

ABY3 Multiplication and Array Shuffling

Shuwei Hu

March 17, 2025

Abstract

We formalizes two protocols from a privacy-preserving machine-learning framework based on ABY3 [2], a particular three-party computation framework where inputs are systematically ‘reshared’ without being considered as privacy leakage. In particular, we consider the multiplication protocol [2] and the array shuffling protocol [1], both based on ABY3’s additive secret sharing scheme. We proved their security in the semi-honest setting under the ideal/real simulation paradigm. These two proof-of-concept opens the door to further verification of more protocols within the framework.

Contents

```
theory Finite-Number-Type
imports
  HOL-Library.Cardinality
begin
```

To avoid carrying the modulo all the time, we introduce a new type for integers $\{0..<L\}$ with the only restriction that the bound L must be greater than 1. It generalizes Z_{2^k} , the group considered in ABY3’s additive secret sharing scheme.

```
consts  $L :: nat$ 
```

```
specification ( $L$ ) L-gt-1:  $L > 1$ 
  <proof>
```

```
typedef  $natL = \{0..<int L\}$ 
  <proof>
```

```
setup-lifting type-definition-natL
```

```
instantiation  $natL :: comm-ring-1$ 
begin
```

```
lift-definition zero-natL ::  $natL$  is  $0$ 
```

```

    <proof>

lift-definition one-natL :: natL is 1
    <proof>

lift-definition uminus-natL :: natL ⇒ natL is λx. (-x) mod int L
    <proof>

lift-definition plus-natL :: natL ⇒ natL ⇒ natL is λx y. (x + y) mod int L
    <proof>

lift-definition minus-natL :: natL ⇒ natL ⇒ natL is λx y. (x - y) mod int L
    <proof>

lift-definition times-natL :: natL ⇒ natL ⇒ natL is λx y. (x * y) mod int L
    <proof>

instance
    <proof>

end
typ int

instantiation natL :: {distrib-lattice, bounded-lattice, linorder}
begin

lift-definition inf-natL :: natL ⇒ natL ⇒ natL is inf
    <proof>

lift-definition sup-natL :: natL ⇒ natL ⇒ natL is sup
    <proof>

lift-definition less-eq-natL :: natL ⇒ natL ⇒ bool is less-eq <proof>

lift-definition less-natL :: natL ⇒ natL ⇒ bool is less <proof>

lift-definition top-natL :: natL is int L-1
    <proof>

lift-definition bot-natL :: natL is 0
    <proof>

instance
    <proof>

end

instantiation natL :: semiring-modulo
begin

```

```

lift-definition divide-natL :: natL ⇒ natL ⇒ natL is divide
  ⟨proof⟩

lift-definition modulo-natL :: natL ⇒ natL ⇒ natL is modulo
  ⟨proof⟩

instance
  ⟨proof⟩

end

instance natL :: finite
  ⟨proof⟩

lemma natL-card[simp]:
   $CARD(natL) = L$ 
  ⟨proof⟩

end
theory Additive-Sharing
  imports
    CryptHOL.CryptHOL
    Finite-Number-Type
begin

datatype Role = Party1 | Party2 | Party3

lemma Role-exhaust'[dest!]:
   $r \neq Party1 \implies r \neq Party2 \implies r \neq Party3 \implies False$ 
  ⟨proof⟩

lemma Role-exhaust:
   $(r1::Role) \neq r2 \implies r2 \neq r3 \implies r3 \neq r1 \implies r=r1 \vee r=r2 \vee r=r3$ 
  ⟨proof⟩

type-synonym 'a sharing = Role ⇒ 'a

instance Role :: finite
  ⟨proof⟩

definition map-sharing :: ('a ⇒ 'b) ⇒ 'a sharing ⇒ 'b sharing where
  map-sharing f x = f ∘ x

definition get-party :: Role ⇒ 'a sharing ⇒ 'a where
  get-party r x = x r

lemma map-sharing-sel[simp]:
  get-party r (map-sharing f x) = f (get-party r x)

```

<proof>

definition *make-sharing'* :: *Role* \Rightarrow *Role* \Rightarrow *Role* \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a *sharing*
where

make-sharing' *r1 r2 r3 x1 x2 x3* = *undefined*(*r1:=x1, r2:=x2, r3:=x3*)

abbreviation *make-sharing* \equiv *make-sharing'* *Party1 Party2 Party3*

lemma *make-sharing'-sel*:

assumes *r1* \neq *r2* *r2* \neq *r3* *r3* \neq *r1*

shows

get-party r1 (make-sharing' r1 r2 r3 x1 x2 x3) = *x1*

get-party r2 (make-sharing' r1 r2 r3 x1 x2 x3) = *x2*

get-party r3 (make-sharing' r1 r2 r3 x1 x2 x3) = *x3*

<proof>

lemma *make-sharing-sel[simp]*:

shows

get-party Party1 (make-sharing x1 x2 x3) = *x1*

get-party Party2 (make-sharing x1 x2 x3) = *x2*

get-party Party3 (make-sharing x1 x2 x3) = *x3*

<proof>

primrec *next-role* **where**

next-role Party1 = *Party2*

| *next-role Party2* = *Party3*

| *next-role Party3* = *Party1*

primrec *prev-role* **where**

prev-role Party1 = *Party3*

| *prev-role Party2* = *Party1*

| *prev-role Party3* = *Party2*

lemma *next-sharing-neq-self[simp]*:

next-role r = r \longleftrightarrow *False* *r = next-role r* \longleftrightarrow *False*

<proof>

lemma *prev-sharing-neq-self[simp]*:

prev-role r = r \longleftrightarrow *False* *r = prev-role r* \longleftrightarrow *False*

<proof>

lemma *next-sharing-neq-prev[simp]*:

next-role r = prev-role r \longleftrightarrow *False* *prev-role r = next-role r* \longleftrightarrow *False*

<proof>

lemma *role-otherE*:

obtains *r* :: *Role* **where** *r0* \neq *r* *r* \neq *r1*

<proof>

lemma *make-sharing-sel-p2*:

shows

$get\text{-}party (prev\text{-}role\ r) (make\text{-}sharing' (prev\text{-}role\ r) r (next\text{-}role\ r) x1\ x2\ x3) =$
 $x1$
 $get\text{-}party\ r (make\text{-}sharing' (prev\text{-}role\ r) r (next\text{-}role\ r) x1\ x2\ x3) = x2$
 $get\text{-}party (next\text{-}role\ r) (make\text{-}sharing' (prev\text{-}role\ r) r (next\text{-}role\ r) x1\ x2\ x3) =$
 $x3$
<proof>

lemma *sharing-cases*[*cases type*]:

obtains $x1\ x2\ x3$ **where** $s = make\text{-}sharing\ x1\ x2\ x3$
<proof>

lemma *sharing-cases'*:

assumes $p1 \neq p2\ p2 \neq p3\ p3 \neq p1$
obtains $x1\ x2\ x3$ **where** $s = make\text{-}sharing'\ p1\ p2\ p3\ x1\ x2\ x3$
<proof>

lemma *make-sharing-collapse*[*simp*]:

$make\text{-}sharing (get\text{-}party\ Party1\ s) (get\text{-}party\ Party2\ s) (get\text{-}party\ Party3\ s) = s$
<proof>

lemma *sharing-eqI'*:

$\llbracket get\text{-}party\ Party1\ x = get\text{-}party\ Party1\ y;$
 $get\text{-}party\ Party2\ x = get\text{-}party\ Party2\ y;$
 $get\text{-}party\ Party3\ x = get\text{-}party\ Party3\ y \rrbracket$
 $\implies x = y$
<proof>

lemma *sharing-eqI*[*intro*]:

$(\bigwedge r. get\text{-}party\ r\ x = get\text{-}party\ r\ y) \implies x = y$
<proof>

abbreviation *prod-sharing* $:: 'a\ sharing \Rightarrow 'b\ sharing \Rightarrow ('a \times 'b)\ sharing$ **where**
 $prod\text{-}sharing \equiv corec\text{-}prod$

abbreviation *map-sharing2* $:: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ sharing \Rightarrow 'b\ sharing \Rightarrow 'c\ sharing$ **where**

$map\text{-}sharing2\ f\ xs\ ys \equiv map\text{-}sharing (case\text{-}prod\ f) (prod\text{-}sharing\ xs\ ys)$

lemma *prod-sharing-sel*[*simp*]:

$get\text{-}party\ r (prod\text{-}sharing\ x\ y) = (get\text{-}party\ r\ x, get\text{-}party\ r\ y)$
<proof>

lemma *prod-sharing-def-alt*:

$prod\text{-}sharing\ x\ y = make\text{-}sharing$
 $(get\text{-}party\ Party1\ x, get\text{-}party\ Party1\ y)$
 $(get\text{-}party\ Party2\ x, get\text{-}party\ Party2\ y)$
 $(get\text{-}party\ Party3\ x, get\text{-}party\ Party3\ y)$

<proof>

lemma *prod-sharing-map-sel*[simp]:

map-sharing fst (prod-sharing x y) = x

map-sharing snd (prod-sharing x y) = y

<proof>

definition *shift-sharing* :: 'a sharing \Rightarrow 'a sharing **where**

shift-sharing x = make-sharing (get-party Party3 x) (get-party Party1 x) (get-party Party2 x)

lemma *shift-sharing-def-alt*:

shift-sharing x = x \circ (make-sharing Party3 Party1 Party2)

<proof>

type-synonym 'a repsharing = ('a \times 'a) sharing

definition *reshare* :: 'a sharing \Rightarrow 'a repsharing **where**

reshare s = prod-sharing (shift-sharing s) s

lemma *reshare-sel*:

get-party Party1 (reshare s) = (get-party Party3 s, get-party Party1 s)

get-party Party2 (reshare s) = (get-party Party1 s, get-party Party2 s)

get-party Party3 (reshare s) = (get-party Party2 s, get-party Party3 s)

<proof>

definition *derep-sharing* :: 'a repsharing \Rightarrow 'a sharing **where**

derep-sharing = map-sharing snd

lemma *derep-sharing-sel*:

get-party r (derep-sharing s) = snd (get-party r s)

<proof>

lemma *derep-sharing-reshare*[simp]:

derep-sharing (reshare s) = s

<proof>

definition *map-repsharing* :: ('a \Rightarrow 'b) \Rightarrow 'a repsharing \Rightarrow 'b repsharing **where**

map-repsharing f s = reshare (map-sharing f (derep-sharing s))

lemma *map-repsharing-reshare*:

map-repsharing f (reshare s) = reshare (map-sharing f s)

<proof>

definition *valid-repsharing* :: 'a repsharing \Rightarrow bool **where**

valid-repsharing s \iff reshare (derep-sharing s) = s

lemma *valid-repsharingE*:

assumes *valid-repsharing* *s*
obtains *p* **where** $s = \text{reshare } p$
 ⟨*proof*⟩

lemma *map-repsharing-def-alt*:
 $\text{valid-repsharing } s \implies \text{map-repsharing } f \ s = \text{map-sharing } (\text{map-prod } f \ f) \ s$
 ⟨*proof*⟩

lemma *reshare-derep-sharing[simp]*:
 $\text{valid-repsharing } s \implies \text{reshare } (\text{derep-sharing } s) = s$
 ⟨*proof*⟩

lemma *valid-reshare[simp]*:
 $\text{valid-repsharing } (\text{reshare } s)$
 ⟨*proof*⟩

definition *make-repsharing* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ *repsharing* **where**
 $\text{make-repsharing } x1 \ x2 \ x3 = \text{reshare } (\text{make-sharing } x1 \ x2 \ x3)$

definition *reconstruct* :: natL *sharing* \Rightarrow natL **where**
 $\text{reconstruct } s = \text{get-party } \text{Party1} \ s + \text{get-party } \text{Party2} \ s + \text{get-party } \text{Party3} \ s$

definition *reconstruct-rep* :: natL *repsharing* \Rightarrow natL **where**
 $\text{reconstruct-rep } s = \text{reconstruct } (\text{derep-sharing } s)$

lemma *reconstruct-share[simp]*:
 $\text{reconstruct-rep } (\text{reshare } s) = \text{reconstruct } s$
 ⟨*proof*⟩

lemma *reconstrust-def'*:
assumes $r1 \neq r2 \ r2 \neq r3 \ r3 \neq r1$
shows $\text{reconstruct } s = \text{get-party } r1 \ s + \text{get-party } r2 \ s + \text{get-party } r3 \ s$
 ⟨*proof*⟩

lemma *reconstruct-make-sharing'[simp]*:
assumes $r1 \neq r2 \ r2 \neq r3 \ r3 \neq r1$
shows $\text{reconstruct } (\text{make-sharing}' \ r1 \ r2 \ r3 \ x1 \ x2 \ x3) = x1 + x2 + x3$
 ⟨*proof*⟩

lemma *reconstruct-make-repsharing[simp]*:
 $\text{reconstruct-rep } (\text{make-repsharing } x1 \ x2 \ x3) = x1 + x2 + x3$
 ⟨*proof*⟩

definition *valid-nat-repsharing* :: $\text{natL} \Rightarrow \text{natL}$ *repsharing* \Rightarrow *bool* **where**
 $\text{valid-nat-repsharing } v \ s \longleftrightarrow \text{reconstruct-rep } s = v \wedge \text{reshare } (\text{derep-sharing } s) = s$

lemma *comp-inj-on-iff'*:
 $\text{inj-on } (f' \circ f) \ A \longleftrightarrow \text{inj-on } f \ A \wedge \text{inj-on } f' \ (f \ ' \ A)$

