The EasyCrypt Tool

The EasyCrypt Team
François Dupressoir

IMDEA Software Institute and INRIA

Pisa — June 3rd–6th, 2014

Welcome

Acknowledgements:

- The EasyCrypt team: Gilles Barthe, Pierre-Yves Strub, Benjamin Grégoire, César Kunz, Juan Manuel Crespo, Benedikt Schmidt

Organization:

- Lectures: overview of key components ($\sim$ 1/3)
- Labs: hands-on experience ($\sim$ 2/3)

Course URL (slides, exercises, related reading):

https://www.easycrypt.info/trac/wiki/CoursePisa2014

EasyCrypt in a nusthell

- EasyCrypt is a tool-assisted platform for proving security of cryptographic constructions in the computational model
- Views cryptographic proofs as relational verification of open parametric probabilistic programs

Goals:

- Leverage PL and PV techniques for cryptographic proofs
- Be accessible to cryptographers (choice of PL)
- Support high-level reasoning principles (still ongoing)
- Provide reasonable level of automation
- Reuse off-the-shelf verification tools (we use Why3)

EasyCrypt usage

- EasyCrypt is generic: no restrictions on
  - ☞ primitives and protocols
  - ☞ security notions and assumptions
- Can be used interactively or as a certifying back-end
  - ☞ for cryptographic compilers (Zero-Knowledge)
  - ☞ for domain-specific logics (ZooCrypt)
- Can verify implementations
  - ☞ C-mode (+ CompCert)
  - ☞ ML code extraction from verified specifications

EasyCrypt: Languages

A higher-order pure expression language:

- User-extensible,
- first-class distributions (α *distr*),
- Used to describe abstract functional primitives.

A typed imperative language (pWhile):

- Used to describe schemes, oracles, adversaries, games...

$\mathcal{C}$	$::=$	skip	skip
	$\mid$	$\mathcal{V} = \mathcal{E}$	assignment
	$\mid$	$\mathcal{V} = \$\mathcal{D}$	random sampling
	$\mid$	$\mathcal{C};\ \mathcal{C}$	sequence
	$\mid$	if $\mathcal{E}$ then $\mathcal{C}$ else $\mathcal{C}$	conditional
	$\mid$	while $\mathcal{E}$ do $\mathcal{C}$	while loop
	$\mid$	$\mathcal{V} = \mathcal{F}(\mathcal{E}, \ldots, \mathcal{E})$	procedure call

Semantics of programs

Discrete sub-distribution transformers

$$[\![c]\!] : \mathcal{M} \to \mathcal{M}\ \text{distr}$$

Probability of an event

$$\Pr[c, m : E] = [\![c]\!]_m\ E$$

Losslessness

$$\Pr[c, m : \top] = 1$$

EasyCrypt: Logics

- Hoare Logic

$$[c : P \Longrightarrow Q]$$

- Probabilistic Hoare Logic

$$[c : P \Longrightarrow Q] \leq \delta \quad [c : P \Longrightarrow Q] = \delta \quad [c : P \Longrightarrow Q] \geq \delta$$

- Probabilistic Relational Hoare Logic

$$[c_1 \sim c_2 : P \Longrightarrow Q]$$

- Ambient higher-order logic

$$\begin{aligned} &\forall\, \mathrm{c}_1, \mathrm{c}_2, \mathrm{m}_1, \mathrm{m}_2. \\ &\quad [c_1 \sim c_2 : \mathrm{true} \Longrightarrow =\{\mathrm{res}\}] \Rightarrow \\ &\quad \Pr[c_1, m_1 : \mathrm{res}] = \Pr[c_2, m_2 : \mathrm{res}] \end{aligned}$$

Lecture Plan

1. Functional Programs, Ambient Logic and Interactive Proofs
2. Formalizing Distributions
3. Interactions between EasyCrypt Logics
4. EasyCrypt Modules
5. Proving and Transforming Programs
6. Structuring Proofs
7. Advanced Tactics

Lecture 1

Functional Programs, Ambient Logic and Interactive Proofs

The Ambient Logic

EasyCrypt's ambient logic is a general higher-order logic.

In this lecture:

- How to specify facts about user-defined operators;
- How to prove them when automatic techniques do not work.

Topic 1

The EasyCrypt Core Language

Types

EasyCrypt is a typed language:

- Core types: unit, bool, int, real, distr
- Declaring abstract types:

  type t
  type α *u*

- Defining new types:

  type α *list* = [Nil | Cons of α & α *list*]
  type complex = {| r:real; i:real |}
  type intlist = int list
  type (α ,β) *pair* = $\alpha * \beta$

EasyCrypt Expressions

- Declaring abstract operators:

  map : $(\alpha \to \beta) \to \alpha$ *list* $\to \beta$ *list*
  fold : $(\beta \to \alpha \to \beta) \to \beta \to \alpha$ *list* $\to \beta$

- Defining concrete operators:

```
op max (x y : int) = (a < b) ? b : a.
op length (xs : 'a list) =
    with xs = Nil => 0
    with xs = Cons x xs => 1 + length xs.
op length_fold (xs : 'a list) = fold (fun v _, 1 + v) 0 xs.
```

- Can make use of let binders:

```
op fst (x:'a * 'b) = let (a,b) = x in a.
```

EasyCrypt Formulas

- Logical operators:

forall (x : t), ϕ	$(\forall\,(x : t), \phi)$	*exists* (x : t), ϕ	$(\exists\,(x : t), \phi)$
ϕ_1 => ϕ_2	$(\phi_1 \Rightarrow \phi_2)$	ϕ_1 <=> ϕ_2	$(\phi_1 \Leftrightarrow \phi_2)$
$\phi_1 \backslash \phi_2$	$(\phi_1 \wedge \phi_2)$	ϕ_1 / ϕ_2	$(\phi_1 \vee \phi_2)$
ϕ_1 && ϕ_2	$(\phi_1 \wedge (\phi_1 \Rightarrow \phi_2))$	ϕ_1 || ϕ_2	$(\phi_1 \vee (!\phi_1 \Rightarrow \phi_2))$
$!\phi$	$(\neg \phi)$		

- Quantification on memories: *forall* &m, ...,
- Quantification on modules: *forall* (M <: T), ...,
- HL, pHL and pRHL judgments,
- Probability expressions (given a module M and a memory &m): Pr[M.f(x) @ &m: E].

Axioms / Lemmas

- Specifying operators axiomatically:

  op count : 'a -> 'a list -> int.

  axiom count_nil (x : 'a): count x [] = 0.

  axiom count_cons_eq (x : 'a) (xs : 'a list):
  count x (x :: xs) = 1 + (count x xs).

  axiom count_cons_neq (x y : 'a) (xs : 'a list):
  x <> y => count x (y :: xs) = count x xs.

- Stating facts:

  lemma fact1 (x y : int): $x \leq 0 \Rightarrow y \leq 0 \Rightarrow 0 \leq x * y$.

  lemma fact2 (xs : α *list*): length xs = length_fold xs.

  lemma fact3 x (xs : α *list*): $0 \leq$ count x xs $\leq$ length xs.

Topic 2

Interactive Proofs

Stating a theorem

```
lemma mylemma b1 b2 b3 :
  (b1 => b2) => (b2 => b3) => b1 => b3.
proof. (* proof starts here *)
```

$$\left.\begin{array}{l} b1 : bool \\ b2 : bool \\ b3 : bool \end{array}\right\} \text{local hypotheses (context)}$$

$$\underbrace{(b1 \Rightarrow b2) \Rightarrow (b2 \Rightarrow b3) \Rightarrow b1}_{\text{assumptions}} \Rightarrow \underbrace{b3}_{\text{conclusion}} \quad \} \text{goal}$$

Progress in the proof happens via tactics that allows the *simplification*, *decomposition* into *subgoals*, or the *resolution* of the goal.

