EasyCrypt - a (brief) tutorial

Vitor Pereira

FMiTF Bootcamp - May 29 - June 2, 2023

1 Introduction

This document serves as support to the *EasyCrypt- Hashed ElGamal semantic security proof exercise*, containing a brief EasyCrypt tutorial, describing the most used proof tactics and also providing an explanation of the EasyCrypt specification language.

The student is encouraged to follow the default EasyCrypt manual available at `https://github.com/EasyCrypt/easycrypt-doc` for a more complete description on the tool.

2 Fuctional types, operators and data structures

EasyCrypt's expression language is a higher-order strongly typed functional language. Following a syntax close to that of ML languages such as OCaml of F⋆, EasyCrypt allows a user to define its own data types and operators as mathematical functions.

2.1 Types

EasyCrypt natively supports the following basic types:

- `unit` - the (empty) *void* type
- `bool` - the boolean type
- `int` - the set of integers
- `real` - the set of real

New types can be defined according to the following syntax, where we specify a new type that captures tuples of integers and booleans.

```
type int_bool_tuple = int * bool.
```

Types can also be left under specified, i.e., without an actual realization of the type. For example, in the following EasyCrypt code, a new type `t` is defined, without provide a concrete value to it.

```
type t.
```

When a type is left under specified, it is common practice to call it an *abstract type*.

2.2 Operators

Operators in EasyCrypt are defined using the `op` keyword, followed by the operator name, arguments, return type and, finally, the operator body. The `fst` and `snd` operators - that extract the first and second elements of the `int_bool_type` type, respectively - can be specified as follows.

```
op fst (x : int_bool_tuple) : int = let (a, b) = x in a.
op snd (x : int_bool_tuple) : bool = let (a, b) = x in b.
```

Operators can also be defined for EasyCrypt native types. For example, the script

```
op double (x : int) : int = x * 2.
```

specifies a function that doubles the value of a given integer.

The operators described above can be combined with the goal of defining a function that, taking as input a tuple of integers and booleans (the `int_bool_tuple` type), returns a new element of the `int_bool_tuple` type, where the integer value has been doubled.

```
op double_int (x : int_bool_tuple) : int_bool_tuple =
  let k = fst x in
  let b = snd b in
  let dk = double k in
  (dk, b).
```

Finally, operators can also be left abstract. For example, suppose one wants to formalize a finite field library in EasyCrypt. First, one would need to specify an (abstract) type to capture elements of the finite field and then define a series of abstract operators that would specify arithmetic operations.

```
op q : int. (* The order of field is a prime q *)

type t. (* Type of elements of the field *)

op fzero : t. (* Zero element *)
op fone : t. (* One element *)
op ( * ) : t -> t -> t. (* Multiplication modulo q *)
op ( + ) : t -> t -> t. (* Addition modulo q *)
op [ - ] : t -> t. (* Additive inverse modulo q *)
```

In the above EasyCrypt script also introduces the syntax to specify infix and prefix operators: operators defined between parentheses - like `op ( * )` and `op ( + )` - are infix operators whereas operators defined between brackets - like `[ - ]` - are prefix operators.

2.3 Data structures

EasyCrypt also supports inductive data structures, commonly found in many functional languages. Natively, it supports inductive lists that may be the empty list `[]`, or a value `x::xs` constructed inductively by prepending `x` to the list xs.

```
type 'a list = [
  | "[]"
  | (::) of 'a & 'a list
].
```

Pattern-matching over an inductive data type is performed using the `with` syntax, as follows.

```
op size (l : 'a list) : int =
  with l = [] => 0
  with l = x :: xs => 1 + size xs.
```

Another example of an inductive data type that is widely used is that of binary trees, comprised of *nodes* and *leafs*. A *node* carries a data item and has a left and right *subtree*. A *leaf* is empty. In EasyCrypt, one can write the binary tree data type as follows.

```
type 'a tree =
  | Leaf
  | Node of 'a & 'a tree & 'a tree.
```

The depth of a tree is given by the total number of edges from the root node to the target node. In EasyCrypt, the depth of a tree can be calculated as showed next, where we assume the existence of the function `max` that returns the maximum of two integers.

```
op depth (t : 'a tree) : int =
  with t = Leaf => 0
  with t = Node x l r => 1 + max (depth l) (depth r).
```

3 Ambient logic

EasyCrypt's is EasyCrypt's foundational proof engine. It allows users to write axioms or prove lemmas about existing or newly defined types and operators. These can be either universally or existentially quantified.

