Using EASYCRYPT’s Ambient Logic

These slides are an example-based introduction to the use of EASYCRYPT’s ambient logic.

See Chapter 2 and Sections 3.1–3.3 of the EASYCRYPT reference manual for more information:
https://www.easycrypt.info/documentation/refman.pdf
**Types**

EasyCrypt’s types include basic types like `unit` (which only has the single element `()`), `int`, `bool` and `real`, as well as product types $t_1 \times t_2 \cdots \times t_n$ and function types $t_1 \to t_2$. $\times$ has higher precedence than $\to$, and $\to$ is right associative.

Thus, e.g., $t_1 \times t_2 \to t_3 \to t_4$ means $(t_1 \times t_2) \to (t_3 \to t_4)$. A value of this type is a function that takes in a pair $(x,y)$, where $x$ has type $t_1$ and $y$ has type $t_2$, and returns a function that takes in a value $z$ of type $t_3$, and returns a result of type $t_4$. 
Operators

EASYCRYPT has typed operators (or functions). E.g.,

\[
\begin{align*}
\text{op } f \ (x \ y : \text{int}) &= x - 2 \times y. \\
\text{op } g \ (a \ b : \text{bool}) &= \neg(a \land b) \land (a \lor b). \\
\text{op } h : \text{int} \to \text{bool}.
\end{align*}
\]

Here the prefix operator \(!\) is boolean negation, and the infix operators \(\land\) and \(\lor\) are conjunction and disjunction, respectively. We also have implication \(\to\) and if-and-only-of \(\iff\). \(f\) and \(g\) have (curried) types:

\[
\begin{align*}
f : \text{int} \to \text{int} \to \text{int} \\
g : \text{bool} \to \text{bool} \to \text{bool}
\end{align*}
\]

Thus you can say

\[
\begin{align*}
\text{op } p : \text{int} \to \text{int} &= f \ 4. \\
\text{op } y : \text{int} &= p \ 5.
\end{align*}
\]

in which case the value of \(y\) will be \(-6\).
Operators

If \( x \) is a value of type \( \text{int} \), then \( x\%r \) is the corresponding element of \( \text{real} \). Operators in \texttt{EASYCRYPT} can be overloaded, so that, e.g., \( * \) is multiplication for both \( \text{int} \) and \( \text{real} \).
Axioms and Lemmas

We can state axioms like

\[
\text{axiom } h_{\text{ax}} \ (x : \text{int}) : x \neq 0 \Rightarrow h \ x.
\]

which says that for all non-zero integers \(x\), the result of applying \(h\) to \(x\) returns \(true\), i.e., \(h \ x\) holds.

We can state and prove lemmas like

\[
\text{lemma } \text{not_or} \ (a \ b : \text{bool}) : \neg(a \lor b) \Rightarrow \neg a \land \neg b.
\]

\[
\text{proof.}
\]

\[
\text{...}
\]

\[
\text{qed.}
\]

which says that the negation of the disjunction of \(a\) and \(b\) implies the conjunction of the negation of \(a\) and the negation of \(b\).

Here the \text{...} should consist of a sequence of tactics proving the lemma.
Theories

**EasyCrypt** has various *theories* in its standard library, each of which contains operators, axioms, lemmas and subtheories. See the subdirectory `theories` of the **EasyCrypt** distribution.

The theory `AllCore` contains some core theories like `Int` and `Real`—corresponding to the integers and real numbers.

Issuing the command

```latex
require import T.
```

makes the definitions of the theory `T` available without qualification (so you can say `f` instead of `T.f`).
Printing and Searching

Operators, lemmas and axioms may be printed using the print command:

\begin{verbatim}
print g.
print [!].
print (\/).
\end{verbatim}

Note the special way unary and binary operators are specified. The \texttt{search} command can be used to search for all lemmas and axioms involving all of a list of operators. E.g.,

\begin{verbatim}
search [!] (\/) (=>).
\end{verbatim}

searches for all lemmas involving all of negation, disjunction and implication.
Proof Process

At each point of proving a lemma, we have some number of goals, and are focused on one of them. Goals consist of an ordered set of assumptions (listed above the horizontal bar) plus a single conclusion (listed below the bar).