<proof>

lemma *corec-prod-inject[simp]*:

corec-prod f g = corec-prod f' g' \longleftrightarrow f = f' \wedge g = g'

<proof>

lemma *inj-on-corec-prodI*:

inj-on f A \vee inj-on g A \implies inj-on ($\lambda x.$ corec-prod (f x) (g x)) A

<proof>

lemma *inj-on-reshare[simp]*:

inj-on reshare A

<proof>

lemma *inj-on-make-repsharing-eq-sharing*:

inj-on ($\lambda x.$ make-repsharing (f x) (g x) (h x)) A \longleftrightarrow inj-on ($\lambda x.$ make-sharing (f x) (g x) (h x)) A

<proof>

lemma *make-sharing'-inject[simp]*:

assumes *r1 \neq r2 r2 \neq r3 r3 \neq r1*

shows *make-sharing' r1 r2 r3 x1 x2 x3 = make-sharing' r1 r2 r3 y1 y2 y3 \longleftrightarrow x1=y1 \wedge x2=y2 \wedge x3=y3*

<proof>

lemma *make-sharing-inject[simp]*:

make-sharing x1 x2 x3 = make-sharing y1 y2 y3 \longleftrightarrow x1=y1 \wedge x2=y2 \wedge x3=y3

<proof>

lemma *reshare-inject[simp]*:

reshare a = reshare b \longleftrightarrow a = b

<proof>

lemma *make-repsharing-inject[simp]*:

make-repsharing x1 x2 x3 = make-repsharing y1 y2 y3 \longleftrightarrow x1=y1 \wedge x2=y2 \wedge x3=y3

<proof>

lemma *inj-on-make-sharingI*:

inj-on f A \vee inj-on g A \vee inj-on h A \implies inj-on ($\lambda x.$ make-sharing (f x) (g x) (h x)) A

<proof>

lemma *inj-on-make-repsharingI*:

inj-on f A \vee inj-on g A \vee inj-on h A \implies inj-on ($\lambda x.$ make-repsharing (f x) (g x) (h x)) A

<proof>

lemma *finite'-card-0*: $\text{finite}' A \longleftrightarrow \text{card } A \neq 0$
and *card-0-finite'*: $\text{card } A = 0 \longleftrightarrow \neg \text{finite}' A$
 ⟨*proof*⟩

lemma *spmf-of-set-bind*:
assumes *fin*: *finite* A
and *disj*: *disjoint-family-on* $f A$
and *card*: $\bigwedge a. a \in A \implies \text{card } (f a) = c$
shows $\text{spmf-of-set } (A \gg f) = \text{spmf-of-set } A \gg (\lambda x. \text{spmf-of-set } (f x))$

⟨*proof*⟩

lemma *ap-set-singleton*:
 $\text{ap-set } \{f\} A = f ' A$
 ⟨*proof*⟩

lemma *ap-set-Union*:
 $\text{ap-set } F A = (\bigcup f \in F. f ' A)$
 ⟨*proof*⟩

lemma *ap-set-curry*:
 $\text{ap-set } (\text{ap-set } F A) B = \text{ap-set } (\text{case-prod } ' F) (A \times B)$
 ⟨*proof*⟩

definition *share-nat* :: $\text{natL} \Rightarrow \text{natL}$ *sharing* *spmf* **where**
 $\text{share-nat } x = \text{spmf-of-set } (\text{reconstruct } - ' \{x\})$

definition *zero-sharing* :: natL *sharing* *spmf* **where**
 $\text{zero-sharing} = \text{share-nat } 0$

lemma *share-nat-def-calc'*:
assumes [*simp*]: $r1 \neq r2 \ r2 \neq r3 \ r3 \neq r1$
shows
 $\text{share-nat } x = (\text{do } \{$
 $\quad (x1, x2) \leftarrow \text{pair-spmf } (\text{spmf-of-set } UNIV) (\text{spmf-of-set } UNIV);$
 $\quad \text{let } x3 = x - (x1 + x2);$
 $\quad \text{return-spmf } (\text{make-sharing}' r1 r2 r3 x1 x2 x3)$
 $\quad \})$
 ⟨*proof*⟩

lemma *share-nat-def-calc*:
 $\text{share-nat } x = (\text{do } \{$

```

    (x1,x2) ← spmf-of-set UNIV;
    let x3 = x - (x1 + x2);
    return-spmf (make-sharing x1 x2 x3)
  })
  ⟨proof⟩

```

definition *repshare-nat* :: *natL* ⇒ *natL* *repsharing* *spmf* **where**

```

repshare-nat x = (do {
  (x1,x2) ← spmf-of-set UNIV;
  let x3 = x - (x1 + x2);
  return-spmf (make-repsharing x1 x2 x3)
})

```

lemma *repshare-nat-def-share*:

```

repshare-nat x = map-spmf reshare (share-nat x)
  ⟨proof⟩

```

lemma *repshare-nat-def-alt*:

```

repshare-nat x = spmf-of-set {reshare s | s. reconstruct s = x}
  ⟨proof⟩

```

lemma *valid-nat-repsharing-reshare*[*simp*]:

```

valid-nat-repsharing (reconstruct s) (reshare s)
  ⟨proof⟩

```

lemma *valid-nat-repsharingE*:

```

assumes valid-nat-repsharing x s
obtains s' where s = reshare s' and reconstruct s' = x
  ⟨proof⟩

```

lemma *repshare-nat-def-alt'*:

```

repshare-nat x = spmf-of-set (Collect (valid-nat-repsharing x))
  ⟨proof⟩

```

lemma *share-nat-valid*:

```

pred-spmf (valid-nat-repsharing x) (repshare-nat x)
  ⟨proof⟩

```

lemma *prod-sharing-map-fst-snd*[*simp*]:

```

prod-sharing (map-sharing fst s) (map-sharing snd s) = s
  ⟨proof⟩