Continuing the proof

---

lemma mylemma b1 b2 b3 : ...
proof.
  move=> hb12.

---

b1 : bool
b2 : bool
b3 : bool
hb12 : b1 $\Rightarrow$ *b2*

(b2 $\Rightarrow$ b3) $\Rightarrow$ b1 $\Rightarrow$ b3

Continuing the proof

---

lemma mylemma b1 b2 b3 : ...
proof.
  move=> hb12 bh23 hb1.

---

b1 : bool
b2 : bool
b3 : bool
hb12 : b1 ⇒ b2
hb23 : b2 ⇒ b3
hb1 : b1

==========

b3

Continuing the proof

```
lemma mylemma b1 b2 b3 : ...
proof.
  move=> hb12 bh23 hb1.
  apply hb23.
```

b1 : bool
b2 : bool
b3 : bool
hb12 : b1 ⇒ b2
hb23 : b2 ⇒ b3
hb1 : b1

---

b2

Continuing the proof

---

lemma mylemma b1 b2 b3 : ...
proof.
  move=> hb12 bh23 hb1.
  apply hb23.
  apply hb12.

---

b1 : bool
b2 : bool
b3 : bool
hb12 : b1 $\Rightarrow$ b2
hb23 : b2 $\Rightarrow$ b3
hb1 : b1

---

b1

Continuing the proof

```
lemma mylemma b1 b2 b3 : ...
proof.
  move=> hb12 bh23 hb1.
  apply hb23.
  apply hb12.
  assumption.
```

Proof completed

Continuing the proof

```
lemma mylemma b1 b2 b3 : ...
proof.
  move=> hb12 bh23 hb1.
  apply hb23.
  apply hb12.
  assumption.
qed.
```

Topic 3

Core Tactics

Propositional logic

- $b1 \Rightarrow b2 \Rightarrow b3$

  As a goal [move⇒ b1 b2]

$$\frac{}{b1 \Rightarrow b2 \Rightarrow b3} \quad \hookrightarrow \quad \frac{\begin{array}{c} b1 : bool \\ b2 : bool \end{array}}{b3}$$

  As a hypothesis [apply]

Propositional logic

- $b1 \Rightarrow b2 \Rightarrow b3$

  As a goal [move$\Rightarrow$ b1 b2]

  As a hypothesis [apply]

$$\frac{h : b1 \Rightarrow b2 \Rightarrow b3}{b3} \hookrightarrow$$

1. $\dfrac{h : b1 \Rightarrow b2 \Rightarrow b3}{b1}$ 2. $\dfrac{h : b1 \Rightarrow b2 \Rightarrow b3}{b2}$

Propositional logic - connectors

- Conjunction: $a \wedge b$

As a goal [split] (prove $a \wedge b$)

$$\frac{}{a \wedge b} \hookrightarrow 1.\ \frac{}{a} \quad 2.\ \frac{}{b}$$

As a hypothesis [elim ab] (destruct $a \wedge b$ in a *and* b)

$$\frac{ab : a \wedge b}{\phi} \hookrightarrow \frac{}{a \Rightarrow b \Rightarrow \phi}$$

Propositional logic - connectors

- Disjunction: $a \vee b$

  As a goal

  - [left] (prove $a \vee b$ by proving a)

$$\frac{}{a \vee b} \hookrightarrow \frac{}{a}$$

  - [right] (prove $a \vee b$ by proving b)

$$\frac{}{a \vee b} \hookrightarrow \frac{}{b}$$

  As a hypothesis [elim ab] (case analysis on $a \vee b$)

$$\frac{ab : a \vee b}{\phi} \hookrightarrow 1. \ \frac{}{a \Rightarrow \phi} \quad 2. \ \frac{}{b \Rightarrow \phi}$$

Propositional logic - existential

- Existential: *exists* x : t, ϕ (x)

As a goal [*exists* v] (prove goal by giving a witness)

$$\frac{}{\textit{exists}\ \mathrm{x} : \mathrm{t}, \phi\,(\mathrm{x})} \hookrightarrow \frac{}{\phi\,(\mathrm{v})}$$

As a hypothesis [elim h] (extract a witness)

$$\frac{\mathrm{h} : \textit{exists}\ \mathrm{x} : \mathrm{t}, \phi\,(\mathrm{x})}{\phi\,'} \hookrightarrow \frac{}{\textit{forall}\ (\mathrm{v} : \mathrm{t}), \phi\,(\mathrm{v}) \Rightarrow \phi\,'}$$

Case analysis

The tactic case performs a case analysis on boolean or inductive expressions.

$$\frac{\begin{array}{l}a : bool\\ b : bool\end{array}}{a \oplus b = (a \wedge !b) \,||\, (!a \wedge b)}$$

(case a) leads to

1. $$\frac{\begin{array}{l}a : bool\\ b : bool\end{array}}{a \Rightarrow true \oplus b = (true \wedge !b) \vee (!true \wedge b)}$$

2. $$\frac{\begin{array}{l}a : bool\\ b : bool\end{array}}{!a \Rightarrow false \oplus b = (false \wedge !b) \vee (!false \wedge b)}$$

Identification up to computations

EasyCrypt comes with a set of simplification rules.

$$\frac{\begin{array}{l}a : \mathrm{bool}\\ b : \mathrm{bool}\end{array}}{\mathrm{false} \oplus b = (\mathrm{false} \wedge \mathrm{!}b) \vee (\mathrm{!false} \wedge b)}$$

[simplify] leads to

$$\frac{\begin{array}{l}a : \mathrm{bool}\\ b : \mathrm{bool}\end{array}}{b = b}$$

that can be easily solved by reflexivity.

Identification up to computations

Computations include (among other notions)

- reduction of function applications (β-reduction),
- inlining operator bodies (δ-reduction),
- logical operators tautology (a $\wedge$ false $\rightarrow$ false) ([logic]).

Terms that are equal up to computations are considered identical

$$\frac{\begin{array}{l}\text{a : bool}\\ \text{b : bool}\end{array}}{\text{!a} \Rightarrow \text{false} \oplus \text{b} = (\text{false} \wedge \text{!b}) \vee (\text{!false} \wedge \text{b})}$$

can be directly solved by reflexivity.

Rewrite - replace equals by equals

The tactic rewrite replaces a subterm a of the goal by an equal one b. It requires a proof of a = b or a $\Leftrightarrow$ b.

rewrite h

$$\frac{h : a = b}{P\ a} \hookrightarrow \frac{h : a = b}{P\ b}$$

Currently, one cannot rewrite under quantifiers, but one can [subst]itute variables. a is also substituted in the context.

subst a

$$\frac{h : a = b}{\phi} \hookrightarrow \frac{}{\phi\,[b/a]}$$

Rewrite - replace equals by equals

rewrite comes in different flavor:

- rewrite −h : from right to left
- rewrite mh where m is a multiplier

?	as many times as possible
!	as many times as possible, at least one
n?	at most n times
n!	exactly n times

- rewrite {o}h where o is a sequence of positive integers

  Rewrites the o^{th} occurrences only.
- rewrite /o where o is a defined operator's name

  Expands (or unfolds) the operator's definition. (rewrite −/o folds the definition back in.)

