A Commentary for "The Joy of EasyCrypt"

A Beginner's Guide to Formal Verification of Cryptography

v 1.0

Ji, Yong-hyeon

(`hacker3740@kookmin.ac.kr`)

Department of Information Security, Cryptology, and Mathematics

College of Science and Technology

Kookmin University

January 24, 2025

Copyright

Changelog

v1.0	2025-01-20	Initial release:

Contents

1 Computer-Aided Cryptography

2 Introduction to EasyCrypt

3 EasyCrypt's Ambient Logic

The ambient logic in **EasyCrypt** serves as the foundation for all proof scripts. To gain a deeper understanding of its functionality, we will explore several examples.

As demonstrated in the earlier motivating example, formal proofs are constructed as a sequence of proof tactics. Up to this point, we have only encountered the admit tactic. In this chapter, we will expand on this by introducing additional basic tactics and applying them to simple mathematical properties of integers.

3.1 Basic Tactics and Theorem Proving

EasyCrypt features a typed expression language, meaning every declared entity must either have an explicitly defined type or a type that can be inferred from the context. As previously noted, **EasyCrypt** provides basic built-in data types, which can be accessed by importing the relevant theories into the current environment. For this particular file, we will work with the '`Int`' theory file. Additionally, to enable **EasyCrypt** to display all the proof goals during our work, we use a directive known as a pragma.

```
require import Int.
pragma Goals: printall.
```

Code 1: Imports and pragma

Typically, the initial steps in **EasyCrypt** scripts involve importing the required theories and setting the appropriate pragmas. Before delving into cryptography, it is essential to understand how to guide **EasyCrypt** in modifying proof goals and making progress. This process is facilitated by the use of tactics. In general, the proofs for lemmas in **EasyCrypt** follow a structured approach, as illustrated below:

```
lemma name (. . .) : (. . .) .
proof.
  tactic-1.
  ...
  tactic-n.
qed.
```

Code 2: Proof script form

3.1.1 Tactic: `trivial`

We begin by exploring some basic properties of integers and demonstrating how a few key tactics operate in **EasyCrypt**.

Reflexivity, the property that any integer is equal to itself, can be expressed mathematically as:

$$\forall n \in \mathbb{Z},\ n = n.$$

In EasyCrypt, this property can be stated as a lemma and proved as follows:

```
lemma int_refl: forall (n: int), n = n.
proof.
  trivial.
qed.
```

Code 3: Using the `trivial` tactic

Once the lemma is declared and evaluated, **EasyCrypt** populates the goal pane with the statement that needs to be proved. For this lemma, the goal pane displays the reflexivity property.

```
Current goal
Type variables: <none>
----------------------------
forall (n : int), n = n
```

The proof script begins with the `proof` keyword, after which **EasyCrypt** expects the application of tactics to close the goal. In this case, we use the trivial tactic to prove the lemma `int_refl`. Upon applying `trivial`, the goal pane is cleared since this tactic successfully resolves the goal. When all goals are resolved, the proof can be concluded with `qed`. This saves the lemma for future use, and **EasyCrypt** provides confirmation in the response pane as shown:

```
+ added lemma: `int_refl'
```

The `trivial` tactic attempts to solve the goal by applying a variety of internal tactics. While it may sometimes be unclear when it will succeed, the advantage of `trivial` is that it never fails—it either resolves the goal or leaves it unchanged. This makes it a safe and effective tactic to apply without any risk of disruption.

3.1.2 Tactic: apply

Once the lemma `int_refl` is proved, **EasyCrypt** stores it and allows us to use it to prove other lemmas. This is accomplished using the `apply` tactic. For example:

```
lemma nineteen_equal: 19 = 19.
proof.
  apply int_refl.
qed.
```

Code 4: Using the `apply` tactic

The `apply` tactic works by attempting to match the conclusion of the provided proof term with the goal's conclusion. If a match is found, the goal is replaced by the subgoals of the proof term.