When working on a proof, one may temporarily accept a goal, without proof, by running the tactic admit.
Basic Tactics

The conclusion of a goal can be logically simplified using the tactic `simplify`. E.g., `simplify` transforms

```
Type variables: <none>

x : int
y : int
--------------------------------------------
!true \/ x < y
```

into

```
Type variables: <none>

x : int
y : int
--------------------------------------------
x < y
```
Basic Tactics

The trivial tactic applies a set of basic logical rules, and can solve certain goals, e.g.:

Type variables: <none>

\[
\begin{align*}
a : & \text{ bool} \\
b : & \text{ bool} \\
\hline
a \Rightarrow & \neg (\text{true} \land b) \Rightarrow \neg \text{true} \lor \neg b
\end{align*}
\]

and (because it can establish a contradiction)

Type variables: <none>

\[
\begin{align*}
a : & \text{ bool} \\
b : & \text{ bool} \\
\text{not\_and} : & \neg (a \land b) \\
a\_true : & a \\
b\_true : & b \\
\hline
a \land b & \Rightarrow \text{false}
\end{align*}
\]
SMT Solvers

The `smt` tactic uses the known SMT solvers to try to solve a goal, using all known lemmas.

Running `smt()` means to only use lemmas built-in to the solvers.

One can also list the previously proved lemmas that may be used, e.g., `smt(foo goo)`.

One can restrict which solvers may be used, e.g.,

```
prover ["Z3" "Alt-Ergo"].
```

One can customize the timeout (in seconds) before an application of `smt` will fail:

```
timeout 2.
```
Introduction Patterns: Simple

Introduction patterns may be used to introduce into the goal’s assumptions universally quantified variables as well as the left sides of implications. E.g., move => x y z le_x_y le_y_z transforms

Type variables: <none>

for all (x y z : int), x <= y => y <= z => x <= z

into

Type variables: <none>

x : int
y : int
z : int
le_x_y: x <= y
le_y_z: y <= z
--------------------------------------------
x <= z
**Introduction Patterns: Elimination**

Introduction patterns may be used to eliminate disjunctions, existentially quantified formulas, and conjunctions on the left sides of implications. E.g., \texttt{move => []} transforms

\[
\begin{align*}
\text{Type variables: <none>}
\end{align*}
\]

\[
\begin{align*}
a & : \text{bool} \\
b & : \text{bool} \\
\hline
a \lor b & => a
\end{align*}
\]

into
Introduction Patterns: Elimination

Type variables: <none>

a : bool
b : bool
---------------------------------------------
a => a

and

Type variables: <none>

a : bool
b : bool
---------------------------------------------
b => a
**Introduction Patterns: Elimination**

And move => [] transforms

Type variables: <none>

\[
\begin{align*}
y & : \text{int} \\
\text{(exists } (x : \text{int}), \ y = x \times 2 + 1) & => \\
\text{exists } (z : \text{int}), \ y - 3 & = z \times 2
\end{align*}
\]

into

Type variables: <none>

\[
\begin{align*}
y & : \text{int} \\
\text{forall } (x : \text{int}), \ y = x \times 2 + 1 & => \\
\text{exists } (z : \text{int}), \ y - 3 & = z \times 2
\end{align*}
\]
Introduction Patterns: Elimination

And move => [] transforms

Type variables: <none>

a : bool
b : bool

--------------

a \(\land\) b => a

into

Type variables: <none>

a : bool
b : bool

--------------

a => b => a
Elimination

One can do elimination of an assumption using \texttt{elim}. E.g.,

\begin{verbatim}
elim H.
\end{verbatim}

transforms

\begin{verbatim}
Type variables: <none>

a : bool
b : bool
H : a \lor b
--------------------------------------------------------------
a
\end{verbatim}

into
Elimination

Type variables: <none>

a : bool
b : bool
--------------------------------------------
a => a

and

Type variables: <none>

a : bool
b : bool
--------------------------------------------
b => a
Introduction Patterns Following Arbitrary Tactic

Any tactic may be followed by an introduction pattern, which applies to the subgoals created by running the tactic. And one may specify different introduction patterns for different subgoals. E.g.,

```
elim H => [a_true | b_true].
```

transforms

```
Type variables: <none>

a : bool
b : bool
H : a ∨ b
--------------------------
a
```

into

```
```
Introduction Patterns Following Arbitrary Tactic

Type variables: <none>

\[
a : \text{bool} \\
b : \text{bool} \\
a_{\text{true}} : a
\]

--------------------------------------------

a

and

Type variables: <none>

\[
a : \text{bool} \\
b : \text{bool} \\
b_{\text{true}} : b
\]

