

EASyCRYPT's Probabilistic Relational Hoare Logic and Probabilistic Noninterference

These slides are an example-based introduction to EASYCRYPT's Probabilistic Relational Hoare Logic (pRHL), focusing on how it can be used to prove probabilistic noninterference results.

More information about pRHL can be found in Section 3.4 of the EASYCRYPT manual:

<https://www.easycrypt.info/documentation/refman.pdf>

When we have been using EASYCRYPT's Relational Hoare Logic, we've already been using pRHL—but just not the probabilistic aspects of the while language or logic.

Datatype, Axioms and Lemmas for Our Examples

The examples that follow use the following datatype, a type `mybool` with elements `tt` (“true”) and `ff` (“false”):

```
type mybool = [tt | ff].
```

We introduce and axiomatize the exclusive or operator as follows:

```
op (^) : mybool -> mybool -> mybool.
```

```
axiom nosmt xor_tt_tt : tt ^ tt = ff.
```

```
axiom nosmt xor_tt_ff : tt ^ ff = tt.
```

```
axiom nosmt xor_ff_tt : ff ^ tt = tt.
```

```
axiom nosmt xor_ff_ff : ff ^ ff = ff.
```

Here we are saying that `smt` may not use these axioms. Because `tt` is an alias for the value `()` of type `unit`, you’ll sometimes see `Top.tt` in goals, meaning the version of `tt` that we have introduced.

Datatype, Axioms and Lemmas for Our Examples

We can then prove the following lemmas by hand (not using smt), which we *will* allow smt to use:

```
(* false on the right *)
lemma xor_ff (x : mybool)      : x ^^ ff = x.

(* canceling *)
lemma xorK   (x : mybool)      : x ^^ x = ff.

(* commutativity *)
lemma xorC   (x y : mybool)     : x ^^ y = y ^^ x.

(* associativity *)
lemma xorA   (x y z : mybool)   : (x ^^ y) ^^ z = x ^^ (y ^^ z).
```

(Sub-)Distributions

To make use of (sub-)distributions in EASYCRYPT, we

```
require import Distr.
```

This gives us types 'a distr, consisting of sub-distributions over the type 'a. This means that the sum of the values of the elements of 'a in the sub-distribution may be strictly less than the real number 1 (which is written 1%r).

We can then declare dmybool to be a sub-distribution on our datatype mybool by:

```
op dmybool : mybool distr.
```

To say that dmybool is a distribution, we introduce the axiom:

```
axiom dmybool_ll : is_lossless dmybool.
```

This says that the sum of the values of tt and ff in dmybool is 1%r.

(Sub-)Distributions

If d is a sub-distribution on type 'a, and E is an event (i.e., a predicate) on 'a (i.e., E is a function from 'a to bool), then $\text{mu } d E$ is the probability that choosing a value from d will satisfy E .

If d is a sub-distribution on type 'a, and x is a value of type 'a, then $\text{mu1 } d x$ is the probability that choosing a value from d will result in x .

mu1 is defined by

$$\text{mu1 } d x = \text{mu } d (\text{pred1 } x)$$

where $\text{pred1 } x$ is the predicate that is only satisfied by x .

(Sub-)Distributions

Thus we can axiomatize that `tt` and `ff` both have probability one half in `dmybool` via:

```
axiom dmybool1E (b : mybool) :  
  mul dmybool b = 1%r / 2%r.
```

We can then prove that `dmybool` is *full*, i.e., that its *support* (the values of the type given non-zero probabilities by the distribution) is all of `dmybool`:

```
lemma dmybool_fu : is_full dmybool.  
proof.  
  rewrite /is_full => x.  
  by rewrite /support dmybool1E StdOrder.RealOrder.divr_gt0.  
qed.
```

(It takes a little digging through the `Distr` theory to understand this proof.)

Random Assignment

If x is a program variable of type 'a, and d is sub-distribution on 'a, i.e., a value of type 'a distr, then

$x \leftarrow d;$

means to assign to x a value from 'a, where the probability that a given value v in 'a is chosen is equal to v 's value in d . If d is not a distribution, there is some probability the random assignment will cause the program to abort.