In this case, **EasyCrypt** matches `int_refl` with the goal's conclusion, verifies the match, and replaces the goal with the subgoals required for `int_refl`. Since there are no additional subgoals to prove for `int_refl`, the proof is concluded successfully.

EasyCrypt also includes a library of predefined lemmas and axioms that can be used to facilitate proofs. For example, `addzC` and `addzA` are axioms related to the commutativity and associativity of integer addition, respectively. We can inspect these predefined lemmas and axioms using the `print` command. For instance, running `print addzC` and `print addzA` prompts **EasyCrypt** to display the following:

```
axiom nosmt addzC: forall (x y : int), x + y = y + x.
```

```
axiom nosmt addzA: forall (x y z : int), x + (y + z) = x + y + z.
```

3.1.3 Tactic: `simplify`

In proofs, it is common for tactics to produce goals that can be simplified. To handle such cases, we use the `simplify` tactic, which reduces the goal to its normal form using principles of lambda calculus. While the underlying mechanics of this process need not concern us, it is crucial to understand that **EasyCrypt** simplifies goals whenever possible, provided it has sufficient knowledge to do so. If the goal is already in normal form, the simplify tactic leaves it unchanged.

For example, the following illustration demonstrates the use of the `simplify` tactic:

```
lemma x_plus_comm (x: int): x + 2*3 = 6 + x.
proof.
  simplify.
  (* EC does the mathematical computation for us and simplifies the goal *)
  simplify.
  (* simplify doesn't fail, and leaves the goal unchanged *)
  trivial.
  (* trivial doesn't fail either, and leaves the goal unchanged *)
  apply addzC.
  (* Discharges the goal *)
qed.
```

Code 5: Using the `simplify` tactic

```
Current goal
Type variables: <none>

x: int
----------------------
x + 2 * 3 = 6 + x
```

$\xrightarrow{\texttt{simplify}}$

```
Current goal
Type variables: <none>

x: int
----------------------
x + 6 = 6 + x
```

$\xrightarrow{\texttt{simplify}}$

```
Current goal
Type variables: <none>

x: int
----------------------
x + 6 = 6 + x
```

3.1.4 Tactics: `move`, `rewrite`, `assumption`

Until now, we have worked with lemmas that did not involve any assumptions, apart from specifying variable types. However, in most cases, we will need to incorporate assumptions about variables into our proofs. These assumptions are treated as given and are introduced into the context using the `move =>` command, followed by the desired name for the assumption.

When such assumptions appear as goals, rather than explicitly applying them, we can use the `assumption` tactic to discharge the goal directly. This tactic instructs **EasyCrypt** to automatically search for assumptions in the context that match the goal and apply them.

Consider the following example, where we use the axiom `addz_gt0`, which is stated as follows:

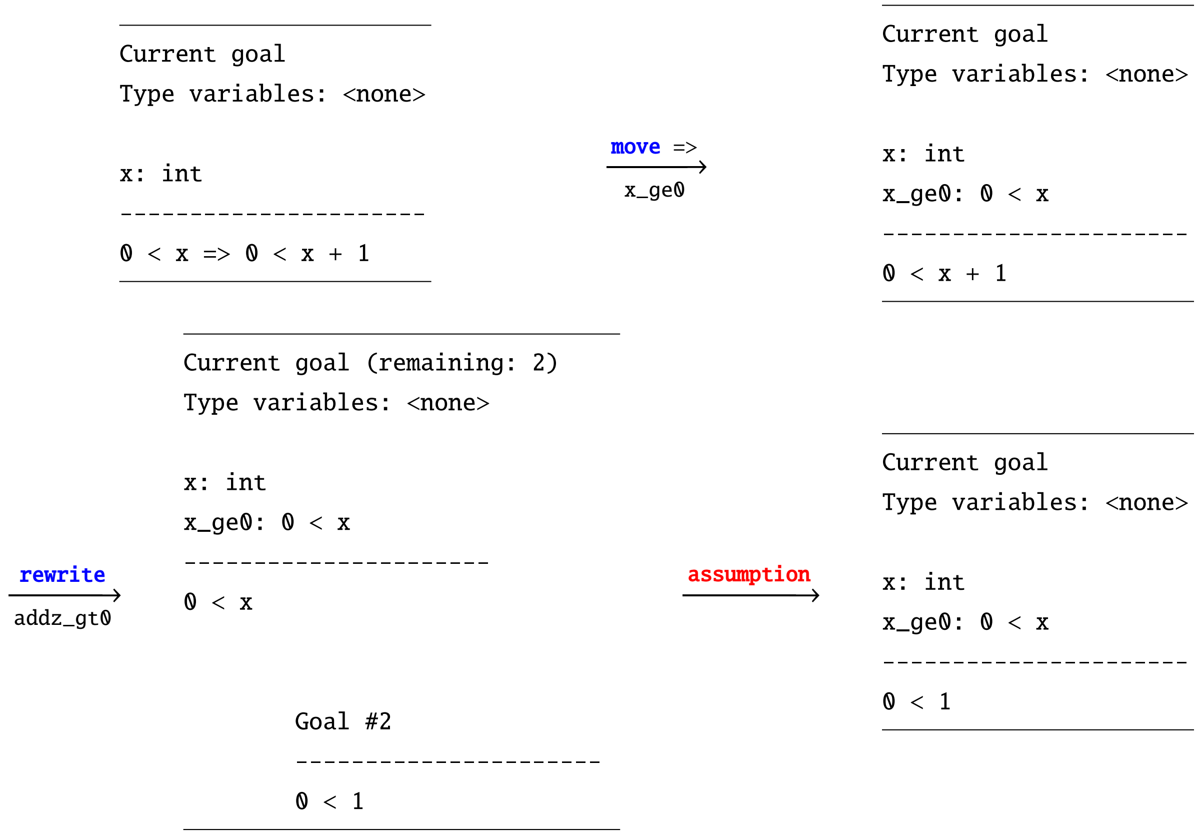
```
axiom nosmt addz_gt0: forall (x y : int), 0 < x => 0 < y => 0 < x + y.
```

Using this result, we construct the following proof script:

```
lemma x_pos (x: int): 0 < x => 0 < x+1.
proof.
  move => x_ge0.
  rewrite addz_gt0.
  (*
  "rewrite" simply rewrites the pattern provided, so in our case it
  rewrites our goal here (0 < x + 1), with the pattern that we provided
  which is addz_gt0, and then requires us to prove the assumptions of
  the pattern which are 0 < x and 0 < 1.
  *)
    (* Goal 1: 0 < x *)
    assumption.
    (* Goal 2: 0 < 1 *)
    trivial.
qed.
```

Code 6: Proof for `x_pos`

In this proof, the `rewrite` tactic modifies the goal by replacing the current expression with a specified pattern. For example, in this case, the goal `0 < x + 1` is rewritten using the pattern from `addz_gt0`, resulting in subgoals that require proving the assumptions `0 < x` and `0 < 1`.



```
Current goal
Type variables: <none>

x: int
----------------------
0 < x => 0 < x + 1
```

```
Current goal
Type variables: <none>

x: int
x_ge0: 0 < x
----------------------
0 < x + 1
```

```
Current goal (remaining: 2)
Type variables: <none>

x: int
x_ge0: 0 < x
----------------------
0 < x

        Goal #2
        ----------------------
        0 < 1
```

```
Current goal
Type variables: <none>

x: int
x_ge0: 0 < x
----------------------
0 < 1
```

Sometimes, a lemma or axiom may be rewritten to the goal, but the left-hand side (LHS) and right-hand side (RHS) of the expression might be flipped. To address this, we can rewrite the lemma or axiom in reverse by prefixing the lemma with a '-' symbol. This approach enables rewriting the sides as follows:

```
lemma int_assoc_rev (x y z: int): x + y + z = x + (y + z).
proof.
  print addzA.
  rewrite -addzA.
  trivial.
qed.
```

Code 7: Rewriting in reverse

These tactics constitute the foundational tools for theorem proving in **EasyCrypt**, particularly when working at the level of ambient logic.