--------------------------------------------

a
Case Analysis

The case tactic can be used to do case analysis. E.g.,

    case a

transforms

    Type variables: <none>

    a : bool
    b : bool

    ! (a \ b) => !a \b

into
Case Analysis

Type variables: <none>

\[ a : \text{bool} \]
\[ b : \text{bool} \]

\[ a \Rightarrow \neg (\text{true} \land b) \Rightarrow \neg \text{true} \lor \neg b \]

and

Type variables: <none>

\[ a : \text{bool} \]
\[ b : \text{bool} \]

\[ \neg a \Rightarrow \neg (\text{false} \land b) \Rightarrow \neg \text{false} \lor \neg b \]
Including /\ (resp., //, /\#) in an introduction pattern means apply simplify (resp., trivial, smt()) to the current goal. E.g.,

```
case a => //
```
solves the goal

Type variables: <none>

a : bool
b : bool

! (a /\ b) => !a \/ !b
**Splitting Conjunctions**

When a goal’s conclusion is a conjunction, if-and-only-if or equality of tuples, it may be split into multiple subgoals using split. E.g., split transforms the goal

Type variables: <none>

\[
\begin{align*}
a &: \text{bool} \\
b &: \text{bool} \\
\text{not_or} &: \neg (a \lor b) \\
\text{---} & \\
\neg a & \lor \neg b
\end{align*}
\]

into

\[
\neg a \lor \neg b
\]
Splitting Conjunctions

Type variables: <none>

a : bool
b : bool
not_or: ! (a \ b)
---------------------------------------------
!a

and

Type variables: <none>

a : bool
b : bool
not_or: ! (a \ b)
---------------------------------------------
!b
Splitting Conjunctions

And split transforms the goal

Type variables: <none>

\[ \neg (a \lor b) \iff \neg a \land \neg b \]

\[ ! (a \lor b) \iff !a \land \neg !b \]

into
Splitting Conjunctions

Type variables: <none>

\[ a : \text{bool} \]
\[ b : \text{bool} \]

\[ ! (a \lor b) \Rightarrow !a \land !b \]

and

Type variables: <none>

\[ a : \text{bool} \]
\[ b : \text{bool} \]

\[ !a \land !b \Rightarrow ! (a \lor b) \]
Splitting Conjunctions

And split transforms the goal

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
x' & : \text{int} \\
y & : \text{bool} \\
y' & : \text{bool} \\
eq_{x\_x'} & : x = x' \\
eq_{y\_y'} & : y = y' \\
\end{align*}
\]

\[(x, y) = (x', y')\]

into
Splitting Conjunctions

Type variables: <none>

\[ x : \text{int} \]
\[ x' : \text{int} \]
\[ y : \text{bool} \]
\[ y' : \text{bool} \]
\[ eq_{x_{-}x'} : x = x' \]
\[ eq_{y_{-}y'} : y = y' \]

\[ x = x' \]

and

\[ y = y' \]
Proving Disjunctions

The tactics left and right can be used to prove disjunctions. E.g., left transforms the goal

Type variables: <none>

a : bool
b : bool
a_true: a
---------------------------------------------
a \lor b

into

Type variables: <none>

a : bool
b : bool
a_true: a
---------------------------------------------
a
Proving Disjunctions

And right transforms the goal

Type variables: <none>

\[ a : \text{bool} \]
\[ b : \text{bool} \]
\[ b_{\text{true}} : b \]

\[ a \lor b \]

into

Type variables: <none>

\[ a : \text{bool} \]
\[ b : \text{bool} \]
\[ b_{\text{true}} : b \]

\[ b \]
Proving Existentially Quantified Formulas

The tactic `exists` can be used to prove existentially quantified formulas. E.g., `exists (x - 1)` transforms the goal

Type variables: <none>

\[
\begin{align*}
y & : \text{int} \\
x & : \text{int} \\
y_{eq} & : y = x \times 2 + 1 \\
\end{align*}
\]

---------

\[\text{exists (z : int), } y - 3 = z \times 2\]

into

Type variables: <none>

\[
\begin{align*}
y & : \text{int} \\
x & : \text{int} \\
y_{eq} & : y = x \times 2 + 1 \\
\end{align*}
\]

---------

\[y - 3 = (x - 1) \times 2\]
Proving Sublemmas

When working on proving a goal, one may prove and then use a sublemma using the tactic have. E.g.,

```
have : a \/ b.
```
	ransforms the goal

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
-----------------------------
false
```

into the two subgoals
Proving Sublemmas

Type variables: <none>

a : bool
b : bool
not_or: ! (a \ b)

\[ \text{a_true: a} \]

\[ \text{--------------------------------------------} \]

\[ \text{a \ b} \]

and

Type variables: <none>

a : bool
b : bool
not_or: ! (a \ b)

\[ \text{a_true: a} \]

\[ \text{--------------------------------------------} \]

\[ \text{a \ b} \Rightarrow \text{false} \]
**Proving Sublemmas**

What comes before `have` is an arbitrary introduction pattern to be applied to the second subgoal. E.g.,