Rewrite - replace equals by equals

$$2 * (a + b) = (b + a) + (a + b)$$

- rewrite {2}addnC

$$2 * (a + b) = (b + a) + (b + a)$$

- rewrite (addnC b a)

$$2 * (a + b) = (a + b) + (a + b)$$

- rewrite -!addnA

$$2 * (a + b) = b + (a + (a + b))$$

Logical cut

The tactic cut: ϕ allows to do a forward chaining

$$\frac{\text{h: ...}}{\phi\text{'}} \hookrightarrow 1.\ \frac{\text{h: ...}}{\phi} \quad 2.\ \frac{\text{h: ...}}{\phi \Rightarrow \phi\text{'}}$$

It is possible to give a name to the new goal (cut my: ϕ)

$$\frac{\text{h: ...}}{\phi\text{'}} \hookrightarrow 1.\ \frac{\text{h: ...}}{\phi} \quad 2.\ \frac{\begin{array}{c}\text{h: ...}\\ \text{my:}\phi\end{array}}{\phi\text{'}}$$

Induction

An induction principle for a type t is any formula of the form:

$\forall$ (p : t $\rightarrow$ bool), $\phi_1 \rightarrow$ ... $\rightarrow \phi_n$, $\forall$ (x : t), ψ_1(x) $\rightarrow$ ... $\rightarrow \psi_n$(x) $\rightarrow$ p x

For example, for natural numbers:
forall (p : int $\rightarrow$ bool), p 0 $\Rightarrow$
(*forall* (x : int), 0 $\leq$ x $\Rightarrow$ p x $\Rightarrow$ p (x + 1)) $\Rightarrow$
forall (x : int), 0 $\leq$ x $\Rightarrow$ p x

Induction - Induction Principles

Induction principles are generated automatically for inductive datatypes.

type α *list* = [Nil | Cons of α & α *list*].

yields

forall (p : '*a* list $\rightarrow$ bool),
p Nil $\Rightarrow$
(*forall* (x : '*a*) (xs : '*a* list), p xs $\Rightarrow$ p (Cons x xs)) $\Rightarrow$
forall (xs : '*a* list), p xs

Induction

Applying the induction principle via apply can be cumbersome.

The tactic elim/ eases the applications of such principles.

$$\frac{\text{P: int} \to \text{bool}}{0 \leq x \Rightarrow P\ x} \quad \hookrightarrow \quad \text{elim/ind x}$$

1. $\dfrac{P : \text{int} \to \text{bool}}{P\ 0}$ 2. $\dfrac{P : \text{int} \to \text{bool}}{\forall\,(x : \text{int}),\ 0 \leq x \to P\ x \to P\ (x+1)}$

Induction

For inductive types, the induction principle is automatically selected by the elim tactic.

$$\frac{\begin{array}{l}\mathsf{P} : \alpha\ \mathit{list} \to \mathsf{bool} \\ \mathsf{x} : \alpha\ \mathit{list}\end{array}}{\mathsf{P\ x}}$$

$$\hookrightarrow\ \mathsf{elim\ x}$$

1. $$\frac{\mathsf{P} : \alpha\ \mathit{list} \to \mathsf{bool}}{\mathsf{P\ Nil}}$$

2. $$\frac{\mathsf{P} : \alpha\ \mathit{list} \to \mathsf{bool}}{\forall\,(\mathsf{x} : \alpha)\ (\mathit{xs} : \alpha\ \mathit{list}),\ \mathsf{P\ xs} \to \mathsf{P\ (Cons\ x\ xs)}}$$

Automation

EasyCrypt comes with some automation tactics:

- progress breaks the goal by repetead applications of the introduction tactics (split, move, ...) and elimination on introduced assumptions. Several variants:
  - progress [nosplit] does not split the conclusion.
  - trivial leaves the goal intact if not entirely discharged.
- smt: try to solve the goal using external SMT solvers.

Topic 4

Tacticals

Tacticals I

Tacticals are operators on tactics.

- t1; t2
  apply t1 and then t2 on all generated subgoals
- t; [t1|...|tn]
  apply t and then each of the t_i to the i^{th} subgoal
- do t
  repeat t as much as possible, at least one time

  this tactic takes the same multiplier of rewrite
  do! t, do? t, do n! e, do n? t
- try t
  try to apply t, or nothing if t cannot by applied
- by t1; ...; tn
  apply t1; ...; tn and then try to close all the subgoals via trivial. fail if all the subgoals cannot be solved.

Tacticals II

- t1; first t2

  apply t1 and then t2 on the first subgoal
- t1; last t2

  apply t1 and then t2 on the last subgoal
- variants: t1; first n t2, t1; last n t2
- t; first n last

  apply t and then shift the n first goals to the end

Tacticals - Intro Patterns

- t=> ip1 ... ipn

  apply t and then execute the introduction of ip1 ... ipn

  - t=> x

    introduce a name / a hypothesis
  - t=> [ip1|...ipn]

    do a case analysis on the top assumption and execute ip_i on the i^{th} subgoal
  - t=> −>, *or* t=> <−

    introduce an equational hypothesis and rewrite it
  - t=> {h}

    clear the hypothesis h
  - t=> //, *or* t=> /=, *or* t=> //=

    execute trivial, simplify, simplify; trivial
- move is the identity tactic.
- In cut ip:ϕ , ip is an arbitrary intro pattern.

Trade-off between interactive / automatic proof

- EasyCrypt has two kinds of tactics
  - low-level, interactive ones
  - the SMT hammer

  The difficulty is to find the right trade-off between the two.
  - SMT goal resolution success can be very unstable
  - SMT is very good at solving large numbers of small problems generated by the program logics

- qed does not mark the end of the proof.

Lecture 2

Formalizing Distributions

Distributions in EasyCrypt

- Discrete sub-distributions
- Described by their probability mass function:
  $\mu : \alpha\ \mathit{distr} \to (\alpha \to \mathit{bool}) \to$ real.
- In this lecture
  - Base axioms on distributions,
  - Some example definitions.

Properties of distributions

General Axioms

```
(* mu d p is always within the unit interval *)
axiom mu_bounded (d:'a distr) (p:'a -> bool):
  0%r <= mu d p <= 1%r.

(* The probability of the false event is 0 *)
axiom mu_false (d:'a distr): mu d (fun _, false) = 0%r.

(* Probability of a disjunction of events *)
axiom mu_or (d:'a distr) (p q:'a -> bool):
  mu d (cpOr p q) = mu d p + mu d q - mu d (cpAnd p q).

(* Point-wise equality is equality *)
axiom pw_eq (d d':'a distr):
  d == d' => d = d'.
```

An example definition

Example (Uniform distribution on booleans)

```
op bool_uf: bool distr.

axiom bool_uf_def (p: bool -> bool):
  mu bool_uf p = (1%r / 2%r) * charfun p true +
                 (1%r / 2%r) * charfun p false.
```

Derived Operators

Derived Operators

```
(* Probability of a particular element *)
op mu_x (d:'a distr) (x:'a): real = mu d ((=) x).

(* Support of a distribution *)
op support (d:'a distr) x: bool = 0%r < mu d x.

(* Point-wise equality *)
pred (==) (d1:'a distr) (d2:'a distr) = mu_x d1 == mu_x d2.
```

Some remarks

- Some lemmas on distributions can be used with rewrite Pr to rewrite probability expressions.

Example (rewrite Pr mu_or)

$$\Pr[\, f, m : A \vee B \,]$$
$$\downarrow$$
$$\Pr[\, f, m : A \,] + \Pr[\, f, m : B \,] - \Pr[\, f, m : A \wedge B \,]$$

- It is better, if possible, to define distributions using mu and prove simplification lemmas for mu_x and support.

