EasyCrypt - Lecture 4
Theories, Cloning and Sections

Tuesday November 25th

Situation

Previously, on EasyCrypt:

- Game-based proofs
- Writing proof sketches in EasyCrypt
- pRHL and the proof engine

And now:

- Theory: collection of related concepts and lemmas;
- Cloning: copying a theory;
- Instantiation: refining a theory;
- Basic sections: isolating a proof artefact.

Theories

Three main uses for theories:

- mathematical constructions and datatypes (Bool, Bitstring, AWord...);
- common cryptographic notions and constructions (OWTP, PKE, ROM...);
- generic reasoning steps (next lecture).

Some are defined once and for all (Lists, Finite Sets), others are parameterized by values or operators (AWord) or types (cryptographic constructions).
Contain lemmas derived from basic axioms on abstract types and operators.

Running Example: A (Very) Simple SRG

```
type output, seed.
op D_seed: seed distr.
axiom D_seed_ll: mu D_seed_ll True = 1%r.
op F^c: seed -> int -> output.

module SRG = {
  var s:seed
  var c:int

  proc init(): unit = { s = $ D_seed; c = 0; }

  proc next(): output = {
    var r = F^c s c;
    c = c + 1;
    return r;
  }
}.
```

- A set of outputs, and a set of seeds equipped with an arbitrary distribution.
- Function family F^c from integers to outputs indexed by seeds.
- Internal state initially a random seed and 0.
- Query output produced by applying F to internal state; increment state.

Running Example: A (Very) Simple SRG

```
type output, seed.
op D_seed: seed distr.
axiom D_seed_ll: mu D_seed_ll True = 1%r.
op F^c: seed -> int -> output.

module SRG = {
  var s:seed
  var c:int

  proc init(): unit = { s = $ D_seed; c = 0; }

  proc next(): output = {
    var r = F^c s c;
    c = c + 1;
    return r;
  }
}.
```

- Types output and seed, distribution $\mathcal{D}_{seed}$, function family F^c are abstract.
- Can later be instantiated...
- For now, we focus on defining security notions.

Security Assumption: PRFs

```
type K, D, R.
op dK: K distr.
axiom dK_ll: mu dK True = 1%r.
op dR: R distr.
axiom dR_ll: mu dR True = 1%r.
op F: K → D → R.

module Real = {
  var k:K
  proc init(): unit = { k = $ dK; }
  proc f(x:D): R = { return F k x; }
}.

module Ideal = {
  var m:(D,R) map
  proc init(): unit = { m = [⊥]; }
  proc f(x:D): R = {
    if (x ∉ dom m) m.[x] = $ dR;
    return m.[x];
  }
}.
```

```
module type PRF = {
  proc init(): unit
  proc f(_:D): R
}.

module type Distinguisher(F:PRF) = {
  proc distinguish(): bool {F.f}
}.

module IND(F:PRF,D:Distinguisher) = {
  module D = D(F)
  proc main(): bool = {
    F.init();
    return D.distinguish();
  }
}.
```

Cloning

Q: How can we assume that F^c is a secure PRF *without* having to rewrite and specialize all those modules by hand?

Cloning creates a fresh verbatim copy of the theory.

```
theory Monoid.
  type m.
  op zerom: m.
  op (+): m → m → m.

  axiom addm0 x: m + zerom = m.
  axiom addmA x y z: x + (y + z) = (x + y) + z.
end Monoid.

clone Monoid as M.
```

Cloning – continued

The command **print** M yields:

```
theory M.
  type m.
  op zerom: m.
  op (+): m → m → m.

  axiom addm0 x: m + zerom = m.
  axiom addmA x y z: x + (y + z) = (x + y) + z.
end M.
```

Monoid.m and M.m are *distinct* abstract types... We now have two types equipped with distinct monoid structures.

Refining Clones – By Instantiation

Abstract symbols can be *refined* in two ways when cloning.
By instantiation:

```
clone Monoid as BMono with
  type m ← bool,
  op zerom ← false.
```

yields

```
theory BMono.
  op (+): bool → bool → bool.

  axiom addm0 x: m + false = m.
  axiom addmA x y z: x + (y + z) = (x + y) + z.
end BMono.
```

Refining Clones – Discharging Axioms

On cloning and refinement, axioms may become provable.

```
clone Monoid as BMono with
  type m ← bool,
  op zerom ← false,
  op (+) = ⊗
proof addm0, addmA.
  realize addm0. smt. qed.
  realize addmA. smt. qed.
```

Axioms can be discharged in any order.

proof $*$. requires all axioms in the refined theory to be discharged.

proof <axioms> by <tactic>. realizes the listed <axioms> and attempts to use <tactic> to discharge them.