Before providing examples of EasyCrypt axioms and lemmas, we first revise some useful proof tactics.

3.1 Commonly used EasyCrypt tactics

`move => `$\phi_1 \ldots \phi_n$ Moves assumptions from the conclusion into the context.

Example: applying `move => x y H` to

```
------------------------------------------------------------------------
forall (x y : int), x = y => y = x
```

is transformed into

```
x: int
y: int
H: x = y
------------------------------------------------------------------------
y = x
```

`move : `$\phi_1 \ldots \phi_n$ Moves assumptions from the context into the conclusion.

Example: applying `move : x y H` to

```
x: int
y: int
H: x = y
------------------------------------------------------------------------
y = x
```

is transformed into

```
------------------------------------------------------------------------
forall (x y : int), x = y => y = x
```

`split` Break a goal whose conclusion is intrinsically conjunctive into goals whose conclusions are its conjunction.

Example: applying `split` to

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
P /\ Q
```

is transformed into the following subgoals

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
P
```

and

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
Q
```

`right` Reduce a goal whose conclusion is a disjunction to one whose conclusion is its right member.
Example: applying `split` to

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
P \/ Q
```

is transformed into

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
Q
```

`left` Reduce a goal whose conclusion is a disjunction to one whose conclusion is its left member.
Example: applying `split` to

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
P \/ Q
```

is transformed into

```
P: bool
Q: bool
Hp: P
Hq: Q
------------------------------------------------------------------------
P
```

`congr` Replace a goal whose conclusion has the form $fp_1 \dots p_n = fq_1 \dots q_n$, where `f` is an assumption identifier or operator, with subgoals having conclusions $p_1 = q_1, \dots, p_n = q_n$.
Example: applying `congr` to

```
x: int
y: int
a: int
b: int
------------------------------------------------------------------------
f x y = f a b
```

is transformed into the following subgoals

```
x: int
y: int
a: int
b: int
------------------------------------------------------------------------
x = a
```

and

```
x: int
y: int
a: int
b: int
------------------------------------------------------------------------
y = b
```

`trivial` Try to solve the goal by using a mixture of low-level tactics.
Example: applying `trivial` to

```
------------------------------------------------------------------
forall (x y : int), x = y => y - 1 = x - 1
```

solves the goal.

`progress` Break the goal into multiple simpler ones by repeatedly applying `move =>`, `split` and `subst`.
Example: applying `progress` to

```
------------------------------------------------------------------
forall (x : 'a) (l : 'a list), x \in l => 1 <= size l
```

is transformed into

```
x: 'a
l: 'a list
H: x \in l
------------------------------------------------------------------
1 <= size l
```

`have : `ϕ Logical cut. Generate two subgoals: one whose conclusion is the cut formula ϕ, and one with conclusion $\phi => \psi$, where ψ is the current goal's conclusion.
Example: applying `have : x / (x => false)` to

```
x: bool
notnot_x: (x => false) => false
------------------------------------------------------------------
x
```

is transformed into the following subgoals

```
x: bool
notnot_x: (x => false) => false
------------------------------------------------------------------
x \/ (x => false)
```

and

```
x: bool
notnot_x: (x => false) => false
------------------------------------------------------------------
x \/ (x => false) => x
```

`apply` ϕ Tries to match the conclusion of the proof term ϕ with the goal's conclusion.
Example: applying `apply H` to

```
x: int
H: P x
------------------------------------------------------------------
P x
```

solves the goal.