First Example

For our first probabilistic noninterference example, consider the program

```
module M1 = {  
  var x : mybool (* private *)  
  var y : mybool (* public *)  
  
  proc f() : unit = {  
    var b : mybool;  
    b <$ dmybool;  
    y <- x ^^ b;  
  }  
}.
```

The procedure `M1.f` exclusive or's the private variable `M1.x` of type `mybool` with a randomly chosen value `b`, updating the public variable `M1.y` with the result.

First Example

We state and begin our noninterference proof as usual:

```
lemma lem1 :  
  equiv[M1.f ~ M1.f : = {M1.y} ==> = {M1.y}].  
proof.  
proc.  
wp.
```

taking us to the goal

First Example

Type variables: <none>

&1 (left) : M1.f [programs are in sync]

&2 (right) : M1.f

pre = ={M1.y}

(1) b <\$ dmybool

post = M1.x{1} ^^ b{1} = M1.x{2} ^^ b{2}

Because *both* programs *end* with random assignments (the same in this case), we can apply the

rnd.

tactic, which pushes the two random assignments into the postcondition, giving us the goal

First Example

Type variables: <none>

```
&1 (left ) : M1.f [programs are in sync]
&2 (right) : M1.f
```

```
pre = ={M1.y}
```

```
post =
  (forall (bR : mybool),
    bR \in dmybool => bR = bR) &&
  (forall (bR : mybool),
    bR \in dmybool =>
      mu1 dmybool bR = mu1 dmybool bR) &&
  forall (bL : mybool),
    bL \in dmybool =>
      (bL \in dmybool) &&
      bL = bL && M1.x{1} ^^ bL = M1.x{2} ^^ bL
```

First Example

Unfortunately, a crucial part of the postcondition requires us to prove

$$M1.x\{1\} \wedge bL = M1.x\{2\} \wedge bL$$

for a universally quantified bL , which is impossible, given that we don't know the relationship between $M1.x$ in the two memories.

Thankfully, though, the `rnd` tactic takes, as an optional argument, an isomorphism h between the distributions of the two random assignments. (See below for exactly what being such an isomorphism means; by default, `rnd` uses the identity function when the types are the same.) In the above part of the postcondition, the right occurrence of bL will be replaced by $h\ bL$, and we should choose h so as to make the resulting equality provable (and still, hopefully, be an isomorphism).

First Example

Consequently, we should run

```
rnd (fun z => M1.x{1} ^^ M1.x{2} ^^ z).
```

(“fun z => ...” is an anonymous function that takes in an argument z and returns ...) which gives us a postcondition where the previously problematic part is

$$M1.x\{1\} \wedge bL = M1.x\{2\} \wedge (M1.x\{1\} \wedge M1.x\{2\} \wedge bL)$$

This is provable using

```
smt(xorA xorC xorK xor_ff).
```

First Example

To understand the rest of the postcondition generated by our use of `rnd`, we can follow it with

```
skip; progress.
```

which gives us five subgoals. The first subgoal is

```
Type variables: <none>
```

```
&1: memory <M1.f>  
&2: memory <M1.f>  
bR: mybool  
H : bR \in dmybool
```

```
bR = M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bR)
```

which makes us prove that, for all `bR` (“R” for chosen from the right-hand distribution) in the support of `dmybool` (all of `mybool`), running our argument to `rnd` twice gets us back to where we started. This can be proved by

```
smt(xorA xorC xorK xor_ff).
```

First Example

The second subgoal is

Type variables: <none>

&1: memory <M1.f>

&2: memory <M1.f>

H : forall (bR0 : mybool),

 bR0 \in dmybool =>

 bR0 =

 M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bR0)

bR: mybool

H0: bR \in dmybool

mu1 dmybool bR = mu1 dmybool (M1.x{1} ^^ M1.x{2} ^^ bR)

This gives us the conclusion of the first subgoal (H), and makes us prove that for all bR in the support of dmybool, the value of bR in dmybool is the same as the value of the result of applying our argument to rnd to bR. This can be solved with

smt(dmybool1E).

