EasyCrypt Library in Jasmin

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 21, 2025

File Structure

```
jasmin
├── compiler/
└── eclib/
    ├── AES.ec
    ├── JArray.ec
    ├── JLeakage.ec
    ├── JMemory.ec
    ├── JModel.ec
    ├── JModel_common.ec
    ├── JModel_m4.ec
    ├── JModel_x86.ec
    ├── JUtils.ec
    ├── JWord.ec
    ├── JWord_array.ec
    ├── Jslh.ec
    ├── Makefile
    ├── easycrypt.project
    └── tests.config
```

Copyright

Changelog

v1.0	2025-01-03	Initial release:

Contents

1 AES

```
require import List JArray JWord.
```

1.1 Operations on bytes and word

```
op Sbox : W8.t -> W8.t.
op InvSbox : W8.t -> W8.t.

axiom InvSboxK w : InvSbox (Sbox w) = w.
```

- $\texttt{Sbox} : \mathbb{F}_{2^8} \longrightarrow \mathbb{F}_{2^8}$
- $\texttt{InvSbox} : \mathbb{F}_{2^8} \longrightarrow \mathbb{F}_{2^8}$
- $\texttt{InvSbox}(\texttt{Sbox}(w)) = w$, where $w \in \mathbb{F}_{2^8}$.

```
op SubWord (w : W32.t) = map Sbox w.
op InvSubWord (w : W32.t) = map InvSbox w.

lemma InvSubWordK w : InvSubWord (SubWord w) = w.
proof.
  rewrite /SubWord /InvSubWord; apply W4u8.wordP => i hi.
  by rewrite !W4u8.mapbE 1,2:// InvSboxK.
qed.

op RotWord (w:W32.t) =
  W4u8.pack4 [w \bits8 1; w \bits8 2; w \bits8 3; w \bits8 0].
```

- Let $w = [w_0, w_1, w_2, w_3]$, where $w_i \in \mathbb{F}_{2^8}$. Then

$$\texttt{SubWord}(w) = [\texttt{Sbox}(w_0), \texttt{Sbox}(w_1), \texttt{Sbox}(w_2), \texttt{Sbox}(w_3)]$$

2 JUtils

```
require import AllCore IntDiv List Bool StdOrder.
        import IntOrder.
```

https://github.com/EasyCrypt/easycrypt
easycrypt/theories/core/AllCore.ec
easycrypt/theories/core/Bool.ec
easycrypt/theories/algebra/IntDiv.ec
easycrypt/theories/algebra/StdOrder.ec
easycrypt/theories/datatypes/List.ec

LEMMA: modz_comp

```
lemma modz_cmp m d : 0 < d => 0 <= m %% d < d.
proof. smt (edivzP). qed.
```

Statement. For two integers m and $d > 0$, the remainder of m divided by d satisfies:

$$0 \le m \bmod d < d.$$

Analysis. This property follows directly from the division algorithm:

$$m = q \cdot d + r, \quad 0 \le r < d$$

where $q = \lfloor m/d \rfloor$ and $r = m \bmod q$.

Proof Tactics. SMT solver with the pre-proved property `edivzP` (in `IntDiv`).

LEMMA: divz_cmp

```
lemma divz_cmp d i n : 0 < d => 0 <= i < n * d => 0 <= i %/ d < n.
proof.
  by move=> hd [hi1 hi2]; rewrite divz_ge0 // hi1 /= ltz_divLR.
qed.
```

Statement. For integers d, i, n where $d > 0$ and $0 \leq i < n \cdot d$, the integer division satisfies

$$0 \leq \frac{i}{d} < n.$$

Analysis. TBA

Proof Tactics. TBA

LEMMA: mulz_cmp_r

```
lemma mulz_cmp_r i m r : 0 < m => 0 <= i < r => 0 <= i * m < r * m.
proof.
  move=> h0m [h0i hir]; rewrite IntOrder.divr_ge0 //=; 1: by apply ltzW.
  by rewrite IntOrder.ltr_pmul2r.
qed.
```