`rewrite` $\phi_1 \dots \phi_n$ Rewrite the rewrite-pattern $\phi_1 \dots \phi_n$ from left to right.
Example: applying `rewrite eq_xy` to

```
x: int
y: int
eq_xy: x = y
z: int
eq_yz: y = z
------------------------------------------------------------------
x = z
```

is transformed into

```
x: int
y: int
eq_xy: x = y
z: int
eq_yz: y = z
------------------------------------------------------------------------
y = z
```

`subst` ϕ Search for the first equation of the form $x = t$ or $t = x$ in the context and replace all the occurrences of x by t everywhere in the context and the conclusion before clearing it.

Example: applying `subst x` to

```
x: bool
y: bool
z: bool
w: bool
eq_yx: y = x
eq_yz: y = z
eq_zw: z = w
------------------------------------------------------------------------
x = w
```

is transformed into

```
x: bool
y: bool
z: bool
w: bool
eq_yz: y = z
eq_zw: z = w
------------------------------------------------------------------------
y = w
```

`case` ϕ Assuming the goal's conclusion is not a statement judgement, do an excluded-middle case analysis on ϕ, substituting ϕ in the goal's conclusion.

Example: applying `case (x <= y)` to

```
x: int
y: int
------------------------------------------------------------------------
0 <= y - x
```

is transformed into the following subgoals

```
x: int
y: int
------------------------------------------------------------------------
x <= y
```

and

```
x: int
y: int
------------------------------------------------------------------------
x <= y => 0 <= y - x
```

`elim` ϕ Eliminates the top assumption of the goal's conclusion, generating subgoals that are dependent upon the kind of assumption eliminated.

Example: applying `elim l` to

```
l: 'a list
------------------------------------------------------------------------
0 <= size l
```

is transformed into the following subgoals

```
------------------------------------------------------------------------
0 <= size []
```

and

```
------------------------------------------------------------------------
forall (x : 'a) (l : 'a list), 0 <= size l => 0 <= size (x :: l)
```

`simplify` Attempts to simplify the proof goal by solving trivial equalities or even by expanding operators being used.

Example: applying `simplify` to

```
x: 'a
l: 'a list
H: 0 <= size l
------------------------------------------------------------------------
0 <= size (x :: l)
```

is transformed into

```
x: 'a
l: 'a list
H: 0 <= size l
------------------------------------------------------------------------
0 <= 1 + size l
```

`assumption` Search in the context for a hypothesis that is convertible to the goal's conclusion, solving the goal if one is found. Fail if none can be found.

Example: applying `assumption` to

```
x: bool
H: P x
------------------------------------------------------------------------
P x
```

solves the goal.

`reflexivity` Solve goals with conclusions of the form $x = x$ (up to computation).

Example: applying `reflexivity` to

```
x: bool
------------------------------------------------------------------------
x = x
```

solves the goal.

`done` Apply `trivial` and fail if the goal is not closed.

`smt` Try to solve the goal using SMT solvers. The goal is sent along with the local hypotheses plus selected axioms and lemmas.

Example: applying `smt` to

```
x: int
y: int
z: int
H: x = y
H0: y = z
------------------------------------------------------------------------
x = z
```

solves the goal.

3.2 Axioms

EasyCrypt's allows users to axiomatize properties regarding types and operators. For example, using the (small) finite field library defined in Section 2.2, it is possible to formalize the expected properties of the field operators, like the commutativity or associative properties, using axioms as follows.

```
axiom addC (x y : t): x + y = y + x. (* Commutative addition property *)
axiom addA (x y z : t) : x + (y + z) = (x + y) + z. (* Associative addition property *)

axiom mulC (x y : t) : x * y = y * x. (* Commutative addition property *)
axiom mulA (x y z : t): x * (y * z) = (x * y) * z. (* Associative multiplication property *)
axiom mulfDl (x y z : t): (x + y) * (x + z) = x * (y + z). (* Distributive multiplication property over the addition *)
```

3.3 Lemmas

Lemmas are properties that, unlike axioms, are not assumed to be true and that require the user to write a complete proof for it. We provide two examples of EasyCrypt lemmas, together with their respective proof script:

1. one that proves that adding a field element to another element that is different than zero will output a different field element; and
2. one that proves that the size of any list is always greater than or equal to zero.