Example

```
lemma support_uf_bool (b:bool): support uf_bool b.

lemma mu_x_uf_bool (b:bool): mu_x uf_bool b = 1%r / 2%r.

lemma lossless : mu uf_bool (fun _, true) = 1%r.
```

Distribution Transformers

- The standard library defines distribution transformers.
  - **op** ($*$): α *distr* $\rightarrow$ β *distr* $\rightarrow$ ($\alpha * \beta$) *distr*. (in Pair.ec)
  - **op** dapply: ($\alpha \rightarrow \beta$) $\rightarrow$ α *distr* $\rightarrow$ β *distr*. (in Distr.ec)
  - ...
- Very useful to describe complex distributions.

Summary

- A way of *axiomatically* defining discrete distributions.
- Very powerful rewriting results on probability expressions.

Lecture 3

Interactions between EasyCrypt Logics

Reminder: The EasyCrypt Logics

- Hoare Logic

$$[c : P \Longrightarrow Q]$$

- Probabilistic Hoare Logic

$$[c : P \Longrightarrow Q] \leq \delta \quad [c : P \Longrightarrow Q] = \delta \quad [c : P \Longrightarrow Q] \geq \delta$$

- Probabilistic Relational Hoare Logic

$$[c_1 \sim c_2 : P \Longrightarrow Q]$$

- Ambient higher-order logic

$$\begin{aligned} &\forall\, \mathrm{c}_1, \mathrm{c}_2, \mathrm{m}_1, \mathrm{m}_2. \\ &\quad [c_1 \sim c_2 : \mathrm{true} \Longrightarrow =\{\mathrm{res}\}] \Rightarrow \\ &\quad \Pr[c_1, m_1 : \mathrm{res}] = \Pr[c_2, m_2 : \mathrm{res}] \end{aligned}$$

byphoare, byequiv, bypr, conseq

In this lecture

- How to interpret HL, pHL and pRHL judgments as statements on probability expressions,
- How to interpret statements on probability expressions as HL, pHL or pRHL judgments,
- How to combine HL, pHL and pRHL judgments together to strengthen them.
- Warning: Whenever a pre or postcondition appears outside of aHL, pHL or pRHL judgment, it is parameterized by one or several memories. I omit this in the slides.

Proving probability expressions

- The formula

$$\begin{aligned}&\forall\, c_1, c_2, m_1, m_2.\\ &\quad [c_1 \sim c_2 : \text{true} \Longrightarrow =\{\text{res}\}] \Rightarrow\\ &\quad \Pr[c_1, m_1 : \text{res}] = \Pr[c_2, m_2 : \text{res}]\end{aligned}$$

  is a direct consequence of the definition for $[\cdot \sim \cdot : \cdot \Longrightarrow \cdot]$.
- Similar rules connect other fragments of the logic together.
- These rules can be applied in proofs using the tactics byequiv, byphoare, **hoare**, and bypr.

Interpreting equiv judgments

Equivalence: **equiv**[c1 ~ c2: P ==> Q]

As a hypothesis [byequiv (_: P $\Longrightarrow$ Q)]

$$\frac{}{\Pr[c1, m1: E1] = \Pr[c2, m2: E2]}$$

$\hookrightarrow$

1. $\dfrac{}{P\ m1\ m2}$ 2. $\dfrac{}{\textbf{equiv}[c1 \sim c2: P \Longrightarrow Q]}$

3. $\dfrac{}{\forall\, m1'\ m2', Q\ m1'\ m2' \Rightarrow (E1\ m1' \Leftrightarrow E2\ m2')}$

- $\Pr[c1, m1: E1] \leq \Pr[c2, m2: E2]$ (with E1 m1' $\Rightarrow$ E2 m2')
- $\Pr[c1, m1: E1] \geq \Pr[c2, m2: E2]$ (with E2 m2' $\Rightarrow$ E1 m1')

Interpreting equiv judgments

Equivalence: **equiv**[c1 ~ c2: P ==> Q]

As a goal bypr (X1) (X2) (prove by point-wise equality)

$$\frac{}{\textbf{equiv}[c1 \sim c2 : P \Longrightarrow Q]}$$

$\hookrightarrow$

1.
$$\frac{\begin{array}{l} m1 : \text{memory} \\ m2 : \text{memory} \\ H : P\ m1\ m2 \end{array}}{\forall X,\ \Pr[c1, m1 : X = X1] = \Pr[c2, m2 : X = X2]}$$

2.
$$\frac{\begin{array}{l} m1 : \text{memory} \\ m2 : \text{memory} \end{array}}{\forall X,\ X = X1\{m1\} \Rightarrow X = X2\{m2\} \Rightarrow Q\ m1\ m2}$$

Interpreting phoare judgments

Probabilistic Hoare Triple: **phoare**[c: P ==> Q] = d

As a hypothesis [byphoare (_: P $\Longrightarrow$ Q)]

$$\frac{}{\mathsf{Pr}[\mathsf{c}, \mathsf{m}: \mathsf{E}] = \mathsf{d}]}$$

$\hookrightarrow$

1. $\dfrac{}{\mathsf{P}\ \mathsf{m}}$ 2. $\dfrac{}{\mathbf{phoare}[\mathsf{c}: \mathsf{P} \Longrightarrow \mathsf{Q}] = \mathsf{d}}$

3. $\dfrac{}{\forall\, \mathsf{m'}, \mathsf{Q}\ \mathsf{m'} \Leftrightarrow \mathsf{E}\ \mathsf{m'}}$

- Pr[c, m: E] $\leq$ d (with **phoare**[c: P $\Longrightarrow$ Q] $\leq$ d and E m' $\Rightarrow$ Q m')
- Pr[c, m: E] $\geq$ d (with **phoare**[c: P $\Longrightarrow$ Q] $\geq$ d and E m' $\Rightarrow$ Q m')

Interpreting phoare judgments

Probabilistic Hoare Triple: **phoare**[c: P ==> Q] = d

As a goal bypr (prove by point mass)

$$\frac{}{\textbf{phoare}[\text{c: P ==> Q}] = \text{d}} \quad \hookrightarrow \quad \frac{\begin{array}{l}\text{m: memory}\\ \text{H: P m}\end{array}}{\text{Pr}[\text{c, m: Q}]\ \text{d}}$$

Intepreting hoare judgments

Hoare Triple: **hoare**[c: P ==> Q]

As a goal bypr (zero probability)

$$\frac{}{\textbf{hoare}[c: P \Longrightarrow Q]} \hookrightarrow \frac{\begin{array}{l}m: \text{memory} \\ H: P\ m\end{array}}{\mathrm{Pr}[c, m: !Q] = 0}$$

Strenghtening contracts

A simple rule of consequence: conseq (_: P $\Longrightarrow$ Q)

$$\frac{}{\textbf{hoare}[\text{c: P'} \Longrightarrow \text{Q'}]}$$

$\hookrightarrow$

$$1.\ \frac{}{\text{P} \Rightarrow \text{P'}} \qquad 2.\ \frac{}{\textbf{hoare}[\text{c: P} \Longrightarrow \text{Q}]} \qquad 3.\ \frac{}{\text{Q'} \Rightarrow \text{Q}}$$

Also works for pHL and pRHL judgments. In pHL, the bound can also be strengthened. Useful variants:

- conseq (_: _ $\Longrightarrow$ Q), weaken only the postcondition,
- conseq (_: P $\Longrightarrow$ Q), weaken only the precondition,
- conseq$*$ (_: P $\Longrightarrow$ Q), quantify only on variables that may be modified by the statement (frame rule).

