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

CSE CRYPTO & SECURITY ENGINEERING Lab

암호 및 보안 공학 연구실

February 14, 2025

4.4 HL in EasyCrypt

4.4.1 Basic Hoare Triples

We introduce a module that defines two procedures for the programs.

```
module Func1 = {
  proc add_1 (x: int) : int = { return x + 1; }
  proc add_2 (x: int) : int = { x <- x + 2; return x; }
}.
```

Code 13: Function definitions (`Func1`)

A Hoare triple of the form $\{P\}\ C\ \{Q\}$, which is conventionally read as "if P holds prior to executing C, then Q holds afterward," is expressed in **EasyCrypt** as

```
hoare [C : P ==> Q]
```

in line with the usual definitions. Moreover, **EasyCrypt** records the return value of the program C in a special keyword called `res`. For instance, the Hoare triple

$$\{x = 1\}\ \texttt{Func1.add\_1}\ \{x = 2\}$$

is represented in **EasyCrypt** as follows:

```
lemma triple1: hoare [ Func1.add_1 : x = 1 ==> res = 2].
```

When working within Hoare logic or its variants, the proof goal typically takes a different form than that of ambient logic. For example, after invoking the `lemma triple1`, the resulting goal in **EasyCrypt** 's proof state exhibits precisely these Hoare-specific features.

```
Type variables: <none>

----------------------------------------------------------------------
pre = arg = 1
   Func1.add_1
post = res = 2
```

Code 14: Goal upon evaluating (`triple1`)

To advance the proof, we must inform **EasyCrypt** about the definition of `Func1.add_1`. This is accomplished by the `proc` tactic, which incorporates the procedure's body into the current

proof context. Since `Func1.add_1` only returns a value, applying **`proc`** replaces **`res`** with that return value and effectively leaves the program body empty for further analysis.

When reasoning about program correctness using Hoare Logic, we systematically analyze how program statements transform preconditions into postconditions, guided by axioms and lemmas. The process involves "consuming" program statements—applying rules to decompose the program until no statements remain. At this point, the goal transitions from a Hoare Logic (HL) goal to a statement in ambient logic, facilitated by the **`skip`** tactic.

Specifically, skip applies the following reasoning:

$$\frac{P \implies Q}{\{P\}\ \text{skip};\{Q\}}\ \textbf{skip}$$

where skip; denotes an empty program, and **`skip`** is the tactic that enacts this transition. As a result, we return to the standard ambient logic setting, allowing us to use all the tactics. The only subtlety is that moving from a Hoare logic goal to an ambient logic goal requires accounting for the memory qualifiers in which the program variables reside.

For instance, in this example, after applying **`skip`**, the resulting goal is:

```
forall &hr, x{hr} = 1 => x{hr} + 1 = 2,
```

where `&hr` denotes a particular memory state, and `x{hr}` is the value of `x` within that memory. Proving this is straightforward: we simply move the memory parameter into our assumptions by prepending the `&` character in the **`move`** `=>` tactic.

Putting these steps together yields the following complete proof for our simple example:

```
lemma triple1: hoare [ Func1.add_1 : x = 1 ==> res = 2].
proof.
  proc.
  skip.
  move => &m H1. (* &m moves memory to the environment *)
  subst. (* Substitutes variables from the assumptions *)
  trivial.
qed.
```

4.4.2 Automation, and Special Cases

4.4.3 Conditionals and Loops

4.5 Hoare Logic with AES

We describe the specifications, the corresponding Hoare triples, and the formal proof strategies used to show that the imperative specifications (for key expansion and AES rounds) are equivalent to their functional counterparts.

4.5.1 Overview

We consider an implementation of the AES algorithm that consists of several imperative procedures. In particular, we verify the following:

1. **Key Expansion.**

   The procedure '`Aes.keyExpansion`' is specified by the Hoare triple

   $$\{\, \texttt{key} = k \,\}\ \texttt{Aes.keyExpansion}\ \{\, \forall i,\ (0 \le i < 11 \implies \texttt{res}[i] = \texttt{key\_i}\ k\ i) \,\}.$$

   That is, when invoked with an initial key k, the result (an array of round keys) satisfies, for each i with $0 \le i < 11$, the equation $\texttt{res}[i] = \texttt{key\_i}(k, i)$.

```
hoare aes_keyExpansion k :
  Aes.keyExpansion : key = k
  ==>
  forall i, 0 <= i < 11 => res.[i] = key_i k i.
```

2. **AES Rounds.**

   The procedure '`Aes.aes_rounds`' is specified by the Hoare triple

   $$\{\, (\forall i\, (0 \le i < 11 \implies \texttt{rkeys}[i] = \texttt{key\_i}\ k\ i)) \wedge (\texttt{msg} = m) \,\}\ \texttt{Aes.aes\_rounds}\ \{\, \texttt{res} = \texttt{aes}\ k\ m \,\}.$$

   That is, given that the round keys are correctly computed (as indicated by the invariant $\forall i,\ (0 \le i < 11 \implies \texttt{rkeys}[i] = \texttt{key\_i}\ k\ i)$) and that the message is m, the final result equals the functional specification $\texttt{aes}\ k\ m$.

```
hoare aes_rounds k m :
  Aes.aes_rounds : (forall i, 0 <= i < 11 => rkeys.[i] = key_i k i) /\ msg = m
  ==>
  res = aes k m.
```

3. **Full AES Procedure.**

   The top-level procedure 'Aes.aes' is then specified by the Hoare triple

   $$\{\, (\texttt{key} = k) \wedge (\texttt{msg} = m) \,\}\ \texttt{Aes.aes}\ \{\, \texttt{res} = \texttt{aes}\ k\ m \,\}.$$

In other words, when provided with key k and message m, the AES implementation computes the correct result.

```
hoare aes_h k m :
  Aes.aes : key = k /\ msg = m
  ==>
  res = aes k m.
```

4.5.2 Formal Specification and Proofs

In our formalism, each Hoare triple is written as

$$[P : C \Longrightarrow Q]\,,$$

where P is the precondition, C is the command (or procedure), and Q is the postcondition. The following are the specifications, along with the outlines of their corresponding formal proofs.

4.5.3 Formal Proof Structure