3.3.1 Adding a field element to another field element different than zero

To prove the desired property, one can write the following two lemmas. It is recommended to follow this example using the EasyCrypt framework in order to get a clear picture of how the proof evolves.

```
lemma add_fzero_imp (x : t) (y : t) : x + y = x => y = fzero.
proof.
  move => H.
  have : y = x + (- x) by smt.
  move => H0.
  rewrite H0.
  apply addfN.
qed.

lemma non_zero_add (x : t) (y : t) :
  y <> fzero => x + y <> x.
proof.
  move => H.
  case (x + y = x).
    move => H2.
    have : y = fzero.
      rewrite (add_fzero_imp x y).
        assumption.
      reflexivity.
    trivial.
  trivial.
qed.
```

3.3.2 Size of any list is always greater than or equal to zero

To prove the desired property, one can write the following lemma. It is recommended to follow this example using the EasyCrypt framework in order to get a clear picture of how the proof evolves.

```
lemma size_ge0 (l : 'a list) : 0 <= size l.
proof.
  elim l.
    simplify.
    trivial.
  move => x l Hind.
  simplify.
  smt.
qed.
```

4 Modules

So far, we have explored the *functional* core of EasyCrypt. Complementary to it, EasyCrypt also discloses an *imperative* subset, captured by *modules*.

EasyCrypt features a module system that provides a structuring mechanism for describing imperative constructions. Modules are composed of a *memory* (a set of global variables, here empty) and a set of procedures. Procedures in the same module may share state; it is therefore not necessary to explicitly add state to the module signature. In addition, modules can be parameterised by other modules (in which case, we often call them *functors*) whose procedures they can query like oracles.

Modules are mainly used for representing cryptographic games - either concrete or abstract. It uses a simple *while* language. For example, the `IND-CPA` security game can be is represented as the following concrete module:

```
module INDCPA (S: Scheme) (A: Adversary) = {
  proc main() : bool = {
    var b, b', m0, m1, k, m;

    k <@ S.key_gen();
    (m0, m1) <@ A.gen_query();
    b <$ {0,1};
    m <- if b then m1 else m0;
    c <@ S.encrypt(k, m);
    b' <@ A.guess(c);

    return b';
  }
}.
```

In this module, the secret key is first generated by accessing the `key_gen` procedure. Then, the *adversary* selects two messages `m0` and `m1`. The game proceeds by randomly sampling a bit that is used to determine which message is going to be encrypted. Finally, the adversary, will try to determine if the ciphertext given to it came from an encryption of `m0` or `m1`.

Note that we make use of different assignment syntaxes:

- `<-` - assignment of an expression
- `<@` - assignment of the output of a function call
- `<$` - random assignment, i.e., a random value will be sampled from a probability distribution

The `INDCPA` module is parameterized by a module of *type* `Scheme` and another of *type* `Adversary`. The constituents of a module and their types are reflected in their *module type*: a module `M` has module type `I` if all procedures declared in `I` are also defined in `M`, with the same type and parameters. For instance, the `Scheme` *module type*, intended to capture the *type* of symmetric encryption schemes, can be defined as follows

```
module type Scheme = {
  module key_gen() : key
  module encrypt(k : key, pt : plaintext) : ciphertext
  module decrypt(k : key, ct : ciphertext) : plaintext
}
```

meaning that a module that follows this interface will be considered to have *type* `Scheme`.

5 Hoare logic

To deconstruct imperative programs, EasyCrypt incorporates a Hoare logic proof engine. In EasyCrypt, a Hoare triple can be written according to the following syntax

```
lemma hoare_triple : hoare [p : pre ==> post ]
```

where `p` is the procedure to be analyzed, `pre` is the precondition and `post` is the postcondition.

5.1 Commonly used Hoare logic tactics

`proc` Turn a goal whose conclusion is a Hoare logic judgement involving concrete procedure(s) into one whose conclusion is a statement judgement by replacing the concrete procedure(s) by their body/ies.