Combining contracts: Hoare + Hoare

The conseq tactic can also be used to combine together several judgments.
[conseq (_: P' $\Longrightarrow$ Q') (_: P'' $\Longrightarrow$ *Q*'')]:

$$\frac{}{\textbf{hoare}[\text{c: P} \Longrightarrow \text{Q}]}$$

$\hookrightarrow$

1. $\dfrac{}{\text{P} \Rightarrow \text{P'} \wedge \text{P''}}$ 2. $\dfrac{}{\textbf{hoare}[\text{c: P'} \Longrightarrow \text{Q'}]}$

3. $\dfrac{}{\textbf{hoare}[\text{c: P''} \Longrightarrow \textit{Q}\text{''}]}$ 4. $\dfrac{}{\text{Q'} \wedge \text{Q''} \Rightarrow \textit{Q}}$

Combining contracts: pHoare + Hoare

[conseq (_: P' $\Longrightarrow$ Q': =d) (_: P'' $\Longrightarrow$ *Q*'')] (also with $\leq$, $\geq$):

$$\frac{}{\textbf{phoare}[c\text{: } P \Longrightarrow Q]\text{: } = d'}$$

$\hookrightarrow$

1. $\dfrac{}{P \Rightarrow P' \wedge P''}$ 2. $\dfrac{}{\textbf{phoare}[c\text{: } P' \Longrightarrow Q'] = d}$

3. $\dfrac{}{\textbf{hoare}[c\text{: } P'' \Longrightarrow Q'']}$ 4. $\dfrac{}{Q' \wedge Q'' \Rightarrow Q}$

Combining contracts: equiv + Hoare

[conseq (_: P' ⟹ Q') (_: P1 ⟹ Q1) (_: P2 ⟹ Q2)]:

$$\frac{}{\textbf{equiv}[c1 \sim c2 : P \Longrightarrow Q]}$$

↪

1.
$$\frac{\begin{array}{c}m1:\text{memory}\\ m2:\text{memory}\end{array}}{P\ m1\ m2 \Rightarrow P'\ m1\ m2\ /\ P1\ m1\ /\ P2\ m2}$$

2. $$\frac{}{\textbf{equiv}[c1 \sim c2 : P' \Longrightarrow Q']}$$

3. $$\frac{}{\textbf{hoare}[c1 : P1 \Longrightarrow Q1]}$$

4. $$\frac{}{\textbf{hoare}[c2 : P2 \Longrightarrow Q2]}$$

5.
$$\frac{\begin{array}{c}m1:\text{memory}\\ m2:\text{memory}\end{array}}{Q'\ m1\ m2 \wedge Q1\ m1 \wedge Q2\ m2 \Rightarrow Q\ m1\ m2}$$

Combining contracts: equiv + pHoare

[conseq (_: P1 ⟹ Q1) (_: P2 ⟹ Q2: =1%r)]:

$$\frac{}{\textbf{equiv}[\text{c1} \sim \text{skip}: P \Longrightarrow Q]}$$

↪

1. $$\frac{\begin{array}{c}\text{m1:memory}\\ \text{m2:memory}\end{array}}{P\ m1\ m2 \Rightarrow P1\ m1\ m2}$$

2. $$\frac{\begin{array}{c}\text{m1:memory}\\ \text{m2:memory}\end{array}}{Q1\ m1\ m2 \Rightarrow Q\ m1\ m2}$$

3. $$\frac{\begin{array}{c}\text{m1:memory}\\ \text{m2:memory}\end{array}}{P1\ m1\ m2 \Rightarrow P2\ m2}$$

4. $$\frac{\begin{array}{c}\text{m1:memory}\\ \text{m2:memory}\end{array}}{Q2\ m2 \Rightarrow Q1\ m1\ m2}$$

5. $$\frac{}{\textbf{phoare}[\text{c1}: P2 \Longrightarrow Q2]}$$

On the notation (_: ...)

- A tactic parameter of the form (_: ...) can be replaced with a proof term whose conclusion have the correct form.
- For example, if you have the following lemma

  lemma c_ll (b':bool): **phoare**[c: b = b' $\Longrightarrow$ true] = 1%r,

  you can write

  conseq (c_ll true)

  instead of

  conseq (_: b = true $\Longrightarrow$ true); first by apply (c_ll true).

A few words on modules

- All the judgments and tactics we discussed talk about programs,
- But we've said nothing about what those programs are...
- They can be *procedures* in modules, or just statements.
- Modules are objects that define a *memory space* and a set of *procedures* that may use variables in that space.
- They can be abstracted by *module types* that define a set of required procedures.

Conclusions

- A lot can be proved about programs without looking at their body.
- Combining judgments can save a lot of time and effort:
  - Real-world programs have complex Hoare invariants.
  - Real-world cryptographic proofs have complex relational invariants
  - It is *almost always* beneficial to consider them separately when doing proofs, until they are both needed.
- It is almost as easy to reason about probabilities of events in programs as it is to reason about probabilities in functional distributions.
- In fact, programs define distributions!

Lecture 4

EasyCrypt Modules

So far, we've seen

- how to specify and prove properties of mathematical operators;
- how to specify and prove properties of (discrete sub-)distributions;
- how to force the interpretation of procedures as distribution transformers (and vice-versa);
- how to combine and strengthen judgments about abstract procedures.

But we still don't know how to write a procedure...

Modules

Objectives:

- Manage complexity by abstraction
- Instantiate 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] + ...

- Support high-level reasoning steps

Central to EasyCrypt:

- Used to specify schemes, oracles, cryptographic assumptions, adversaries and game-based properties.

Contents 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

My first module

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

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

A module RO with a global variable m and a procedure h. Outside the module, the global variable is denoted RO.m and the procedure RO.f

Modules can call and use the memory of others

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

module M2 = {
  proc 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 *)
  proc choose (pk:pkey) : msg * msg          (* procedure declarations *)
  proc guess (c:cipher) : bool
}.
```

Remarks:

- 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) = {
  proc 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 *)
```

Module types are checked when applying functors.

The memory model

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

module M2 = M1.

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

Question: After the execution of T.f what is the value of M2.x and M1.x in?

Functor application is not generative

A module should be understood as a pair of:

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

Remarks:

- Functor application does not generate a new memory space.
- Global variables of a functor can be read or written without applying the functor: `F.x = F.x + 1;`
- Procedures cannot be called until the functor is fully applied.

Back to the CPA game

```
module CPA (A:Adv) = {
  proc 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 the adversary's capabilities 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 capture these notions

A first try, declaration of the IND-CCA adversary

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

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

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

the notation

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

is a shortcut for

```
proc 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 = {
  proc dec_c(c:cipher) : msg
  proc dec_g(c:cipher) : msg
}.
module type ADVCCA(D:DEC2) = {
  proc choose (pk:pkey) : msg * msg    { 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 “reject” queries on the challenge.