Instantiating the Assumption

```
type output, seed.
op D_seed: seed distr.
axiom D_seed_ll: mu D_seed_ll True = 1%r.
op F^c: seed → int → output.

module SRG = {
  var s:seed
  var c:int

  proc init(): unit = { s = $ D_seed; c = 0; }

  proc next(): output = {
    var r = F^c s c;
    c = c + 1;
    return r;
  }
}.
```

```
op D_out: output distr.
axiom D_out_ll: mu D_out True = 1%r.

clone PRF with
  type K ← seed,
  op   D_K ← D_seed,
  type D ← int,
  type R ← output,
  op   D_R ← D_out,
  op   F ← F^c
proof * by smt.
```

Security Goal: PRGs

```
type O.
op dO: O distr.
axiom dO_ll: mu dO True = 1%r.

module Ideal = {
  proc init(): unit = { }
  proc next(): O = {
    var r;
    r = $ dO;
    return r;
  }
}.
```

```
module type PRG = {
  proc init(): unit
  proc next(): O
}.

module type Distinguisher(G:PRG) = {
  proc distinguish(): bool {G.next}
}.

module IND(G:PRG,D:Distinguisher) = {
  module D = D(G)
  proc main(): bool = {
    G.init();
    return D.distinguish();
  }
}.
```

Instantiating the Goal

```
type output, seed.
op D_seed: seed distr.
axiom D_seed_ll: mu D_seed_ll True = 1%r.
op F^c: seed → int → output.

module SRG = {
  var s:seed
  var c:int

  proc init(): unit = { s = $ D_seed; c = 0; }

  proc next(): output = {
    var r = F^c s c;
    c = c + 1;
    return r;
  }
}.
```

```
op D_out: output distr.
axiom D_out_ll: mu D_out True = 1%r.

clone PRF with
  type K ← seed,
  op   D_K ← D_seed,
  type D ← int,
  type R ← output,
  op   D_R ← D_out,
  op   F ← F^c
proof * by smt.

clone PRG with
  type O ← output,
  op   D_O ← D_out
proof * by smt.
```

lemma (D <: Distinguisher$^{\mathrm{PRG}}$): ∃ (D' <: Distinguisher$^{\mathrm{PRF}}$),
'|Pr[IND$^{\mathrm{PRG}}$(Real$^{\mathrm{PRG}}$,D): *res*] − Pr[IND$^{\mathrm{PRG}}$(Ideal$^{\mathrm{PRG}}$,D): *res*]|
= '|Pr[IND$^{\mathrm{PRF}}$(Real$^{\mathrm{PRF}}$,D'): *res*] − Pr[IND$^{\mathrm{PRF}}$(Ideal$^{\mathrm{PRF}}$,D'): *res*]|.

Other Uses of Cloning

- Instantiate cryptographic constructions:

```
clone ROM as H with
  type input ← ...
proof * ...
```

- Specialize parameterized data structures:

```
op k:int.
axiom lt0k: 0 < k.

clone Word as Rand with
  op length ← k,
proof * by smt.
```

Other Uses of Cloning – Continued

Refining down to implementations:

- Proofs performed on abstract axiomatized structures
  Example: BR93 from yesterday was done on abstract types plain and rand equipped with algebraic sturctures.
- Cloning and instantiation lets you push any proofs down to the level of concrete datatypes

---

clone BR93 **with**
  type plain $\leftarrow \{0,1\}^p$,
  op dplain $\leftarrow \mathcal{U}_{\{0,1\}^p}$,
  type rand $= \{0,1\}^k$,
  op dplain $\leftarrow \mathcal{U}_{\{0,1\}^k}$,
  op f $\leftarrow$ rsa$_{k+p+1}$,
  $\vdots$
proof $*$ by ....

- Core libraries can be concretely realized.

Miscellaneous benefits of cloning

Abstraction often helps SMT solvers, and with the complexity of interactive proofs.

Abstraction helps identify common proof techniques and constructions that can be turned into program transformations and generic proof libraries (next lecture).

Structuring Proofs: Sections

In BR93, a Section was used to simplify notations.

```
section.
  declare module A:AdvCPA {CPA,OW,RO,BR}.
  axiom a1_ll (O:Oracle): islossless O.enc ⇒ islossless A(O).a1.
  axiom a2_ll (O:Oracle): islossless O.enc ⇒ islossless A(O).a2.
  :
  lemma Reduction &m:
    Pr[CPA(BR,A).main() @ &m : res]
    ≤ 1%r / 2%r + Pr[OW(BR_OW(A)).main() @ &m: res].
end section.

print Reduction.
```

yields

```
lemma Reduction (A <: AdvCPA {CPA,OW,RO,BR}):
  (forall (O:Oracle), islossless O.enc ⇒ islossless A(O).a1) ⇒
  (forall (O:Oracle), islossless O.enc ⇒ islossless A(O).a2) ⇒
  forall &m,
    Pr[CPA(BR,A).main() @ &m : res]
    ≤ 1%r / 2%r + Pr[OW(BR_OW(A)).main() @ &m: res].
```

Section Locals

Another use: hiding proof artefacts.

```
section.
  declare module A:AdvCPA {CPA,OW,RO,BR}.
  axiom a1_ll (O:Oracle): islossless O.enc ⇒ islossless A(O).a1.
  axiom a2_ll (O:Oracle): islossless O.enc ⇒ islossless A(O).a2.

  local module G1 = {
    var r:rand

    ...
  }.
end section.
```

Without **local**, A must be restricted with BR2.
Marking BR2 as **local** allows it to be used in local lemmas and modules, and in proofs inside the section, but causes it to disappear from the namespace on **end section**.

Summary

- Theories group related concepts together:
  - Data structures and algorithms;
  - Cryptographic primitives and assumptions;
  - Cryptographic constructions and security notions;
  - Generic reasoning steps.
- Cloning and refinement instantiate those concepts:
  - Parameterized data structures;
  - Applying generic constructions to concrete primitives;
  - Instantiation of generic reasoning steps in particular proofs (next).
- Sections keep security statements clean:
  - Isolating separate proof steps (reduction/bounding);
  - Keeping proof artefacts out of final statements.