Example: applying `proc` to

```
M : M
------------------------------------------------------------------------
pre = true

   Example(M).main

post = true
```

is transformed into

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true

(1--) z <@ M.gen()
(2--) r <$ [0..100]
(3--) if (x < 100) {
(3.1) y <- x * 2
(3--) } else {
(3?1) y <- r + z
(3--) }

post = true
```

where `r <$ [0..100]` captures the integer random sampling in the $[0; 100]$ range.

`wp` Applies the weakest precondition calculus strategy to the current program. `wp` will consume assignments, as well as *if* conditionals whose body does not encompass any random sample or function calls.

Example: applying `wp` to

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true

(1--) z <@ M.gen()
(2--) r <$ [0..100]
(3--) if (x < 100) {
(3.1) y <- x * 2
(3--) } else {
(3?1) y <- r + z
(3--) }

post = true
```

is transformed into

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true

(1) z <@ M.gen()
(2) r <$ [0..100]

post = if x < 100 then true else true
```

`rnd` If the conclusion is a Hoare logic judgment whose program ends with a random assignments `x <$ d`, then consume those random assignments, replacing the conclusion's postcondition by the probabilistic weakest precondition of the random assignments.

Example: applying `rnd` to

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true

(1) z <@ M.gen()
(2) r <$ [0..100]

post = if x < 100 then true else true
```

is transformed into

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true

(1) z <@ M.gen()

post =
  forall (r0 : int),
    (r0 \in [0..100])%Distr => if x < 100 then true else true
```

`call (_ : )`ϕ If the conclusion is a Hoare logic judgement whose program end with a function call of the same abstract procedure, then use the specification argument to call generated from the invariant ϕ, and automatically apply `proc` ϕ to its first subgoal, pruning the first two subgoals the application generates, because their conclusions consist of ambient logic formulas that are true by construction.

Example: applying `call _ : true` to

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true

(1) z <@ M.gen()

post =
  forall (r0 : int),
    (r0 \in [0..100])%Distr => if x < 100 then true else true
```

is transformed to

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true


post =
  forall (r0 : int),
    (r0 \in [0..100])%Distr => if x < 100 then true else true
```

`skip` If the goal's conclusion is a statement judgement whose program(s) are empty, reduce it to the goal whose conclusion is the ambient logic formula ϕ => ψ, where ϕ is the original conclusion's precondition, and ψ is its postcondition.

Example: applying `skip` to

```
M : M
------------------------------------------------------------------------
Context : {x, r, y, z : int}

pre = true


post =
  forall (r0 : int),
    (r0 \in [0..100])%Distr => if x < 100 then true else true
```

is transformed to

```
M : M
------------------------------------------------------------------------
forall &hr,
  true =>
  forall (r0 : int),
    (r0 \in [0..100])%Distr => if x{hr} < 100 then true else true
```

`while` ϕ If the goal's conclusion is a Hoare logic judgement whose program ends with a `while` statements, reduce the goal to two subgoals whose conclusions are Hoare logic judgments:

- One whose program is the body of the `while` statement, whose precondition is the conjunction of ϕ and the while statements' boolean expressions and whose postcondition is the conjunction of ϕ and the assertion that the while statements' boolean expressions (interpreted in the appropriate memories) are equivalent. Essentially, one is required to prove that the invariant ϕ is preserved throughout the loop execution
- One whose precondition is the original goal's precondition, whose program is the results of removing the while statement from the original program, and whose postcondition is the conjunction of:
  - the conjunction of ϕ and the assertion that the while statement's boolean expressions are equivalent; and
  - the assertion that, for all values of the variables modified by the while statement, if the while statement's boolean expressions don't hold, but ϕ holds, then the original goal's postcondition holds.

  Essentially, one is required to prove that the invariant holds at the beginning of the loop and at the end of the loop.

Example: applying `while (0 <= i <= 10 / y = x * i)` to

