global variables of a functor can be read or written without applying of the functor: F.x = F.x + 1;

To create a fresh memory space and recover the usual (Ocaml) semantics of functor application, use theories and cloning (see next talk).

Back to the CPA game

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

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

Capabilities of adversary

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

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 = {
  fun dec(c:cipher) : msg option
}.

module type ADV(D:Dec) = {
  fun choose (pk:pkey) : msg * msg
  fun 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 = { fun dec(c:cipher) : msg }
module type ADVCCA1(D:DEC) = {
  fun choose (pk:pkey) : msg * msg    { D.dec }
  fun guess  (c:cipher) : bool        { }
}.
```

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

the notation

```
fun choose (pk:pkey) : msg * msg
```

is a shortcut for

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

IND-CCA: the decryption oracle

The decryption oracle in the guess stage can now “reject” queries on the challenge.

```
module D : DEC2 = {
  var sk : skey
  var challenge : cipher

  fun dec_c (c:cipher) : msg option = {
    var r : msg option;
    r = S.dec(sk, c);
    return r;
  }
  fun dec_g (c:cipher) : msg option = {
    var r : msg = None;
    if (c <> challenge) r = S.dec(sk, c);
    return r;
  }
}.
```

EasyCrypt allows quantification over modules

```
forall &m (A <: Adv) :
   exists (I <: Inverter),
      Pr[CPA(A).main() @ &m : res] − (1%r/2%r) <=
         Pr[OW(I).main() @ &m : res].

forall (A<:Adv), equiv [CCA(A).main ~ G(A).main : ... ==> ...]
```

Allows to express formulas like:

- *Forall adversary A there exists a simulator B ...*
- *There exists a simulator B such that forall adversary A ...*

Restriction: Formulas containing quantification over abstract module are not sent to SMT provers

Negative constraints

```
module X = { var x : int }.
module G(A:Adv) = {
  fun g () : unit = {
    X.x = 3;
    A.f();
  }
}.

lemma F : forall (A<:Adv), hoare[G(A).g : true ==> X.x = 3].
```

Can we prove such a lemma ?

Negative constraints

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

```
module G(A:Adv) = {
  fun g () : unit = { X.x = 3; A.f(); }
}.

module A1 = {
  fun f() : unit = { X.x = 4; }
}

lemma F : forall (A<:Adv), hoare[G(A).g : true ==> X.x = 3].
```

But F becomes true, if we restrict the quantification to modules that do not use the “memory of X” (here simply do not write).

Negative constraints

EasyCrypt allow to restrict the quantification over adversary, using negative constraints:

```
lemma T : forall (A<:Adv{X}), hoare[G(A).g : true ==> X.x = 3].
```

The “forall (A<:Adv{X})” should be understand as

for all “adversary” A whose implementation do not use the “memory space” of X.

What is the memory space of a module ?

The memory space of a module M is:

- The global variables declared inside the module (and its sub-modules)
- The global variables of the external modules used in M (also indirectly)

Restrictions are checked during instantiation

```
lemma T : forall (A<:Adv{X}), hoare[G(A).g : true ==> X.x = 3].

module A2 = { }.
lemma Error1 : hoare[G(A2).g : true ==> X.x = 3].
```

Error message: invalid module application: arguments do not match required interfaces

```
lemma Error2 : hoare[G(A1).g : true ==> X.x = 3].
  apply (T A1).
```

Error message: the module A1 should not use X

Reasoning over universally quantified modules

Main difficulties

- We need rules to perform proofs on universally quantified modules
- The rules should be valid independently of the *implementation* of the module

Example:

forall (A<:Adv), **equiv**[A(RO).f ~ A(RO').f : true ==> ={res}]

This formula should be valid for all A. So in particular for a module A depending on the module RO (or RO').

For the presentation: $A(o).f$ should be understood as function $A(O).f$ where the function f can only call the function $O.o$

Hoare rule for adversary

$$\frac{o : I \Longrightarrow I}{A(o).f : I \Longrightarrow I}$$

Restriction:
The invariant I should not depend on program variables that can be written by A.

Syntax: fun I

pRHL **rule for adversary**

$$\frac{o_1 \sim o_2 : y_1\langle 1\rangle = y_2\langle 2\rangle \wedge I \Longrightarrow =\{res\} \wedge I}{A(o_1).f \sim A(o_2).f : =\{x, \text{glob } A\} \wedge I \Longrightarrow =\{res, \text{glob } A\} \wedge I}$$

where x is the parameter $A.f$ and y_i the parameter of o_i
glob A correspond to the memory space of A

Restriction: The invariant I should not depend of program variables that can be written by A.

Syntax: fun I

Fundamental lemma

A very frequent step in cryptographic proof:

$$\Pr[G : E] \leq \Pr\left[G' : E\right] + \Pr\left[G' : B\right]$$

To do this we should proof that:

$$\Pr[G : E] \leq \Pr\left[G' : E \vee B\right]$$

This can be established using pRHL:

$$G \sim G' : \top \Longrightarrow \neg B\langle 2\rangle \Rightarrow E\langle 1\rangle = E\langle 2\rangle$$

(see day1/proba.ec)

How to establish such a property?

the upto bad rule for adversary(simplified)

$$\frac{\begin{array}{l} o_1 \sim o_2 : \neg B\langle 2\rangle \wedge =\{y\} \wedge I \Longrightarrow \neg B\langle 2\rangle \Rightarrow =\{res\} \wedge I \\ o_2 : B \Longrightarrow B \end{array}}{\begin{array}{l} A(o_1).f \sim A(o_2).f : \\ \quad \neg B\langle 2\rangle \Rightarrow =\{x, \text{glob } A\} \wedge I \quad \Longrightarrow \\ \quad \neg B\langle 2\rangle \Rightarrow =\{res, \text{glob } A\} \wedge I \end{array}}$$

Also some *anecdotic* looslessness side condition ...

Syntax: fun B I

concrete versus abstract

Concrete/Concrete	fun	
Abstract/Abstract	fun I	
	fun B I	(upto)
Concrete/Abstract	fun *	