```

end

theory *Spmf-Common*

imports *CryptHOL.CryptHOL*

begin

no-adhoc-overloading *Monad-Syntax.bind* ⇔ *bind-pmf*

lemma *mk-lossless-back-eq*:
 $scale\text{-}spmf\ (weight\text{-}spmf\ s)\ (mk\text{-}lossless\ s) = s$
 $\langle proof \rangle$

lemma *cond-spmf-enforce*:
 $cond\text{-}spmf\ sx\ (Collect\ A) = mk\text{-}lossless\ (enforce\text{-}spmf\ A\ sx)$
 $\langle proof \rangle$

definition *rel-scale-spmf* $s\ t \longleftrightarrow (mk\text{-}lossless\ s = mk\text{-}lossless\ t)$

lemma *rel-scale-spmf-refl*:
 $rel\text{-}scale\text{-}spmf\ s\ s$
 $\langle proof \rangle$

lemma *rel-scale-spmf-sym*:
 $rel\text{-}scale\text{-}spmf\ s\ t \implies rel\text{-}scale\text{-}spmf\ t\ s$
 $\langle proof \rangle$

lemma *rel-scale-spmf-trans*:
 $rel\text{-}scale\text{-}spmf\ s\ t \implies rel\text{-}scale\text{-}spmf\ t\ u \implies rel\text{-}scale\text{-}spmf\ s\ u$
 $\langle proof \rangle$

lemma *rel-scale-spmf-equiv*:
 $equivp\ rel\text{-}scale\text{-}spmf$
 $\langle proof \rangle$

lemma *spmf-eq-iff*: $p = q \longleftrightarrow (\forall i. spmf\ p\ i = spmf\ q\ i)$
 $\langle proof \rangle$

lemma *spmf-eq-iff-set*:
 $set\text{-}spmf\ a = set\text{-}spmf\ b \implies (\bigwedge x. x \in set\text{-}spmf\ b \implies spmf\ a\ x = spmf\ b\ x) \implies$
 $a = b$
 $\langle proof \rangle$

lemma *rel-scale-spmf-None*:
 $rel\text{-}scale\text{-}spmf\ s\ t \implies s = return\text{-}pmf\ None \longleftrightarrow t = return\text{-}pmf\ None$
 $\langle proof \rangle$

lemma *rel-scale-spmf-def-alt*:
 $rel\text{-}scale\text{-}spmf\ s\ t \longleftrightarrow (\exists k > 0. s = scale\text{-}spmf\ k\ t)$
 $\langle proof \rangle$

lemma *rel-scale-spmf-def-alt2*:
 $rel\text{-}scale\text{-}spmf\ s\ t \longleftrightarrow$
 $(s = return\text{-}pmf\ None \wedge t = return\text{-}pmf\ None)$
 $| (weight\text{-}spmf\ s > 0 \wedge weight\text{-}spmf\ t > 0 \wedge s = scale\text{-}spmf\ (weight\text{-}spmf\ s /$

weight-spmf t) *t*)
 (is ?lhs \longleftrightarrow ?rhs)
 <proof>

lemma *rel-scale-spmf-scale*:
 $r > 0 \implies \text{rel-scale-spmf } s \ t \implies \text{rel-scale-spmf } s \ (\text{scale-spmf } r \ t)$
 <proof>

lemma *rel-scale-spmf-mk-lossless*:
 $\text{rel-scale-spmf } s \ t \implies \text{rel-scale-spmf } s \ (\text{mk-lossless } t)$
 <proof>

lemma *rel-scale-spmf-eq-iff*:
 $\text{rel-scale-spmf } s \ t \implies \text{weight-spmf } s = \text{weight-spmf } t \implies s = t$
 <proof>

lemma *rel-scale-spmf-set-restrict*:
 $\text{finite } A \implies \text{rel-scale-spmf } (\text{restrict-spmf } (\text{spmof-of-set } A) \ B) \ (\text{spmof-of-set } (A \cap B))$
 <proof>

lemma *spmof-of-set-restrict-empty*:
 $A \cap B = \{\} \implies \text{restrict-spmf } (\text{spmof-of-set } A) \ B = \text{return-pmf None}$
 <proof>

lemma *spmof-of-set-restrict-scale*:
 $\text{finite } A \implies \text{restrict-spmf } (\text{spmof-of-set } A) \ B = \text{scale-spmf } (\text{card } (A \cap B) / \text{card } A) \ (\text{spmof-of-set } (A \cap B))$
 <proof>

lemma *min-em2*:
 $\text{min } a \ b = c \implies a \neq c \implies b = c$
 <proof>

lemma *weight-0-spmf*:
 $\text{weight-spmf } s = 0 \implies \text{spmof } s \ i = 0$
 <proof>

lemma *mk-lossless-scale-absorb*:
 $r > 0 \implies \text{mk-lossless } (\text{scale-spmf } r \ s) = \text{mk-lossless } s$
 <proof>

lemma *scale-spmf-None-iff*:
 $\text{scale-spmf } k \ s = \text{return-pmf None} \longleftrightarrow k \leq 0 \vee s = \text{return-pmf None}$
 <proof>

lemma *spmof-of-pmf-the*:
 $\text{lossless-spmf } s \implies \text{spmof-of-pmf } (\text{map-pmf the } s) = s$
 <proof>

lemma *lossless-mk-lossless*:

$s \neq \text{return-pmf None} \implies \text{lossless-spmf (mk-lossless s)}$
<proof>

definition *pmf-of-spmf where*

$\text{pmf-of-spmf } p = \text{map-pmf the (mk-lossless } p)$

lemma *scale-weight-spmf-of-pmf*:

$p = \text{scale-spmf (weight-spmf } p) (\text{spmof-of-pmf (pmf-of-spmf } p))$
<proof>

lemma *pmf-spmf*:

$\text{pmf-of-spmf (spmof-of-pmf } p) = p$
<proof>

lemma *spmof-pmf*:

$\text{lossless-spmf } p \implies \text{spmof-of-pmf (pmf-of-spmf } p) = p$
<proof>

lemma *pmf-of-spmf-scale-spmf*:

$r > 0 \implies \text{pmf-of-spmf (scale-spmf } r \text{ } p) = \text{pmf-of-spmf } p$
<proof>

lemma *nonempty-spmf-weight*:

$p \neq \text{return-pmf None} \iff \text{weight-spmf } p > 0$
<proof>

lemma *pmf-of-spmf-mk-lossless*:

$\text{pmf-of-spmf (mk-lossless } p) = \text{pmf-of-spmf } p$
<proof>

lemma *spmof-pmf'*:

$p \neq \text{return-pmf None} \implies \text{spmof-of-pmf (pmf-of-spmf } p) = \text{mk-lossless } p$
<proof>

lemma *rel-scale-spmf-cond-UNIV*:

$\text{rel-scale-spmf } s (\text{cond-spmf } s \text{ UNIV})$
<proof>

lemma *set-pmf* $p \cap g \neq \{\}$ $\implies \text{pmf-prob } p (f \cap g) = \text{pmf-prob (cond-pmf } p \text{ } g) f$

$* \text{pmf-prob } p \text{ } g$
<proof>

lemma *bayes*:

$\text{set-pmf } p \cap A \neq \{\} \implies \text{set-pmf } p \cap B \neq \{\} \implies$
 $\text{measure-pmf.prob (cond-pmf } p \text{ } A) B$
 $= \text{measure-pmf.prob (cond-pmf } p \text{ } B) A * \text{measure-pmf.prob } p \text{ } B / \text{measure-pmf.prob } p \text{ } A$