```
_x: int
------------------------------------------------------------------------
Context : {x, y, i : int}

pre = x = _x

(1--) i <- 0
(2--) y <- 0
(3--) while (i < 10) {
(3.1) y <- y + x
(3.2) i <- i + 1
(3--) }

post = y = _x * 10
```

is transformed into the following subgoals

```
_x: int
------------------------------------------------------------------------
Context : {x, y, i : int}

pre = ((0 <= i && i <= 10) /\ y = x * i) /\ i < 10

(1) y <- y + x
(2) i <- i + 1

post = (0 <= i && i <= 10) /\ y = x * i
```

and

```
_x: int
------------------------------------------------------------------------
Context : {x, y, i : int}

pre = x = _x

(1) i <- 0
```

```
(2) y <- 0

post =
  ((0 <= i && i <= 10) /\ y = x * i) /\
  forall (i0 y0 : int),
    ! i0 < 10 => (0 <= i0 && i0 <= 10) /\ y0 = x * i0 => y0 = _x * 10
```

`if` If the goal's conclusion is a Hoare logic judgement whose program begin with an if statement, reduces the goal to two subgoals:

- One in which the if statement has been replaced by its *then* branch, and where the assertion of the truth of the if statement's boolean expression has been added to the conclusion's precondition.
- One in which the if statement has been replaced by its *else* part, and where the assertion of the falsity of the if statement's boolean expression has been added to the conclusion's precondition.

Example: applying `if` to

```
_x: int
----------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100

(1--) if (x < 100) {
(1.1) y <- x
(1--) } else {
(1?1) y <- x * 2
(1--) }

post = y = _x
```

is transformed into the following subgoals

```
_x: int
----------------------------------------------------------------------
Context : {x, y : int}

pre = (x = _x /\ x < 100) /\ x < 100

(1) y <- x

post = y = _x
```

and

```
_x: int
----------------------------------------------------------------------
Context : {x, y : int}

pre = (x = _x /\ x < 100) /\ ! x < 100

(1) y <- x * 2

post = y = _x
```

`rcondt n` If the goal's conclusion is an Hoare logic judgement whose n statement is an `if` statement, reduce the goal to two subgoals:

- One whose concludion is an Hoare logic judgement whose precondition is the original goal's precondition, whose program is the first $n-1$ statements of the original goal's program, and whose postcondition is the boolean expression of the if statement.
- One whose conclusion is an Hoare logic judgement that's the same as that of the original goal except that the `if` statement has been replaced by its then part.

Example: applying `rcondt 1` to

```
_x: int
------------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100

(1--) if (x < 100) {
(1.1) y <- x
(1--) } else {
(1?1) y <- x * 2
(1--) }

post = y = _x
```

is transformed into the following subgoals

```
_x: int
------------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100


post = x < 100
```

and

```
_x: int
------------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100

(1) y <- x

post = y = _x
```

`rcondf n` If the goal's conclusion is an HL statement judgement whose n statement is an `if` statement, reduce the goal to two subgoals:

- One whose concludion is an Hoare logic judgment whose precondition is the original goal's precondition, whose program is the first $n-1$ statements of the original goal's program, and whose postcondition is the negation of the boolean expression of the if statement.
- One whose conclusion is an Hoare logic judgement that's the same as that of the original goal except that the `if` statement has been replaced by its else part.

Example: applying `rcondf 1` to

```
_x: int
------------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100

(1--) if (x < 100) {
(1.1) y <- x
(1--) } else {
(1?1) y <- x * 2
(1--) }

post = y = _x
```

is transformed into the following subgoals

```
_x: int
------------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100


post = ! x < 100
```

and

```
_x: int
------------------------------------------------------------------------
Context : {x, y : int}

pre = x = _x /\ x < 100

(1) y <- x * 2

post = y = _x
```

5.2 EasyCrypt Hoare logic example

Consider the following EasyCrypt module

```
module type M = {
  proc gen() : int
}.

module Example (M : M) = {

  proc main(x : int) : int = {
    var r, y, z;

    z <@ M.gen();
    r <$ [0..100];
    if (x < 100) { y <- x*2; }
    else { y <- r + z; }

    return y;
  }
}.
```

where we define a module `Example` with a procedure `main` that either doubles its input or that assigns it to the output of a procedure call added to a random value. In this example, we will prove that if the input value `x` is less than 100 (precondition), then the output will be `x*2` (postcondition).

The EasyCrypt Hoare triple lemma is written bellow.