Statement. TBA

Analysis. TBA

Proof Tactics. TBA

LEMMA: cmpW

```
lemma cmpW i d : 0 <= i < d => 0 <= i <= d.
proof. by move=> [h1 h2];split => // ?;apply ltzW. qed.
```

Statement. TBA

Analysis. TBA

Proof Tactics. TBA

LEMMA: le_modz

```
lemma le_modz m d : 0 <= m => m %% d <= m.
proof.
  move=> hm.
  have [ ->| [] hd]: d = 0 \/ d < 0 \/ 0 < d by smt().
  + by rewrite modz0.
  + by rewrite -modzN {2}(divz_eq m (-d)); smt (divz_ge0).
  by rewrite {2}(divz_eq m d); smt (divz_ge0).
qed.
```

Statement. TBA

Analysis. TBA

Proof Tactics. TBA

3 JArray

References

A Prelude

A.1 Logic

Principle of Functional Extensionality

```
axiom fun_ext ['a 'b] (f g:'a -> 'b): f = g <=> f == g.
```

The axiom asserts:

$$f = g \iff f == g$$

- Left-hand side ($f = g$):

  This refers to the equality of functions as mathematical objects. Two functions f and g are equal if they are identical, meaning they are the same function in every aspect.

- Right-hand side ($f == g$):

  This refers to pointwise equality: $f(x) = g(x)$ for all $x \in' a$

- Interpretation:

  The axiom establishes that two functions are equal as mathematical objects if and only if they produce the same output for every input. This is the essence of functional extensionality.

Set Theory	In classical set theory (ZFC), functional extensionality is implicitly satisfied because functions are defined as sets of ordered pairs: $$f = \{(x, f(x)) : x \in \mathrm{dom}(f)\}.$$ Thus, two functions are equal if and only if their values agree for every input.
Category Theory	In category theory, functional extensionality corresponds to the notion that morphisms (arrows) between objects are determined by their action on elements.
Constructive Mathematics	In constructive frameworks (e.g., type theory), functional extensionality may not hold by default, as functions can be defined by their computational behavior rather than just their input-output relations. In such settings, extensionality is often treated as an additional axiom.

Constructive Choice Function

```
op choiceb ['a] (P : 'a -> bool) (x0 : 'a) : 'a.

axiom choicebP ['a] (P : 'a -> bool) (x0 : 'a):
  (exists x, P x) => P (choiceb P x0).

axiom choiceb_dfl ['a] (P : 'a -> bool) (x0 : 'a):
  (forall x, !P x) => choiceb P x0 = x0.

lemma eq_choice ['a] (P Q : 'a -> bool) (x0 : 'a):
  (forall x, P x <=> Q x) => choiceb P x0 = choiceb Q x0.
proof. smt(fun_ext). qed.

axiom choice_dfl_irrelevant ['a] (P : 'a -> bool) (x0 x1 : 'a):
  (exists x, P x) => choiceb P x0 = choiceb P x1.
```

Definition of choiceb For a predicate P : 'a $\longrightarrow$ $\{\texttt{true}, \texttt{false}\}$ and a default element $x_0 \in$ 'a, it selects an element $x \in$ 'a such that $P(x) = \texttt{true}$, if such an x exists. Otherwise, it defaults to x_0. That is,

$$\texttt{choiceb}(P, x_0) = \begin{cases} x & : \exists x \in \texttt{'a} : P(x) = \texttt{true} \\ x_0 & : \forall x \in \texttt{'a} : P(x) = \texttt{false} \end{cases}$$

Axioms for choiceP If there exists an element x such that $P(x) = \texttt{true}$, then $\texttt{choiceb}(P, x_0)$ satisfies P, i.e., $P(\texttt{choiceb}(P, x_0)) = \texttt{true}$.

This axiom asserts that the choice function correctly selects an element from the subset defined by P, whenever such an element exists. It embodies the constructive aspect of the function.

choiceb_dfl If $P(x) = \texttt{false}$ for all $x \in$ 'a, then $\texttt{choiceb}(P, x_0)$ returns the default value x_0.

This axiom ensures that the function choiceb respects its fallback behavior when the subset defined by P is empty.