<proof>

definition *spmf-prob* :: 'a *spmf* \Rightarrow 'a *set* \Rightarrow *real* **where**
spmf-prob *p* = *Sigma-Algebra.measure* (*measure-spmf* *p*)

lemma *spmf-prob = measure-measure-spmf*
<proof>

lemma *spmf-prob-pmf*:
spmf-prob *p* *A* = *pmf-prob* *p* (*Some* ' *A*)
<proof>

lemma *bayes-spmf*:
spmf-prob (*cond-spmf* *p* *A*) *B*
= *spmf-prob* (*cond-spmf* *p* *B*) *A* * *spmf-prob* *p* *B* / *spmf-prob* *p* *A*
<proof>

lemma *spmf-prob-pmf-of-spmf*:
spmf-prob *p* *A* = *weight-spmf* *p* * *pmf-prob* (*pmf-of-spmf* *p*) *A*
<proof>

lemma *cond-spmf-Int*:
cond-spmf (*cond-spmf* *p* *A*) *B* = *cond-spmf* *p* (*A* \cap *B*)
<proof>

lemma *cond-spmf-prob*:
spmf-prob *p* (*A* \cap *B*) = *spmf-prob* (*cond-spmf* *p* *A*) *B* * *spmf-prob* *p* *A*
<proof>

definition *empty-spmf* = *return-pmf* *None*

lemma *spmf-prob-empty*:
spmf-prob *empty-spmf* *A* = 0
<proof>

definition *le-spmf* :: 'a *spmf* \Rightarrow 'a *spmf* \Rightarrow *bool* **where**
le-spmf *p* *q* \longleftrightarrow ($\exists k \leq 1. p = \text{scale-spmf } k \text{ } q$)

definition *lt-spmf* :: 'a *spmf* \Rightarrow 'a *spmf* \Rightarrow *bool* **where**
lt-spmf *p* *q* \longleftrightarrow ($\exists k < 1. p = \text{scale-spmf } k \text{ } q$)

lemma *class.order-bot empty-spmf le-spmf lt-spmf*
<proof>

lemma *spmf-prob-cond-Int*:
spmf-prob (*cond-spmf* *p* *C*) (*A* \cap *B*)
= *spmf-prob* (*cond-spmf* *p* (*B* \cap *C*)) *A* * *spmf-prob* (*cond-spmf* *p* *C*) *B*
<proof>

lemma *cond-spmf-mk-lossless*:

$cond\text{-}spmf\ (mk\text{-}lossless\ p)\ A = cond\text{-}spmf\ p\ A$

<proof>

primrec *sequence-spmf* :: 'a spmf list \Rightarrow 'a list spmf **where**

$sequence\text{-}spmf\ [] = return\text{-}spmf\ []$

| $sequence\text{-}spmf\ (x\#xs) = map\text{-}spmf\ (case\text{-}prod\ Cons)\ (pair\text{-}spmf\ x\ (sequence\text{-}spmf\ xs))$

lemma *set-sequence-spmf*:

$set\text{-}spmf\ (sequence\text{-}spmf\ xs) = \{l.\ list\text{-}all2\ (\lambda x\ s.\ x \in set\text{-}spmf\ s)\ l\ xs\}$

<proof>

lemma *map-spmf-map-sequence*:

$map\text{-}spmf\ (map\ f)\ (sequence\text{-}spmf\ xs) = sequence\text{-}spmf\ (map\ (map\text{-}spmf\ f)\ xs)$

<proof>

lemma *sequence-map-return-spmf*:

$sequence\text{-}spmf\ (map\ return\text{-}spmf\ xs) = return\text{-}spmf\ xs$

<proof>

lemma *sequence-bind-cong*:

$[\![xs=ys;\ \bigwedge y.\ length\ y = length\ ys \implies f\ y = g\ y]\!] \implies bind\text{-}spmf\ (sequence\text{-}spmf\ xs)\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ ys)\ g$

<proof>

lemma *bind-spmf-sequence-replicate-cong*:

$(\bigwedge l.\ length\ l = n \implies f\ l = g\ l)$

$\implies bind\text{-}spmf\ (sequence\text{-}spmf\ (replicate\ n\ x))\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ (replicate\ n\ x))\ g$

<proof>

lemma *bind-spmf-sequence-map-cong*:

$(\bigwedge l.\ length\ l = length\ x \implies f\ l = g\ l)$

$\implies bind\text{-}spmf\ (sequence\text{-}spmf\ (map\ m\ x))\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ (map\ m\ x))\ g$

<proof>

lemma *lossless-pair-spmf-iff*:

$lossless\text{-}spmf\ (pair\text{-}spmf\ a\ b) \longleftrightarrow lossless\text{-}spmf\ a \wedge lossless\text{-}spmf\ b$

<proof>

lemma *lossless-sequence-spmf*:

$(\bigwedge x.\ x \in set\ xs \implies lossless\text{-}spmf\ x) \implies lossless\text{-}spmf\ (sequence\text{-}spmf\ xs)$

<proof>

end

theory *Sharing-Lemmas*

imports

Additive-Sharing

begin

lemma *share-nat-2party-uniform*:

$p \neq q \implies \text{map-spmf } (\lambda s. (\text{get-party } p \ s, \text{get-party } q \ s)) \ (\text{share-nat } x) = \text{spmof-of-set UNIV}$
<proof>

lemma *share-nat-party-uniform*:

$\text{map-spmf } (\text{get-party } p) \ (\text{share-nat } x) = \text{spmof-of-set UNIV}$
<proof>

definition *is-uniform-sharing* :: *natL sharing spmf* \implies *bool* **where**

is-uniform-sharing $s \iff (\exists x :: \text{natL spmf}. s = \text{bind-spmf } x \ \text{share-nat})$

definition *is-uniform-sharing2* :: (*natL sharing* \times *natL sharing*) *spmf* \implies *bool*
where

is-uniform-sharing2 $s \iff (\exists xy :: (\text{natL} \times \text{natL}) \ \text{spmf}.$
 $s = \text{bind-spmf } xy \ (\lambda(x,y). \text{pair-spmf } (\text{share-nat } x) \ (\text{share-nat } y)))$

lemma *share-nat-uniform*:

is-uniform-sharing (*share-nat* x)
<proof>

lemma *share-nat-lossless*:

lossless-spmf (*share-nat* x)
<proof>

lemma *uniform-sharing2*:

is-uniform-sharing $s \implies p1 \neq p2 \implies \text{map-spmf } (\lambda x. (\text{get-party } p1 \ x, \text{get-party } p2 \ x)) \ s = \text{scale-spmf } (\text{weight-spmf } s) \ (\text{spmof-of-set UNIV})$
<proof>

lemma *uniform-sharing*:

is-uniform-sharing $s \implies \text{map-spmf } (\text{get-party } p) \ s = \text{scale-spmf } (\text{weight-spmf } s)$
(*spmof-of-set UNIV*)
<proof>

lemma *uniform-sharing'*:

$\llbracket \text{is-uniform-sharing } s; \text{lossless-spmf } s \rrbracket \implies \text{map-spmf } (\text{get-party } p) \ s = \text{spmof-of-set UNIV}$
<proof>

lemma *zero-masking-uniform*:

$p \neq q \implies (\text{map-spmf } ((\lambda t. (\text{get-party } p \ t, \text{get-party } q \ t)) \circ \text{map-sharing2 } (+) \ s) \ \text{zero-sharing}) = \text{spmof-of-set UNIV}$

<proof>

lemma *sharing-eqI2*[*consumes 3*]:

assumes $p1 \neq p2 \ p2 \neq p3 \ p3 \neq p1 \ \wedge p. p \in \{p1, p2, p3\} \implies \text{get-party } p \ s = \text{get-party } p \ t$
shows $s = t$
<proof>

lemma *sharing-map*[*simp*]:

assumes $p1 \neq p2 \ p2 \neq p3 \ p3 \neq p1$
shows $\text{map-sharing } f \ (\text{make-sharing}' \ p1 \ p2 \ p3 \ x1 \ x2 \ x3) = \text{make-sharing}' \ p1 \ p2 \ p3 \ (f \ x1) \ (f \ x2) \ (f \ x3)$
<proof>

lemma *sharing-prod*[*simp*]:

assumes $p1 \neq p2 \ p2 \neq p3 \ p3 \neq p1$
shows $\text{prod-sharing } (\text{make-sharing}' \ p1 \ p2 \ p3 \ x1 \ x2 \ x3) \ (\text{make-sharing}' \ p1 \ p2 \ p3 \ y1 \ y2 \ y3) = \text{make-sharing}' \ p1 \ p2 \ p3 \ (x1, y1) \ (x2, y2) \ (x3, y3)$
<proof>

lemma *add-sharing-inj*:

inj (*map-sharing2* (+) (*s* :: *natL sharing*))
<proof>

lemma *add-sharing-surj*:

surj (*map-sharing2* (+) (*s* :: *natL sharing*))
<proof>

lemma *sharing-add-inv-eq-minus*:

Hilbert-Choice.inv (*map-sharing2* (+) (*s*::*natL sharing*)) *t* = *map-sharing2* (-) *t s*
<proof>

lemma *zero-masking-eq-share-nat*:

map-spmf (*map-sharing2* (+) (*s* :: *natL sharing*)) *zero-sharing* = *share-nat* (*reconstruct s*)
<proof>

lemma *inv-uniform'*:

assumes $ss: s \subseteq U$ **and** *inj*: *inj-on* *f* *U*
shows $\text{map-spmf } (\lambda x. (x, f \ x)) \ (\text{spmf-of-set } s) = \text{map-spmf } (\lambda y. (\text{Hilbert-Choice.inv-into } U \ f \ y, y)) \ (\text{spmf-of-set } (f \ ' \ s))$
<proof>

lemma *inv-uniform*:

inj $f \implies \text{map-spmf } (\lambda x. (x, f \ x)) \ (\text{spmf-of-set } s) = \text{map-spmf } (\lambda y. (\text{Hilbert-Choice.inv } f \ y, y)) \ (\text{spmf-of-set } (f \ ' \ s))$
<proof>

lemma reconstruct-plus:

$reconstruct (map-sharing2 (+) x y) = reconstruct x + reconstruct y$
 $\langle proof \rangle$

lemma reconstruct-minus:

$reconstruct (map-sharing2 (-) x y) = reconstruct x - reconstruct y$
 $\langle proof \rangle$

lemma plus-reconstruct:

$map-sharing2 (+) x \text{ ` } reconstruct - \text{ ` } \{y\} = reconstruct - \text{ ` } \{reconstruct x + y\}$
 $\langle proof \rangle$

lemma inv-zero-sharing:

$map-*spmf* (\lambda\zeta. (\zeta, map-sharing2 (+) x \zeta)) zero-sharing = map-*spmf* (\lambda y. (map-sharing2 (-) y x, y)) (share-nat (reconstruct x))$
 $\langle proof \rangle$

lemma hoist-map-*spmf*:

$(do \{x \leftarrow s; g x (f x)\}) = (do \{(x,y) \leftarrow map-*spmf* (\lambda x. (x, f x)) s; g x y\})$
 $\langle proof \rangle$

lemma hoist-map-*spmf*':

$(do \{x \leftarrow s; g x (f x)\}) = (do \{(y,x) \leftarrow map-*spmf* (\lambda x. (f x, x)) s; g x y\})$
 $\langle proof \rangle$

definition HOIST-TAG $x = x$

lemmas hoist-tag = HOIST-TAG-def[symmetric]

lemma tagged-hoist-map-*spmf*:

$(do \{x \leftarrow s; g x (HOIST-TAG (f x))\}) = (do \{(x,y) \leftarrow map-*spmf* (\lambda x. (x, f x)) s; g x y\})$
 $\langle proof \rangle$

lemma get-prev-sharing[simp]:

$get-party p (shift-sharing s) = get-party (prev-role p) s$
 $\langle proof \rangle$

lemma shift-sharing-make-sharing:

$shift-sharing (make-sharing x1 x2 x3) = make-sharing x3 x1 x2$
 $\langle proof \rangle$

lemma reconstruct-share-nat:

$map-*spmf* (\lambda xs. (xs, reconstruct xs)) (share-nat x) = map-*spmf* (\lambda xs. (xs, x)) (share-nat x)$ **for x**
 $\langle proof \rangle$

lemma weight-share-nat:

$weight-*spmf* (share-nat x) = 1$

<proof>

end

theory *Shuffle*

imports

CryptHOL.CryptHOL

Additive-Sharing

Spmf-Common

Sharing-Lemmas

begin

This is the formalization of the array shuffling protocol defined in [1] adapted for the ABY3 sharing scheme. For the moment, we assume an oracle that generates uniformly distributed permutations, instead of instantiating it with e.g. Fischer-Yates algorithm.