First Example

The third subgoal is

Type variables: <none>

&1: memory <M1.f>

&2: memory <M1.f>

H : forall (bR : mybool),

 bR \in dmybool =>

 bR = M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bR)

H0: forall (bR : mybool),

 bR \in dmybool =>

 mu1 dmybool bR =

 mu1 dmybool (M1.x{1} ^^ M1.x{2} ^^ bR)

bL: mybool

H1: bL \in dmybool

M1.x{1} ^^ M1.x{2} ^^ bL \in dmybool

First Example

This makes us prove that for all bL in the support of the left distribution, the result of applying our argument to rnd to bL is in the support of the right distribution (both distributions are the same in our case). This can be solved with

```
smt(dmybool_fu).
```

First Example

The fourth subgoal is

Type variables: <none>

&1: memory <M1.f>

&2: memory <M1.f>

H : forall (bR : mybool),

 bR \in dmybool =>

 bR = M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bR)

H0: forall (bR : mybool),

 bR \in dmybool =>

 mu1 dmybool bR =

 mu1 dmybool (M1.x{1} ^^ M1.x{2} ^^ bR)

bL: mybool

H1: bL \in dmybool

H2: M1.x{1} ^^ M1.x{2} ^^ bL \in dmybool

bL = M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bL)

First Example

This is the same as H, except starting from the left distribution instead of the right one. Consequently, we can solve this goal in our case by

by apply H.

First Example

Finally, the fifth subgoal is

Type variables: <none>

&1: memory <M1.f>

&2: memory <M1.f>

H : forall (bR : mybool),

 bR \in dmybool =>

 bR = M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bR)

H0: forall (bR : mybool),

 bR \in dmybool =>

 mu1 dmybool bR =

 mu1 dmybool (M1.x{1} ^^ M1.x{2} ^^ bR)

bL: mybool

H1: bL \in dmybool

H2: M1.x{1} ^^ M1.x{2} ^^ bL \in dmybool

H3: bL = M1.x{1} ^^ M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bL)

M1.x{1} ^^ bL = M1.x{2} ^^ (M1.x{1} ^^ M1.x{2} ^^ bL)

As we noted before, this can be solved by

smt(xorA xorC xorK xor_ff).

First Example

Putting it all together, the complete probabilistic noninterference proof for our first example is:

```
lemma lem1 :
  equiv[M1.f ~ M1.f : = {M1.y} ==> = {M1.y}].
proof.
proc.
wp.
rnd (fun z => M1.x{1} ^^ M1.x{2} ^^ z).
skip; progress.
smt(xorA xorC xorK xor_ff).
smt(dmybool1E).
smt(dmybool_fu).
by apply H.
smt(xorA xorC xorK xor_ff).
qed.
```

Second Example

For our second probabilistic noninterference example, consider the program

```
module M2 = {
  var x : mybool (* private *)
  var y : mybool (* public *)

  proc f() : unit = {
    var b : mybool;
    if (x = tt) {
      b <$ dmybool;
      if (b = tt) {
        y <- y ^^ tt;
      }
    }
    else {
      b <- ff;
    }
    y <- y ^^ b;
  }
}.
```

Second Example

Because the procedure `M2.f` is branching on the value of the private variable `M2.x`, we will have to use the one-sided `if` tactics. In fact, we can begin our proof by running

```
proc; wp; if{1}; if{2}.
```

which gives us four subgoals, corresponding to the four combinations of whether the conditional's boolean expression holds or does not hold in the two memories.

Second Example

The first subgoal is

```
Type variables: <none>
```

```
-----  
&1 (left ) : M2.f [programs are in sync]  
&2 (right) : M2.f
```

```
pre = (={M2.y} /\ M2.x{1} = Top.tt) /\ M2.x{2} = Top.tt
```

```
(1-- ) b <$ dmybool  
(2-- ) if (b = Top.tt) {  
(2.1)   M2.y <-  
(   )   M2.y ^^ Top.tt  
(2-- ) }
```

```
post = M2.y{1} ^^ b{1} = M2.y{2} ^^ b{2}
```

We can solve this goal by running `auto`, which correctly guesses that `rnd` should be applied using the identity isomorphism.

Second Example

But it's instructive to see how we could prove it by first running

```
seq 1 1 : (=M2.y, b).
```

giving us two sub-subgoals, the first of which is

```
Type variables: <none>
```

```
-----  
&1 (left ) : M2.f [programs are in sync]
```

```
&2 (right) : M2.f
```

```
pre = (=M2.y / \ M2.x{1} = Top.tt) / \ M2.x{2} = Top.tt
```

```
(1) b <$ dmybool
```

```
post = =M2.y, b}
```

We can solve this goal with

```
rnd; auto.
```

Second Example

This leaves us with

```
Type variables: <none>
```

```
-----  
&1 (left ) : M2.f [programs are in sync]
```

```
&2 (right) : M2.f
```

```
pre = ={M2.y, b}
```

```
(1-- )  if (b = Top.tt) {
```

```
(1.1)   M2.y <-
```

```
( )     M2.y ^^ Top.tt
```

```
(1-- ) }
```

```
post = M2.y{1} ^^ b{1} = M2.y{2} ^^ b{2}
```

which auto will solve.