Equality of Choices If two predicates P and Q are logically equivalent (i.e., $P(x) \Leftrightarrow Q(x)$ for all $x \in$ 'a), then $\texttt{choiceb}(P, x_0) = \texttt{choiceb}(Q, x_0)$.

The proof relies on the extensionality of predicates: functions P and Q are equivalent if they have identical output for all inputs.

Axiom: Irrelevance of Default for Non-Empty Subsets If P is satisfied by some element x, the value of $\texttt{choiceb}(P, x_0)$ is independent of the default value x_0.

This axiom guarantees that when $\exists x \in \texttt{'a} : P(x) = \texttt{true}$, the choice function selects an element satisfying P, irrespective of the fallback x_0. This is because the fallback x_0 is only used when P is unsatisfiable.

This property reflects the **stability** of the constructive choice under changes to the fallback parameter, provided the primary condition is satisfied.

Summary These axioms and lemmas collectively define a constructive choice principle, a central concept in intuitionistic mathematics and proof systems. Unlike classical logic's unrestricted use of the axiom of choice, this constructive version ensures explicit constructability of the chosen element.

Application This construct is useful in formal verification for:

- Selecting elements satisfying a predicate (e.g., in search or optimization problems).
- Ensuring robust behavior under different edge cases (e.g., empty or singleton sets).
- Proving properties of algorithms that depend on choice.

B Algebra

B.1 IntDiv

```
op euclidef (m d : int) (qr : int * int) =
     m = qr.`1 * d + qr.`2
  /\ (d <> 0 => 0 <= qr.`2 < `|d|).

op edivn (m d : int) =
  if (d < 0 \/ m < 0) then (0, 0) else
    if d = 0 then (0, m) else choiceb (euclidef m d) (0, 0)
  axiomatized by edivn_def.

op edivz (m d : int) =
  let (q, r) =
    if 0 <= m then edivn m `|d| else
      let (q, r) = edivn (-(m+1)) `|d| in
      (- (q + 1), `|d| - 1 - r)
    in (signz d * q, r)
  axiomatized by edivz_def.

abbrev (%/) (m d : int) = (edivz m d).`1.
abbrev (%%) (m d : int) = (edivz m d).`2.

op (%|) (m d : int) = (d %% m = 0).
```

Quotient (%/)

Remainder (%%)

Divisibility (%|) $m \,\%|\, d \iff m \bmod d = 0.$

Applications The modular arithmetic constructs are essential for reasoning about cryptographic algorithms (e.g., RSA, Diffie-Hellman). Division and modulus operations are cornerstones for verifying number-theoretic properties.

```
lemma edivzP_r (m d : int): euclidef m d (edivz m d).
proof.
rewrite edivz_def; case: (0 <= m).
+ move=> ge0_m; case _: (edivn _ _) => q r E /=.
  case: (edivnP m `|d| _ _) => //; rewrite ?normr_ge0 E /= => mE.
  rewrite normr0P normr_id => lt_rd; split=> /= [|/lt_rd] //.
  by rewrite mulrAC -signVzE mE mulrC.
rewrite lerNgt /= => lt0_m; case _: (edivn _ _) => q r E /=.
case: (edivnP (-(m+1)) `|d| _ _) => /=; rewrite ?E /=.
  by rewrite oppr_ge0 -ltzE. by rewrite normr_ge0.
rewrite normr0P normr_id=> mE lt_rd; split=> /=; last first.
  move/lt_rd=> {lt_rd}[ge0_r lt_rd]; rewrite -addrA -opprD.
  rewrite subr_ge0 (addrC 1) -ltzE lt_rd /= ltr_snaddr //.
  by rewrite oppr_lt0 ltzS.
apply/(addIr 1)/oppr_inj; rewrite mE; case: (d = 0) => [|nz_d].
  by move=> ->; rewrite normr0 /=.
by rewrite mulrN mulNr -addrA opprD opprK mulrAC -signVzE #ring.
qed.

lemma edivzP (m d : int) :
  m = (m %/ d) * d + (m %% d) /\ (d <> 0 => 0 <= m %% d < `|d|).
proof. by case: (edivzP_r m d). qed.
```