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

  proc dec_c (c:cipher) : msg option = {
    var r : msg option;
    r = S.dec(sk, c);
    return r;
  }
  proc 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 : ... ==> ...]
```

Can express formulas like:

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

Memory constraints and access control

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

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

lemma F (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”.

Negative constraints

EasyCrypt allows quantifications over restricted classes of modules using negative constraints:

---

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

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

for all “adversary” A whose implementation does 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

- Modules are used for everything stateful.
- Functors are used to specify and prove systems in a compositional way.
- One module, one memory space: there are never two copies of a global.
- The full importance of restricting the memory of quantified modules will become clear when we discuss tactics.

Lecture 5

Proving and Transforming Programs

Tactics for the Program Logics

Three kinds:

- Tactics that operate on the contracts (Lecture 3),
- Tactics that deduce properties of the program from their semantics,
- Tactics that transform programs into equivalent ones.

An overview of the proof rules for the program logics was given in week 1... (if you remember that far)

Overview

- Consider a HL, pHL or pRHL judgment.
- Program logics turn it into a purely logical formula.
- Follow the program's structure from the bottom-up.
- Use the language semantics to transform the pre and post according to the statement that is being dealt with.
- The tactics themselves implement the core proof rules from Week 1 composed with the sequential composition and conseq rule.

Starting a proof about a concrete procedure: [proc]

Assume that M.f has the following definition.
module M = { proc f(): t = { var r; c; **return** r; } }.

$$\textbf{hoare}[\text{M.f: P} \Longrightarrow \text{Q}] \hookrightarrow \textbf{hoare}[\text{c: P} \Longrightarrow \text{Q[r/res]}]$$

$$\textbf{hoare}[\text{M.f: P} \Longrightarrow \text{Q}]$$
$$\hookrightarrow$$
$$[\text{proc} *] \ \textbf{hoare}[\text{r = M.f(): P} \Longrightarrow \text{Q[r/res]}]$$

Similar rules for other types of judgments.

Dealing with the empty program: [skip]

$$\textbf{hoare}[\text{skip: } P \Longrightarrow Q] \hookrightarrow \forall\, \&m,\ P\ m \Rightarrow Q\ m$$

$$\textbf{phoare}[\text{skip: } P \Longrightarrow Q] = 1\%r \hookrightarrow \forall\, \&m,\ P\ m \Rightarrow Q\ m$$

$$\textbf{equiv}[\text{skip} \sim \text{skip: } P \Longrightarrow Q]$$

$$\hookrightarrow$$

$$\forall\, \&m1\ \&m2,\ P\ m1\ m2 \Rightarrow Q\ m1\ m2$$

- The **phoare** version can also be used to prove other statements, but you will *never* be able to prove **phoare**[skip: $P \Longrightarrow Q$] = d (or $\leq$ d) when d < 1.

Straight-line deterministic code: [wp] and [sp]

- [wp] consumes uninterrupted sequences of deterministic assignments at the end of a program and transform the postcondition.
- [sp] consumes uninterrupted sequences of deterministic assignments at the beginning of a program and transform the precondition.
- Variants:
  - [wp n] and [sp n] (HL and pHL): Consume up to (sp: down to) line number n.
  - [wp n n'] and [sp n n'] (pRHL): Consume up to (sp: down to) line number n in the left program, and line number n' in the right program.

Sequential Composition: [seq]

As expected for HL, pRHL and lower-bound pHL:

$$\textbf{hoare}[c; c' : P \Longrightarrow Q] \hookrightarrow [\text{seq } n : (\phi)] \text{ with } n = |c|$$

1. $\textbf{hoare}[c : P \Longrightarrow \phi]$ 2. $\textbf{hoare}[c' : \phi \Longrightarrow Q]$

$$\textbf{equiv}[c1; c2 \sim c1'; c2' : P \Longrightarrow Q] \hookrightarrow [\text{seq } n\ n' : (\phi)] \text{ with } n = |c1|, n' = |c1'|$$

1. $\textbf{equiv}[c1 \sim c2 : P \Longrightarrow \phi]$ 2. $\textbf{hoare}[c1' \sim c2' : \phi \Longrightarrow Q]$

$$\textbf{phoare}[c; c' : P \Longrightarrow Q] \geq d \hookrightarrow [\text{seq } n : (\phi)\ d1\ d2] \text{ with } n = |c|$$

1. $\textbf{phoare}[c : P \Longrightarrow \phi] \geq d1$ 2. $\textbf{hoare}[c' : \phi \Longrightarrow Q] \geq d2$
3. $d1 * d2 \geq d$

Sequential Composition for pHL

Probabilistic intermediate invariants cause issues:

phoare[c; c': P $\Longrightarrow$ Q] = d

$\hookrightarrow$ [seq n: (ϕ) d1 d2 d3 d4 (ψ)] with n = |c|

1. **hoare**[c: P $\Longrightarrow \psi$]
2. **phoare**[c: P $\Longrightarrow \phi$] = d1
3. **phoare**[c': $\phi \wedge \psi \Longrightarrow$ Q] = d2
4. **phoare**[c: P $\Longrightarrow$!ϕ] = d3
5. **phoare**[c': !$\phi \wedge \psi \Longrightarrow$ Q] = d4
6. d1 $*$ d2 + d3 $*$ d4 = d

- ψ is a *deterministic* invariant: we want it to hold whenever we reach the end of c.
- ϕ is a *probabilistic* invariant: it may or may not hold when we reach the end of c depending on random samplings.
- Some simplifications possible:
  - if ψ is true, it can be omitted,
  - if one of d1, d2 (or d3, d4) is 0, the other can be _.
- Note the hidden application of conseq: makes the tactic more applicable than the proof rule from Week 1.

Case analysis on program variables: [case]

Introduces a case analysis on some expression of the program variables in the precondition.

$$\textbf{hoare}[c\colon P \Longrightarrow Q] \hookrightarrow [\text{case } (\phi)]$$

1. $\textbf{hoare}[c\colon P \wedge \phi \Longrightarrow Q]$ 2. $\textbf{hoare}[c\colon P \wedge \not\phi \Longrightarrow Q]$

$$\textbf{phoare}[c\colon P \Longrightarrow Q] = d \hookrightarrow [\text{case } (\phi)]$$

1. $\textbf{phoare}[c\colon P \wedge \phi \Longrightarrow Q] = d$ 2. $\textbf{phoare}[c\colon P \wedge \not\phi \Longrightarrow Q] = d$

$$\textbf{equiv}[c \sim c'\colon P \Longrightarrow Q] \hookrightarrow [\text{case } (\phi)]$$

1. $\textbf{equiv}[c \sim c'\colon P \wedge \phi \Longrightarrow Q]$ 2. $\textbf{hoare}[c' \sim c'\colon P \wedge \not\phi \Longrightarrow Q]$

One-sided pRHL tactics

- Almost all pRHL tactics from here on have one-sided variants.
- Allow relational proofs on programs that have different control-flow structures.
- The one-sided tactics are built by composition of pRHL [seq], the pRHL + pHL version of conseq, and the corresponding pHL tactic.
- To apply the one-sided variant of a tactic, use {1} or {2} immediately after the tactic's name.

Random assignments: [rnd]

hoare[c; x $\overset{\$}{\leftarrow}$ dx: P $\Longrightarrow$ Q]

$\hookrightarrow$ [rnd]

hoare[c: P $\Longrightarrow$ $\forall$ v, support dx v $\Rightarrow$ Q]

(Non-Deterministic interpretation of random sampling.)

---

phoare[c; x $\overset{\$}{\leftarrow}$ dx: P $\Longrightarrow$ Q] = d

$\hookrightarrow$ [rnd (ϕ)]

phoare[c: P $\Longrightarrow$ μ dx True = d $\wedge$

$\forall$ v, support dx v $\Rightarrow$ ϕ v $\Leftrightarrow$ Q[v/x]] = 1%r

(If ϕ is omitted, the postcondition Q is used.)

---

equiv[c; x $\overset{\$}{\leftarrow}$ dx ~ c'; y $\overset{\$}{\leftarrow}$ dx': P $\Longrightarrow$ Q]

$\hookrightarrow$ [rnd (f) (f')]

"f and f' are inverse bijections and preserve weights + hoare-style non-deterministic post-condition"

Conditionals: [if]

Do a case analysis on a conditional test *at the beginning* of the program. pRHL variant is synchronized.

hoare[**if** (e) { c1 } **else** { c2 }; c: P $\Longrightarrow$ Q]

$\hookrightarrow$ [**if**]

1. **hoare**[c1; c: P $\wedge$ e $\Longrightarrow$ Q]
2. **hoare**[c2; c: P $\wedge$!e $\Longrightarrow$ Q]

---

phoare[**if** (e) { c1 } **else** { c2 }; c: P $\Longrightarrow$ Q] = d

$\hookrightarrow$ [**if**]

1. **phoare**[c1; c: P $\wedge$ e $\Longrightarrow$ Q] = d
2. **phoare**[c2; c: P $\wedge$!e $\Longrightarrow$ Q] = d

---

equiv[**if** (e) { c1 } **else** { c2 }; c ~ **if** (e') { c1' } **else** { c2' }; c': P $\Longrightarrow$ Q]