It is worth noting that these tactics, while simple, include a variety of options and intricacies. For instance, the `move =>` tactic supports many introduction patterns, and the keyword move can be substituted with other tactics depending on the context. Expanding on these introduction patterns with clear examples would be a valuable addition to this discussion on basic tactics.

3.1.5 Commands: `search` and `print`

When working with theorems, it is often necessary to search through results already available in the environment. While we have encountered a few examples of printing in the content covered so far, it is useful to take a more detailed look at this aspect of **EasyCrypt**, as it is a feature we frequently rely on.

The `print` command outputs the requested information in the response pane. This command can be used to print various elements, such as types, modules, operations, and lemmas, by using the print keyword. For example:

```
print op (+).
(* abbrev (+) : int -> int -> int = CoreInt.add. *)
print op min.
(* op min (a b : int) : int = if a < b then a else b. *)
print axiom Int.fold0.
(* lemma fold0 ['a]: forall (f : 'a -> 'a) (a : 'a), fold f a 0 = a. *)
```

Code 8: Using the `print` command

Keywords serve as qualifiers and filters to refine the results. However, it is not mandatory to use qualifiers; printing without them will display broader results, while qualifiers help narrow the scope of the output.

The `search` command allows us to locate axioms and lemmas involving specific operators. This command takes arguments enclosed in braces to perform the search:

1. `[]` - Square brackets for unary operators
2. `()` - Round brackets for binary operators
3. Names of operators
4. Combination of these separated by a space

```
search [-].
search (+).
search ( * ).

search min.
(* Shows lemmas and axioms that include the operator "min". *)

search (+) (=) (=>).
(* Shows lemmas and axioms which have all the listed operators. *)
```

```
lemma exp_prod (x: real) (a b: int):
  x^(a*b) = x ^ a ^ b.
```

Code 9: Using the `search` command

3.1.6 External solvers: smt

It is essential to understand that **EasyCrypt** (**EC**) was primarily designed to handle cryptographic properties and more complex constructs. While it is possible to prove general mathematical theorems and claims in **EC**, the process can be cumbersome. To address the challenge of low-level logic and tactics, **EC** integrates with powerful automated tools through the `smt` tactic.

When the `smt` tactic is invoked, **EC** sends the current goal and its context to external **SMT** solvers, such as **Z3** and **Alt-ergo**, which are pre-configured for use with **EC**. If the **SMT** solver can resolve the goal, the `smt` tactic automatically discharges the specific subgoal. However, if the solver fails to find a solution, the proof remains incomplete, and the responsibility of resolving the goal falls back on the user.

For instance, consider the following results:

$$\forall x \in \mathbb{R},\ \forall a, b \in \mathbb{Z},\ x^{a \cdot b} = (x^a)^b,$$
$$\forall x \in \mathbb{R},\ \forall a, b \in \mathbb{Z},\ x \neq 0 \implies x^a \cdot x^b = x^{a+b}.$$

These can be proved in EC using the following script:

```
lemma exp_prod (x: real) (a b: int):
  x^(a*b) = x ^ a ^ b.
proof.
  search (^) ( * ) (=).
  by apply RField.exprM.
qed.

lemma exp_prod2 (x: real) (a b: int):
  x <> 0%r
  => x^a * x^b = x^(a + b).
proof.
  move => x_pos.
  search (^) (=).
  print  RField.exprD.
  rewrite -RField.exprD.
    assumption.
  trivial.
qed.
```

Code 10: Manual proof for `exp_prod` and `exp_prod2`

Alternatively, the proof can be simplified to:

```
lemma exp_prod_smt (x: real) (a b: int): x^(a*b) = x ^ a ^ b.
proof.
  smt.
qed.

lemma exp_prod2_smt (x: real) (a b: int): x <> 0%r => x^a * x^b = x^(a + b).
proof.
  smt.
qed.
```

Code 11: Using `smt` to prove `exp_prod_smt` and `exp_prod2_smt`

The key takeaway is that we will depend heavily on external solvers to handle a significant portion of the computational workload, particularly for results involving low-level mathematics.