no-notation (*ASCII*) *comp* (**infixl** *<o>* 55)

unbundle *no m-inv-syntax*

no-adhoc-overloading *Monad-Syntax.bind* \Rightarrow *bind-pmf*

fun *shuffleF* :: *natL sharing list* \Rightarrow *natL sharing list spmf* **where**
shuffleF xsl = *spmf-of-set (permutations-of-multiset (mset xsl))*

type-synonym *zero-sharing* = *natL sharing list*

type-synonym *party2-data* = *natL list*

type-synonym *party01-permutation* = *nat* \Rightarrow *nat*

type-synonym *phase-msg* = *zero-sharing* \times *party2-data* \times *party01-permutation*

type-synonym *role-msg* = (*natL list* \times *natL list* \times *natL list*) \times *party2-data* \times (*party01-permutation* \times *party01-permutation*)

definition *aby3-stack-sharing* :: *Role* \Rightarrow *natL sharing* \Rightarrow *natL sharing* **where**

aby3-stack-sharing r s = *make-sharing'* *r (next-role r) (prev-role r)*
(get-party r s)
(get-party (next-role r) s + get-party (prev-role r) s)
0

definition *aby3-do-permute* :: *Role* \Rightarrow *natL sharing list* \Rightarrow (*phase-msg* \times *natL sharing list*) *spmf* **where**

aby3-do-permute r x = (*do* {
 let n = length x;
 $\zeta \leftarrow$ *sequence-spmf (replicate n zero-sharing);*
 $\pi \leftarrow$ *spmf-of-set { π . π permutes {..*n*} };*
 let x2 = map (get-party (prev-role r)) x;
 let y' = map (aby3-stack-sharing r) x;
 let y = map2 (map-sharing2 (+)) (permute-list π y') ζ ;
 let msg = (ζ , x2, π);

```

return-spmf (msg, y)
})

```

definition *aby3-shuffleR* :: natL sharing list ⇒ (role-msg sharing × natL sharing list) spmf **where**

```

aby3-shuffleR x = (do {
  ((ζ a, x', π a), a) ← aby3-do-permute Party1 x;    — 1st round
  ((ζ b, a', π b), b) ← aby3-do-permute Party2 a;    — 2nd round
  ((ζ c, b', π c), c) ← aby3-do-permute Party3 b;    — 3rd round
  let msg1 = ((map (get-party Party1) ζ a, map (get-party Party1) ζ b, map
    (get-party Party1) ζ c), b', π a, π c);
  let msg2 = ((map (get-party Party2) ζ a, map (get-party Party2) ζ b, map
    (get-party Party2) ζ c), x', π a, π b);
  let msg3 = ((map (get-party Party3) ζ a, map (get-party Party3) ζ b, map
    (get-party Party3) ζ c), a', π b, π c);
  let msg = make-sharing msg1 msg2 msg3;
  return-spmf (msg, c)
})

```

definition *aby3-shuffleF* :: natL sharing list ⇒ natL sharing list spmf **where**

```

aby3-shuffleF x = (do {
  π ← spmf-of-set {π. π permutes {..

```

definition *S1* :: natL list ⇒ natL list ⇒ role-msg spmf **where**

```

S1 x1 yc1 = (do {
  let n = length x1;

  π a ← spmf-of-set {π. π permutes {..

```

— round 1

```

let ζ a1 = map2 (−) ya1 (permute-list π a x1);

```

— round 2

```

let ζ b1 = yb1;

```

— round 3
 let $b' = yb2$;
 let $\zeta c1 = \text{map2 } (-) (yc1) (\text{permute-list } \pi c (\text{map2 } (+) yb1 yb2))$; —
 non-free message

let $\text{msg1} = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c)$;
 return-spmf msg1
 })

definition $S2 :: \text{natL list} \Rightarrow \text{natL list} \Rightarrow \text{role-msg spmf}$ **where**

$S2\ x2\ yc2 = (\text{do } \{$
 let $n = \text{length } x2$;
 $x3 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } UNIV))$;

$\pi a \leftarrow \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\}$;
 $\pi b \leftarrow \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\}$;

$ya2::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } UNIV))$;
 $yb2::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } UNIV))$;

— round 1
 let $x' = x3$;
 let $\zeta a2 = \text{map2 } (-) ya2 (\text{permute-list } \pi a (\text{map2 } (+) x2 x3))$;

— round 2
 let $\zeta b2 = \text{map2 } (-) yb2 (\text{permute-list } \pi b ya2)$;

— round 3
 let $\zeta c2 = yc2$; — non-free message

let $\text{msg2} = ((\zeta a2, \zeta b2, \zeta c2), x', \pi a, \pi b)$;
 return-spmf msg2
 })

definition $S3 :: \text{natL list} \Rightarrow \text{natL list} \Rightarrow \text{role-msg spmf}$ **where**

$S3\ x3\ yc3 = (\text{do } \{$
 let $n = \text{length } x3$;

$\pi b \leftarrow \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\}$;
 $\pi c \leftarrow \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\}$;

$ya3::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } UNIV))$;
 $ya1::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } UNIV))$;

$yb3::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } UNIV))$;

— round 1
 $let \zeta a3 = ya3;$

— round 2
 $let a' = ya1;$
 $let \zeta b3 = map2 (-) yb3 (permute-list \pi b (map2 (+) ya3 ya1));$

— round 3
 $let \zeta c3 = map2 (-) yc3 (permute-list \pi c yb3);$ — non-free message

$let msg3 = ((\zeta a3, \zeta b3, \zeta c3), a', \pi b, \pi c);$
 $return-spmf msg3$

})

definition $S :: Role \Rightarrow natL\ list \Rightarrow natL\ list \Rightarrow role\text{-}msg\ spmf$ **where**
 $S\ r = get\text{-}party\ r\ (make\text{-}sharing\ S1\ S2\ S3)$

definition $is\text{-}uniform\text{-}sharing\text{-}list :: natL\ sharing\ list\ spmf \Rightarrow bool$ **where**
 $is\text{-}uniform\text{-}sharing\text{-}list\ xss \longleftrightarrow (\exists\ xs.\ xss = bind\text{-}spmf\ xs\ (sequence\text{-}spmf \circ map\ share\text{-}nat))$

lemma $case\text{-}prod\text{-}nesting\text{-}same:$
 $case\text{-}prod\ (\lambda a\ b.\ f\ (case\text{-}prod\ g\ x)\ a\ b)\ x = case\text{-}prod\ (\lambda a\ b.\ f\ (g\ a\ b)\ a\ b)\ x$
 $\langle proof \rangle$

lemma $zip\text{-}map\text{-}map\text{-}same:$
 $map\ (\lambda x.\ (f\ x,\ g\ x))\ xs = zip\ (map\ f\ xs)\ (map\ g\ xs)$
 $\langle proof \rangle$

lemma $dup\text{-}map\text{-}eq:$
 $length\ xs = length\ ys \implies (xs,\ map2\ f\ ys\ xs) = (\lambda xys.\ (map\ fst\ xys,\ map\ snd\ xys))$
 $(map2\ (\lambda x\ y.\ (y,\ f\ x\ y))\ ys\ xs)$
 $\langle proof \rangle$