$\hookrightarrow$ [**if**]

1. $\forall$ m m', P m m' $\Rightarrow$ e{m} $\Leftrightarrow$ e'{m'}
2. **equiv**[c1; c ~ c1'; c': P $\wedge$ e $\Longrightarrow$ Q]
3. **equiv**[c2; c ~ c2'; c': P $\wedge$!e $\Longrightarrow$ Q]

Desynchronized Conditionals: [if{.}]

equiv[**if** (e) { c1 } **else** { c2 }; c ~ c': P $\Longrightarrow$ Q]

$\hookrightarrow$ [**if**{1}]

1. **equiv**[c1; c ~ c': P $\wedge$ e $\Longrightarrow$ Q]
2. **equiv**[c2; c ~ c': P $\wedge$!e $\Longrightarrow$ Q]

---

equiv[c ~ **if** (e') { c1' } **else** { c2' }; c': P $\Longrightarrow$ Q]

$\hookrightarrow$ [**if**{2}]

1. **equiv**[c ~ c1'; c': P $\wedge$ e' $\Longrightarrow$ Q]
2. **equiv**[c ~ c2'; c': P $\wedge$!e' $\Longrightarrow$ Q]

Procedure calls: [call]

hoare[c; x = M.f(y): P $\Longrightarrow$ Q
$\hookrightarrow$ [call (_: P' $\Longrightarrow$ Q')]
1. **hoare**[M.f: P' $\Longrightarrow$ Q']
2. **hoare**[c: P $\Longrightarrow$ P' $\wedge$ ($\forall$ v, Q'[v/x] $\Rightarrow$ Q[v/x])]

(The "quantification on v" introduces fresh values for every program variable x that may be modified by M.f.)

---

phoare[c; x = M.f(y): P $\Longrightarrow$ Q] = d
$\hookrightarrow$ [call (_: P' $\Longrightarrow$ Q')]
1. **phoare**[M.f: P' $\Longrightarrow$ Q'] = d
2. **phoare**[c: P $\Longrightarrow$ P' $\wedge$ ($\forall$ v, Q'[v/x] $\Rightarrow$ Q[v/x])]

(Use [seq] if you need to adjust the probability use for M.f.)

---

equiv[c; x = M.f(y) ~ c'; x' = M'.f(y'): P $\Longrightarrow$ Q
$\hookrightarrow$ [call (_: P' $\Longrightarrow$ Q')]
1. **equiv**[M.f ~ M'.f: P' $\Longrightarrow$ Q']
2. **equiv**[c ~ c': P $\Longrightarrow$ P' $\wedge$ ($\forall$ v, Q'[v/x] $\Rightarrow$ Q[v/x])]

(A useful variant to try out call (_:ϕ).)

Loops: [while]

pRHL loops must be synchronized (or the one-sided rule must be used...)

$$\textbf{hoare}[c1; \textbf{while}\ (e)\ \{\ c2\ \}: P \Longrightarrow Q] \hookrightarrow [\textbf{while}\ (\phi)]$$

1. $\textbf{hoare}[c2: \phi \wedge e \Longrightarrow \phi]$
2. $\textbf{hoare}[c1: P \Longrightarrow \phi \wedge (\forall v, \phi[v/x] \Rightarrow Q[v/x])]$

$$\textbf{equiv}[c1; \textbf{while}\ (e)\ \{\ c2\ \} \sim c1'\ \textbf{while}\ (e')\ \{\ c2'\ \}: P \Longrightarrow Q] \hookrightarrow [\textbf{while}\ (\phi)]$$

1. $\textbf{equiv}[c2 \sim c2': \phi \wedge e \wedge e' \Longrightarrow \phi \wedge e \Leftrightarrow e']$
2. $\textbf{equiv}[c1 \sim c1': P \Longrightarrow \phi \wedge (\forall v, !e[v/x] \Rightarrow !e'[v/x] \Rightarrow \phi[v/x] \Rightarrow Q[v/x]]$

pHL Loops: [while]

Loops in pHL are more complicated.

$$\textbf{phoare}[\text{c1}; \textbf{while}\ (\text{e})\ \{\ \text{c2}\ \}: \text{P} \Longrightarrow \text{Q}] = \text{d}$$
$$\hookrightarrow [\textbf{while}\ (\phi)\ (\text{va})]$$

1. $\forall\, \text{z},\ \textbf{phoare}[\text{c2}: \phi \wedge \text{e} \wedge \text{va} = \text{z} \Longrightarrow \phi \wedge \text{va} < \text{z}] = 1\%\text{r}$
2. $\textbf{phoare}[\text{c1}: \text{P} \Longrightarrow \phi \wedge (\forall\, \text{v},$
$(\phi\,[\text{v/x}] \Rightarrow \text{e}[\text{v/x}] \Rightarrow \text{va}[\text{v/x}] \leq 0 \Rightarrow !\text{e}[\text{v/x}]) \wedge$
$(!\text{e}[\text{v/x}] \Rightarrow \phi\,[\text{v/x}] \Rightarrow \text{Q}[\text{v/x}]] = \text{d}$

- The rule works with $\leq$ and $\geq$ as well.
- There are rules for while loops where termination does not matter ($\leq$) or is not variant-based ("sample until"). They are discussed in the final lecture (if we make it).

Dealing with abstract procedures

- We need rules to perform proofs on universally quantified modules
- The rules should be valid independently of the *implementation* of the module
- In the following, assume that A.f is abstract and may query an oracle o.

hoare[A.f: $\phi \Longrightarrow \phi$
$\hookrightarrow$ [proc (ϕ)]
1. "ϕ does not depend on the memory of A"
2. **hoare**[o: $\phi \Longrightarrow \phi$]

---

equiv[A.f ~ A.f: ={arg} $\wedge \phi \Longrightarrow$ ={res} $\wedge \phi$]
$\hookrightarrow$ [proc (ϕ)]
1. "ϕ does not depend on the memory of A"
2. rges**equiv**[o ~ o: =a $\wedge \phi \Longrightarrow$ =r $\wedge \phi$]

Fundamental Lemma

A very frequent step in cryptographic proof

$$\Pr[G: E] \leq \Pr[G': E] + \Pr[G': F]$$

This can be established using the following pRHL lemma (see tutorial on conseq)

$$\textbf{equiv}[G.f \sim G'.f: !F\{2\} \Longrightarrow E\{1\} = E\{2\}]$$

How do we prove this judgment?

Upto failure abstract procedure call

Several conditions on the oracles:

- Assuming the failure event has not occured, prove the invariant;
- Prove that the failure event is monotonic;
- Since execution is no longer synchronized after the failure event, need to prove losslessness of everyone involved.

equiv[A.f ~ A.f: ={arg} $\wedge \phi \wedge$!bad{2} $\Longrightarrow$!bad{2} $\Rightarrow$ ={res} $\wedge \phi$]
$\hookrightarrow$ [proc (bad) (ϕ)]

1. "bad, ϕ do not depend on the memory of A"
2. **equiv**[o ~ o: ={arg} $\wedge \phi \wedge$!bad{2} $\Longrightarrow$!bad{2} $\Rightarrow$ ={res} $\wedge \phi$]
3. **equiv**[o ~ o: !bad{2} $\Longrightarrow$!bad{2}]
4. "losslessness conditions"

A more precise version takes a second invariant that should hold after bad becomes true: [proc (bad) (ϕ) (ϕ ')].

Transforming Programs

- We've seen tactics that operate on the pre and postconditions,
- and tactics that consume a program to prove properties about it.
- We can also transform a program into an equivalent one.

Inlining a procedure call: [inline]

- inline M.f inlines the code of M.f, with some additional assignments to avoid overwriting of local variables.
- inline{1} M.f inlines only in the left program.
- inline $*$ inline all concrete procedures iteratively until no more inlining is possible.

Swapping statements: [swap]

- swap n off move statement at line n to line ‘n + off’ (if offset is negative, it is moved up)
- swap{1} n off, swap{2} n off...
- swap [n..m] off move the block between lines n and m and move it by offset off.
- The swapped statements need to be independent from all the statements they cross.

Reducing conditional: [rcondt, rcondf]

- rcondt n asks to prove that the condition at line n is true and inlines the appropriate branch.
- rcondf n does the same but with the else branch
- a side needs to be specified in pRHL
- can be used to unroll the first iteration of a loop, or make a loop disappear if its condition is initially false!
- Note that the **if** tactic is a composition of case and rcond.

Advanced loop tactics: [splitwhile,fusion,fission]

Warning: Syntax is ugly, broken and unstable.

- splitwhile (cond): {side} n: splits the loop at line n using additional condition cond.
- fusion can merge two loops with the same indices into a single loop
- fission can split a loop into two distinct loops if the loop body can be split into independent chunks
- We won't be using these, but it's good to know they exist

Conclusions

- Which tactic to apply can usually be decided by the last statement in the program(s).
- When dealing with an abstract procedure, the invariants and the proof obligations should be about the oracles that procedure is allowed to query.
- Don’t forget about one-sided tactic variants in pRHL.
- In pRHL, losslessness conditions arise whenever programs are out of sync. They are annoying, but easy to deal with.

Automation

- In pRHL, when proving simple properties of programs that are very similar, try using [sim].
- In all program logics, the [auto] tactic may also become useful.

Lecture 6

Structuring Proofs

Instantiation

- Concrete schemes and abstract adversaries
- Reuse existing proofs when realizing cryptographic assumptions? (e.g., one-way trapdoor with RSA)
- Sections hide proof artifacts from final theorems, automatically infer restrictions on adversaries, and generalize theorems with module quantification.
- Cloning avoids user-level code duplication by instantiating abstract types and operators with concrete values, creating module copies with disjoint memories.

Sections