```lean
have contrad : a \/ b.
```

transforms the goal

```
Type variables: <none>

a : bool
b : bool
not_or: ! (a \/ b)
a_true: a
--------------------------------------------
false
```

into the two subgoals
Proving Sublemmas

Type variables: <none>

a : bool  
b : bool  
not_or: ! (a \or b)  
a_true: a  
--------------------------------------------  
a \or b  

and  

Type variables: <none>

a : bool  
b : bool  
not_or: ! (a \or b)  
a_true: a  
contrad: a \or b  
--------------------------------------------  
false
Applying Lemmas

We can apply an already proven lemma using the apply tactic; it can also be used to apply an assumption. E.g., if we’ve already proved

\[
\text{lemma not_or_imp } (a \ b : \text{bool}) : \neg(a \lor b) \Rightarrow \neg a \land \neg b.
\]

then running

\[
\text{apply } (\text{not_or_imp } (x < y) (y < x)).
\]

solves the goal

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
y & : \text{int}
\end{align*}
\]

\[
\neg (x < y \lor y < x) \Rightarrow \neg x < y \land \neg y < x
\]
Applying Lemmas

And EasyCrypt can often infer the instantiations of the applied lemma’s parameters. E.g., running

```
apply not_or_imp.
```

solves the goal

```
Type variables: <none>

x : int
y : int

! (x < y \ y < x) => ! x < y \ \ ! y < x
```
Applying Lemmas

Furthermore, if we’ve proved an if-and-only-iff lemma, we can apply it in place of either the left-to-right or right-to-left implications. E.g., if we have

\[
\text{lemma not_or_iff (a b : bool) : !}(a \lor b) \iff !a \land !b.
\]

then running

\[
\text{apply not_or_iff}.
\]

solves the goal

Type variables: <none>

\[
x : \text{int}
y : \text{int}
\]

\[
! (x < y \lor y < x) \Rightarrow ! x < y \land ! y < x
\]
Rewriting Equational Lemmas

If we have equational lemmas like

\[
\text{lemma foo } (x : \text{int}) : f x = x + 1
\]

where the operator \( f \) has type \( \text{int} \to \text{int} \), we can rewrite them in formulas using the rewrite tactic.

E.g., the tactic

\[
\text{rewrite } (f\text{_eq }x).
\]

transforms the goal

\[
\text{Type variables: } <\text{none}>
\]

\[
x : \text{int} \\
y : \text{int}
\]

\[
-------------------------
\]

\[
f (f x \ast f y) = (x + 1) \ast (y + 1) + 1
\]

into
Rewriting Equational Lemmas

Type variables: <none>

\[ x : \text{int} \]
\[ y : \text{int} \]

\[ f \left( (x + 1) \times f \ y \right) = (x + 1) \times (y + 1) + 1 \]
Rewriting Equational Lemmas

And the tactic

\[ \text{rewrite (f_eq y).} \]

transforms the goal

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
\text{x : int} & \\
\text{y : int} & \\
\hline \\
f ((x + 1) * f y) & = (x + 1) * (y + 1) + 1
\end{align*}
\]

into

\[
\begin{align*}
\text{Type variables: } & <\text{none}> \\
\text{x : int} & \\
\text{y : int} & \\
\hline \\
f ((x + 1) * (y + 1)) & = (x + 1) * (y + 1) + 1
\end{align*}
\]
Rewriting Equational Lemmas

And the tactic

\[ f_{\text{eq}} \ ((x + 1) \ast (y + 1)). \]

transforms the goal

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
y & : \text{int} \\
\end{align*}
\]

\[ f \ ((x + 1) \ast (y + 1)) = (x + 1) \ast (y + 1) + 1 \]

into

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
y & : \text{int} \\
\end{align*}
\]

\[ (x + 1) \ast (y + 1) + 1 = (x + 1) \ast (y + 1) + 1 \]
Rewriting Equational Lemmas

As with apply, rewrite can often infer the parameters of the equational lemma, and we can also do rewriting from right-to-left by prepending a - . E.g., the tactic

   rewrite -f_eq -f_eq -f_eq

transforms the goal

   Type variables: <none>

   x : int
   y : int

   f (f x * f y) = (x + 1) * (y + 1) + 1

into

   Type variables: <none>

   x : int
   y : int

   f (f x * f y) = f (f x * f y)
Rewriting Equational Lemmas

The rewrite tactic can also be used with conditional equational lemmas like