This chapter concludes with an overview of ambient logic. We covered fundamental tactics such as **`apply`**, **`simplify`**, **`move`**, and **`rewrite`**, along with the usage of `search` and `print` commands. Additionally, we explored how to work with external solvers to streamline the proving process. In subsequent chapters, we will introduce more advanced tactics and techniques to expand on this foundation.

3.2 Ambient Logic's Exercies: Level 1

[1] The '`admit`' tactic resolves the current goal by assuming its truth without proof. Replace the '`admit`' tactic in the following lemma and prove it. (**Do not use '`smt`'.**)

```
lemma x_minus_equal (x: int): x - 10 = x - 9 - 1.
proof.
  admit.
qed.
```

Sol.

```
proof.
  trivial.
qed.
```

□

[2] Apply the '`split`' tactic to divide the disjunction into two separate goals. Then, utilize the earlier defined axioms to resolve these goals. (**Do not use '`smt`'.**)

```
lemma int_assoc_comm (x y z: int): x + (y + z) = (x + y) + z /\ x + y = y + x.
proof.
  admit.
qed.
```

Sol.

```
proof.
  split.
  (* Goal 1: x + (y + z) = x + y + z *)
  rewrite addzA.
  trivial.
  (* Goal 2: x + y = y + x *)
  rewrite addzC.
  trivial.
qed.
```

□

[3] **Do not use ‘smt’.**

```
require import AllCore.  (* for `%r' *)
require import RealExp.  (* for `ln' *)

lemma ln_prod (x y: real): 0%r < x  => 0%r < y => ln (x*y) = ln x + ln y.
proof.
  search (ln) (+).
  admit.
qed.
```

Sol.

```
proof.
  search (ln) (+).
  move => H1 H2.
  by apply lnM.
qed.
```

□

[4] **Do not use ‘smt’.**

```
require import AllCore.
require import IntDiv.

lemma mod_add (x y z: int): (x %% z + y %% z) %% z = (x + y) %% z.
proof.
  admit.
qed.
```

Sol.

```
proof.
  print IntDiv.
  print modzDm.
  by apply modzDm.
qed.
```

□

3.3 Ambient Logic's Exercies: Level 2

[1] We will now prove the equivalence:

$$\neg(a \vee b) \iff (\neg a \wedge \neg b)$$

```
require import AllCore.  (* core theories *)
(*
here's a lemma for learning some simple Ambient Logic proof techniques;
in fact, it's already in EasyCrypt's library with the same name;  and smt() will prove it
*)

lemma negb_or (a b : bool) :
        !(a \/ b) <=> !a /\ !b.
proof.
  admit.
qed.
```

Sol.

```
proof.
  split.
  (* proving ! (a \/ b) => !a /\ !b *)
  move => not_or.
  split.
  case a.
  move => a_true.
  simplify.
  have contrad : a \/ b.
    left.
    trivial.
  trivial.
  trivial.
  case b.
  move => b_true.
  simplify.
  have contrad : a \/ b.
    right.
    trivial.
  trivial.
  trivial.
  (* proving !a /\ !b => ! (a \/ b) *)
  move => and_not.
  elim and_not => a_false b_false.
  case (a \/ b).
  move => or.
  elim or.
  trivial.
  trivial.
  trivial.
qed.
```

□

References

[1] Rosulek, Mike. *The Joy of Cryptography*. 2019.

[2] Shah, Tejas Anil. *The Joy of EasyCrypt*. Master's thesis, University of Tartu, 2022.