```
section.
  declare module Adv : A{Prop,Hyp}.

  local module G1 = {
    var count:int (* some state *)
    fun main() = { (* uses A *) }
  }.

  local module Dist : D = { (* uses A *) }

  local equiv Prop_G1:
    [Prop(A).main ~ G1.main: true ==> ={res}].

  local equiv G1_Hyp:
    [G1.main ~ Hyp(D).main: true ==> ={res}].

  lemma &m final: exists (Dist<:D),
    Pr[Prop(A).main() @ m: res] = Pr[Hyp(D).main() @ m: res].
end section.
```

Sections

- Inside the section, declared modules are independent from modules defined (declared) after it.
- **print axiom** final.

  yields (after the section is closed)

  ```
  lemma final (Adv:>A{Prop,Hyp}) &m:
    exists (Dist:>D),
      Pr[Prop(A).main() @ &m: res] = Pr[Hyp(D).main() @ &m:res].
  ```

- Declared modules become parameters to lemmas.
- Local lemmas disappear.
- Local modules disappear and restrictions can be dropped.

Usages of Sections

- Simplify proofs by inferring adversary restrictions and quantifications.

```
module G(Adv:A) = { ... }.

lemma foo (Adv<:A{G}):
  equiv [ Pr[G(A).f() ~ ... ].
```

```
section.
  declare Adv<:A.
  module G = { ... }.

  lemma foo:
    equiv [ Pr[G.f() ~ ...].
end section.
```

- Generalize theorem statements by hiding proof artifacts.
  - Adversary restrictions,
  - Intermediate games,
  - Intermediate equivs, lemmas and proofs.

Theories: Generalities

Theories provide an additional layer of generalization:

- declared abstract types yield “polymorphism”,
- declared constants and operators yield “universal quantifications”,
- in forms that EasyCrypt cannot reason about...
- ... but that allow efficient code reuse.

Theories: A simple example

```
theory Monoid.
  type t.

  op one: t.
  op ( * ): t -> t -> t.

  axiom mul1m (x : t): one * x = x.
  axiom mulm1 (x : t): x * one = x.
  axiom mulmA (x y z : t): (x * y) * z = x * (y * z).
end Monoid.
```

Cloning: A simple example

```
require import Int.
clone Monoid as MInt with
  type t <- int,
  op one = 1,
  op ( * ) <- ( * )
  proof * by smt.

print theory MInt. (* yields *)

theory MInt.
  op one: int = 1.

  lemma mul1m (x : int): one * x = x.
  lemma mulm1 (x : int): x * one = x.
  lemma mulmA (x y z : int): (x * y) * z = x * (y * z).
end MInt.
```

Theories: Cloning and Realization

When cloning, you can:

- define (=) or override (<−)
  - abstract types,
  - abstract operators (and constants),
- define *abstract* sub-theories,
  - All declared types and operators are abstract,
  - the theory contains only axioms (no lemmas).
- discharge some (or all) axioms
  - by giving a single proof for all axioms (usually smt),
  - or by giving individual proofs.

Theories: Cloning modules

- You can clone theories that contain modules.
- You get an exact copy of the module that works in a separate memory space.
- This is useful for code reuse and may have unforeseen applications in proofs.

Cloning: An example

```
theory ROM.
  module RO = {
    var m: (word,word) map

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

    fun h(x:word): word = {
      if (!in_dom x m) m.[x] =
        $dword;
      return m.[x];
    }
  }.
end ROM.
```

```
theory ROM'.
  module RO = {
    var m: (word,word) map

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

    fun h(x:word): word = {
      if (!in_dom x m) m.[x] =
        $dword;
      return m.[x];
    }
  }.
end ROM'.
```

Or just use **clone** ROM **as** ROM'.

Cloning: Some notes

- Cloning a theory that declares abstract types creates new, distinct abstract types unless you define or override them.
- Cloning is not especially suited to equipping existing theories with new algebraic structures.
- When used carelessly, cloning can cause SMT to give up.

Summary

Several ways to generalize proofs and theorems:

- Automatically infer module dependencies and adversary restrictions using sections.
- Abstract away the proof and its artifacts and keep only the relevant theorems and hypotheses using local modules and lemmas.
- Perform crypto proofs on abstract modules and operators before instantiating them using theories and cloning.

Lecture 7

Advanced Tactics