Second Example

The second subgoal is

Type variables: <none>

&1 (left) : M2.f

&2 (right) : M2.f

pre = (={M2.y} /\ M2.x{1} = Top.tt) /\ M2.x{2} <> Top.tt

b <\$	(1--)	b <- ff
dmybool	(-)	
if (b =	(2--)	
Top.tt) {	(-)	
M2.y <-	(2.1)	
M2.y ^^	()	
Top.tt	()	
}	(2--)	

post = M2.y{1} ^^ b{1} = M2.y{2} ^^ b{2}

Second Example

Again, it's instructive to start with using `seq`, this time to pick off just the random assignment in the left program:

```
seq 1 0 : (=M2.y).
```

giving us two sub-subgoals, the first of which is

```
Type variables: <none>
```

```
-----  
&1 (left ) : M2.f
```

```
&2 (right) : M2.f
```

```
pre = (=M2.y /\ M2.x{1} = Top.tt) /\ M2.x{2} <> Top.tt
```

```
b <$ (1)
```

```
  dmybool ( )
```

```
post = =M2.y
```

Second Example

Here, only the first program *ends* with a random assignment, and so we can't run `rnd`. We can however run the one-sided version

```
rnd{1}.
```

which gives us

Second Example

Type variables: <none>

&1 (left) : M2.f [programs are in sync]

&2 (right) : M2.f

pre = (={M2.y} /\ M2.x{1} = Top.tt) /\ M2.x{2} <> Top.tt

post =

 is_lossless dmybool &&

 forall (b0 : mybool), b0 \in dmybool => =={M2.y}

The postcondition of this goal makes us prove that the distribution we are choosing from in the left program is lossless (is a distribution, not just a sub-distribution).

Second Example

The postcondition also makes us prove that for all values b_0 in the support of the distribution, the original postcondition holds, where b_0 would have been substituted for any occurrences of $b\{1\}$ (there are none, though). We can solve this goal with

```
skip; smt(dmybool_11).
```

Second Example

This leaves us with the sub-subgoal

Type variables: <none>

&1 (left) : M2.f

&2 (right) : M2.f

pre = ={M2.y}

```
if (b = (1--)) b <- ff
  Top.tt) { ( -)
  M2.y <- (1.1)
  M2.y ^^ ( )
  Top.tt ( )
}
```

post = M2.y{1} ^^ b{1} = M2.y{2} ^^ b{2}

which can be solved using techniques we've already studied.

Second Example

The third subgoal is symmetric to the second one

Type variables: <none>

&1 (left) : M2.f

&2 (right) : M2.f

pre = (={M2.y} /\ M2.x{1} <> Top.tt) /\ M2.x{2} = Top.tt

```
b <- ff                (1-- ) b <$
                       ( - )   dmybool
                       (2-- ) if (b =
                       ( - )     Top.tt) {
(2.1)   M2.y <-
(   )   M2.y ^^
(   )   Top.tt
(2-- ) }
```

post = M2.y{1} ^^ b{1} = M2.y{2} ^^ b{2}

Second Example

And the fourth and final subgoal is

Type variables: <none>

&1 (left) : M2.f [programs are in sync]

&2 (right) : M2.f

pre = (={M2.y} /\ M2.x{1} <> Top.tt) /\ M2.x{2} <> Top.tt

(1) b <- ff

post = M2.y{1} ^^ b{1} = M2.y{2} ^^ b{2}

which can be solved by

auto.

Second Example

Putting it all together, and simplifying a bit, the complete probabilistic noninterference proof for our second example is:

```
lemma lem2' :
  equiv[M2.f ~ M2.f : = {M2.y} ==> = {M2.y}].
proof.
proc; if{1}; if{2}; wp.
(* first case *)
auto.
(* second case *)
auto; progress.
smt(dmybool_ll).
smt(xorA xorK).
smt().
(* third case *)
auto; progress.
smt(dmybool_ll).
smt(xorA xorK).
smt().
(* fourth case *)
auto.
qed.
```