```
lemma example (M <: M) (_x : int) : hoare [Example(M).main : _x = x /\ x < 100 ==> res = _x * 2].
```

The `example` lemma does two universal quantifications: i. one over every possible modules of type `M` (using the `<:` notation); and ii. one over every possible integer `_x` While the former is done to correctly instantiate the `Example` module, the latter is done to be able to refer to the value of `x` before the program is executed. This allows us to store the value of `x` at the beginning of the evaluation and refer to it at the postcondition. Note also that the postcondition uses a special value dubbed `res`. This is an EasyCrypt keyword used to refer to the output of the program.

The following proof script is able to discharge the afore mentioned Hoare triple.

```
lemma example (M <: M) (_x : int) : hoare [Example(M).main : _x = x /\ x < 100 ==> res = _x * 2].
proof.
  proc.
  wp.
  rnd.
  call (_ : true).
  skip.
  move => &hr H result r H0.
  have : x{hr} < 100 by smt().
  have : x{hr} = _x by smt().
  move => H1 H2.
  rewrite H2.
  simplify.
  rewrite H1.
  reflexivity.
qed.
```

Again, for a better understanding of the proof process, it is highly recommended to reproduce the proof script in EasyCrypt.

6 Probabilistic Hoare logic

Probabilistic Hoare logic (pHL) allows one to write HL lemmas that are bounded by some probability. Intuitively, it allows the proof of statements where, given some precondition, the postcondition only occurs with a given probability P. A pHL triple can be written according to the following syntax

```
lemma phoare_triple : phoare [p : pre ==> post ] < P
```

Dealing with probability distributions inside EasyCrypt is, perhaps, the most complicated aspect of EasyCrypt. EasyCrypt provides a series of libraries to deal with probabilistic reasoning that we will not cover here. Instead, we will resort to the most relevant aspects of how probability distributions are formalized in EasyCrypt.

6.1 EasyCrypt probability distributions

Probability distributions in EasyCrypt are defined using the special `distr` type. For example, a probability distribution over the integers can be defined as

```
op int_distr : int distr.
```

The `support` of a distribution represents the elements that compose the domain of that distribution, i.e., those than can be sampled. For example, to restrict the domain of `int_distr` to the values between 0 and 100, one can write

```
axiom int_distr_support : forall (x : int), 0 <= x <= 100 => x \in int_distr.
axiom int_distr_supportN : forall (x : int), !(0 <= x <= 100) => x \notin int_distr.
```

The `weight` of a distribution establishes the sum of the probabilities of all elements of the distribution domains. Informally, we say that if the weight of a distribution is 1, then it is defined for all elements of the domain. In EasyCrypt, this is captured by the `is_lossless` predicate.

```
axiom int_distr_lossless : is_lossless int_distr.
```

Finally, it is possible to specify the probability of sampling a value in a probability distribution. In order to do so, EasyCrypt includes a special operator `mu`, that defines the probability of some event occurs in a distribution. Therefore, to define the sampling probability of an element, one can follow the next EasyCrypt script.

```
axiom int_distr_mu1 : forall (x : int), 0 <= x <= 100 => mu int_distr (fun k => k = x) = (1%r / 101%r).
```

6.2 EasyCrypt pHL example

Consider the following EasyCrypt module, that uses the previously formalized `int_distr` probability distribution.

```
module Example = {

  proc main() : int = {
    var x;

    x <$ int_distr;

    return (x * 2);
  }
}.
```

where we define a module `Example` that samples a value from `int_distr` and then doubles it. In this example, we will prove that the probability of this program outputting 100 is $\frac{1}{101}$, i.e., the probability of sampling 50.

The EasyCrypt pHL statement is written and proved bellow.