```
lemma f_eq (x : int) :
  0 <= x => f x = x + 1
```

In this case,

```
rewrite f_eq.
```

transforms the goal

```
Type variables: <none>

x : int
y : int
ge0_x: 0 <= x
ge0_y: 0 <= y

--------------------------------------------
f (x + y) = x + y + 1
```

into
Rewriting Equational Lemmas

Type variables: <none>

\[ x : \text{int} \]
\[ y : \text{int} \]
\[ \text{ge0}_x : 0 \leq x \]
\[ \text{ge0}_y : 0 \leq y \]
---------------------------------------------
\[ 0 \leq x + y \]

and

Type variables: <none>

\[ x : \text{int} \]
\[ y : \text{int} \]
\[ \text{ge0}_x : 0 \leq x \]
\[ \text{ge0}_y : 0 \leq y \]
---------------------------------------------
\[ x + y + 1 = x + y + 1 \]
Rewriting Equational Lemmas

If we combine conditional equational lemmas $l_1$ and $l_2$ in a single use of `rewrite`, then $l_2$ is rewritten in the conclusion of every subgoal generated by the rewriting of $l_1$ in the conclusion of the original goal.

We can say, e.g.,

```
rewrite {3}\ l.
```

to rewrite $l$ in the current goal's conclusion in only the third applicable position.

We can also say

```
rewrite $l$ in $H$.
```

to rewrite $l$ is the assumption $H$. 

Rewriting Equational Lemmas

We can include //, /= and /# in rewriting, to apply trivial, simplify and smt(), respectively.

If an operator f has a concrete definition, e.g.,

\[ \text{op } f(x : \text{int}) = x \times 2 - 1. \]

Then rewrite /f substitutes f’s argument for its parameter(s) in its body (\(x \times 2 - 1\) in this case).
Rewriting Nonequational Lemmas

Rewrite can also be used with non-equational lemmas, rewriting the conclusion of the lemma (what we get after introducing all universally quantified variables and left sides of implications) with true. E.g., if we have

\[
\text{axiom } f\_ax (x : \text{int}) : 3 \leq x \Rightarrow f(x).
\]

then rewrite \( f\_ax \) transforms

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
y & : \text{int} \\
lt\_x\_y & : x < y \\
le\_3\_x & : 3 \leq x
\end{align*}
\]

\[
\vdash f(x) \land x < y + 1
\]
Rewriting Nonequational Lemmas

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
y & : \text{int} \\
\text{lt}_x_y & : x < y \\
\text{le}_3_x & : 3 \leq x
\end{align*}
\]

\[
\begin{align*}
3 \leq x
\end{align*}
\]

and

Type variables: <none>

\[
\begin{align*}
x & : \text{int} \\
y & : \text{int} \\
\text{lt}_x_y & : x < y \\
\text{le}_3_x & : 3 \leq x
\end{align*}
\]

\[
\begin{align*}
\text{true} \land x < y + 1
\end{align*}
\]
The progress tactic uses other tactics like application of introduction patterns and split to reduce the current goal to one of more subgoals. E.g., progress reduces

Type variables: <none>

\[ k' : \text{int} \]
\[ n' : \text{int} \]
\[ n : \text{int} \]
\[ r : \text{int} \]
\[ k : \text{int} \]

-----------------------------

\[ ((0 < k' \land n' ^ k' * r = n ^ k) \land 1 < k') \land f k' \Rightarrow \\
0 < g k' 2 \land (n' * n') ^ g k' 2 * r = n ^ k \]

to the two subgoals
Progress

Type variables: <none>

\[ k': \text{int} \]
\[ n': \text{int} \]
\[ n: \text{int} \]
\[ r: \text{int} \]
\[ k: \text{int} \]
\[ H: 0 < k' \]
\[ H0: n'^{k'} * r = n^k \]
\[ H1: 1 < k' \]
\[ H2: f k' \]

-------------------------------

\[ 0 < g k' 2 \]

and
Type variables: <none>

k’: int
n’: int
n : int
r : int
k : int
H : 0 < k’
H0: n’ ^ k’ * r = n ^ k
H1: 1 < k’
H2: f k’

--------------------------------------------
(n’ * n’) ^ g k’ 2 * r = n ^ k
Sequencing Tactics

If $t_1$ and $t_2$ are tactics, then $t_1; t_2$ applies $t_2$ to all the subgoals (if any) generated by running $t_1$. Sequencing groups to the left, so that $t_1; t_2; t_3$ means $(t_1; t_2); t_3$.

E.g., $t; \text{trivial}$ applies $\text{trivial}$ to every subgoal generated by running $t$. 