```
lemma example : phoare [Example.main : true ==> res = 10] <= (1%r / 101%r).
proof.
  proc.
  rnd.
  skip.
```

```
  progress.
  have : mu int_distr (fun (x : int) => x * 2 = 10) =
         mu int_distr (fun (x : int) => x = 5).
    congr.
    rewrite fun_ext /(==).
    move => x.
    smt.
  move => H.
  rewrite H int_distr_mu1.
  trivial.
  trivial.
qed.
```

7 Probabilistic Relational Hoare logic

Probabilistic Relational Hoare logic (pRHL) allows one to write HL lemmas that compare the execution of two programs. Intuitively, it allows the proof of statements where two programs are compared and where users can write pre and postconditions that refer to variables on both programs. Concretely, variables on the *left* program can be referred to using the `1` tag, whereas variables on the *right* program can be referred to using the `2` tag. A pRHL triple can be written according to the following syntax.

```
lemma p_relational_hoare_logic : equiv [p1 ~ p2 : pre ==> post].
```

The same tactics that were analyzed in Section 5 can be applied to pRHL judgments but, instead of consuming statements in a singe program, it will consume statements on both programs being analyzed.

7.1 Relational `rnd` tactic

The `rnd` tactic, when applied in a pRHL context, it follows a different behavior when comparing to HL. Concretely

`rnd | rnd f | rnd f g` If the conclusion is a pRHL judgement whose programs end with random assignments `x1 <$ d1` and `x2 <$ d2`, and f and g are functions between the types of x1 and x2, then consume those random assignments, replacing the conclusion's postcondition by the probabilistic weakest precondition of the random assignments wrt. f and g. The new postcondition checks that:

- `f` and `g` are an isomorphism between the distributions d1 and d2
- for all elements `u` in the support of `d1`, the result of substituting `u` and `f u` for `x11` and `x22` in the conclusion's original postcondition holds

Example: applying `rnd (fun b => if b then 3 else 2)(fun m => m = 3)` to

```
n: int
------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = y{2} = n

x <$ {0,1} (1) y <- y - 1
                    (2) x <$ [2..3]

post = x{1} <=> x{2} + y{2} = n + 2
```

is transformed into

```
n: int
------------------------------------------------------------------------
&1 (left ) : M.h
&2 (right) : N.h

pre = y{2} = n

x <$ {0,1} (1) y <- y - 1
                    (2) x <$ [2..3]
```

```
post =
  (forall (xR : int),
    xR \in [2..3] => xR = if xR = 3 then 3 else 2) &&
  (forall (xR : int),
    xR \in [2..3] => mu1 [2..3] xR = mu1 {0,1} (xR = 3)) &&
  forall (xL : bool),
    xL \in {0,1} =>
    ((if xL then 3 else 2) \in [2..3]) &&
    xL = ((if xL then 3 else 2) = 3) &&
    (xL <=> (if xL then 3 else 2) + y{2} = n + 2)
```

7.2 EasyCrypt pRHL example

Consider the following two EasyCrypt modules

```
module type M = {
  proc gen() : int
}.

module Example1 (M : M) = {

  proc main(x : int) : int = {
    var r;

    z <@ M.gen()
    r <$ [0..100]
    if (x < 100) {
      y <- x * 2
    } else {
      y <- r + z
    }

    return y;
  }
}.

module Example2 (M : M) = {

  proc main(x : int) : int = {
    var r;

    z <@ M.gen()
    r <$ [0..100]
    if (x < 50) {
      y <- x * 2
    } else {
      y <- r + z
    }

    return y;
  }
}.
```

Using pRHL, it is possible to prove that, when the input of both programs is lower than 50, they will produce the same output.

The EasyCrypt pRHL statement is written and proved bellow.

```
lemma example (M <: M) : equiv [Example1(M).main ~ Example2(M).main : (glob M){1} = (glob M){2} /\ x{1} = x{2} /\ x{1} < 50
     ==> res{1} = res{2}].
proof.
  proc.
  wp.
  rnd.
  call (_ : true).
  skip.
  progress.
  smt().
  smt().
qed.
```