abbreviation $map2\text{-}spmf\ f\ xs\ ys \equiv map\text{-}spmf\ (case\text{-}prod\ f)\ (pair\text{-}spmf\ xs\ ys)$
abbreviation $map3\text{-}spmf\ f\ xs\ ys\ zs \equiv map2\text{-}spmf\ (\lambda a.\ case\text{-}prod\ (f\ a))\ xs\ (pair\text{-}spmf\ ys\ zs)$

lemma $map\text{-}spmf\text{-}cong2:$
assumes $p = map\text{-}spmf\ m\ q \wedge x.\ x \in set\text{-}spmf\ q \implies f\ (m\ x) = g\ x$
shows $map\text{-}spmf\ f\ p = map\text{-}spmf\ g\ q$
 $\langle proof \rangle$

lemma $bind\text{-}spmf\text{-}cong2:$
assumes $p = map\text{-}spmf\ m\ q \wedge x.\ x \in set\text{-}spmf\ q \implies f\ (m\ x) = g\ x$
shows $bind\text{-}spmf\ p\ f = bind\text{-}spmf\ q\ g$

<proof>

lemma *map2-spmf-map2-sequence*:

$length\ xss = length\ yss \implies map2\text{-}spmf\ (map2\ f)\ (sequence\text{-}spmf\ xss)\ (sequence\text{-}spmf\ yss) = sequence\text{-}spmf\ (map2\ (map2\text{-}spmf\ f)\ xss\ yss)$

<proof>

abbreviation $map3 :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list \Rightarrow 'd\ list$
where

$map3\ f\ a\ b\ c \equiv map2\ (\lambda a\ (b,c).\ f\ a\ b\ c)\ a\ (zip\ b\ c)$

lemma *map3-spmf-map3-sequence*:

$length\ xss = length\ yss \implies length\ yss = length\ zss \implies map3\text{-}spmf\ (map3\ f)\ (sequence\text{-}spmf\ xss)\ (sequence\text{-}spmf\ yss)\ (sequence\text{-}spmf\ zss) = sequence\text{-}spmf\ (map3\ (map3\text{-}spmf\ f)\ xss\ yss\ zss)$

<proof>

lemma *in-pairD2*:

$x \in A \times B \implies snd\ x \in B$

<proof>

lemma *list-map-cong2*:

$x = map\ m\ y \implies (\bigwedge z. z \in set\ y = simp \implies f\ (m\ z) = g\ z) \implies map\ f\ x = map\ g\ y$

<proof>

lemma *map-swap-zip*:

$map\ prod.swap\ (zip\ xs\ ys) = zip\ ys\ xs$

<proof>

lemma *inv-zero-sharing-sequence*:

$n = length\ x \implies$

$map\text{-}spmf\ (\lambda \zeta s. (\zeta s, map2\ (map\text{-}sharing2\ (+))\ x\ \zeta s))\ (sequence\text{-}spmf\ (replicate\ n\ zero\text{-}sharing))$

$=$

$map\text{-}spmf\ (\lambda ys. (map2\ (map\text{-}sharing2\ (-))\ ys\ x, ys))\ (sequence\text{-}spmf\ (map\ (share\text{-}nat\ \circ reconstruct)\ x))$

<proof>

lemma *get-party-map-sharing2*:

$get\text{-}party\ p \circ (case\text{-}prod\ (map\text{-}sharing2\ f)) = case\text{-}prod\ f \circ map\text{-}prod\ (get\text{-}party\ p)$

<proof>

lemma *map-map-prod-zip*:

$map\ (map\text{-}prod\ f\ g)\ (zip\ xs\ ys) = zip\ (map\ f\ xs)\ (map\ g\ ys)$

<proof>

lemma *map-map-prod-zip'*:

$map (h \circ map\text{-}prod\ f\ g)\ (zip\ xs\ ys) = map\ h\ (zip\ (map\ f\ xs)\ (map\ g\ ys))$
<proof>

lemma *eq-map-spmf-conv*:

assumes $\bigwedge x. f\ (f'\ x) = x\ f' = inv\ f\ map\text{-}spmf\ f'\ x = y$
shows $x = map\text{-}spmf\ f\ y$
<proof>

lemma *lift-map-spmf-pairs*:

$map2\text{-}spmf\ f = F \implies map\text{-}spmf\ (case\text{-}prod\ f)\ (pair\text{-}spmf\ A\ B) = F\ A\ B$
<proof>

lemma *measure-pair-spmf-times'*:

$C = A \times B \implies measure\ (measure\text{-}spmf\ (pair\text{-}spmf\ p\ q))\ C = measure\ (measure\text{-}spmf\ p)\ A * measure\ (measure\text{-}spmf\ q)\ B$
<proof>

lemma *map-spmf-pairs-tmp*:

$map\text{-}spmf\ (\lambda(a,b,c,d,e,f,g). (a,e,b,f,c,g,d))\ (pair\text{-}spmf\ A\ (pair\text{-}spmf\ B\ (pair\text{-}spmf\ C\ (pair\text{-}spmf\ D\ (pair\text{-}spmf\ E\ (pair\text{-}spmf\ F\ G))))))$
 $= (pair\text{-}spmf\ A\ (pair\text{-}spmf\ E\ (pair\text{-}spmf\ B\ (pair\text{-}spmf\ F\ (pair\text{-}spmf\ C\ (pair\text{-}spmf\ G\ D))))))$
<proof>

lemma *case-case-prods-tmp*:

$(case\ case\ x\ of\ (a, b, c, d, e, f, g) \Rightarrow (a, e, b, f, c, g, d)\ of$
 $(ya, yb, yc, yd, ye, yf, yg) \Rightarrow F\ ya\ yb\ yc\ yd\ ye\ yf\ yg)$
 $= (case\ x\ of\ (a,b,c,d,e,f,g) \Rightarrow F\ a\ e\ b\ f\ c\ g\ d)$
<proof>

lemma *bind-spmf-permutes-cong*:

$(\bigwedge \pi. \pi\ permutes\ \{..<x::nat\}) \implies f\ \pi = g\ \pi$
 $\implies bind\text{-}spmf\ (spmf\text{-}of\text{-}set\ \{\pi. \pi\ permutes\ \{..<x\}\})\ f = bind\text{-}spmf\ (spmf\text{-}of\text{-}set\ \{\pi. \pi\ permutes\ \{..<x\}\})\ g$
<proof>

lemma *map-eq-iff-list-all2*:

$map\ f\ xs = map\ g\ ys \iff list\text{-}all2\ (\lambda x\ y. f\ x = g\ y)\ xs\ ys$
<proof>

lemma *bind-spmf-sequence-map-share-nat-cong*:

$(\bigwedge l. map\ reconstruct\ l = x \implies f\ l = g\ l)$
 $\implies bind\text{-}spmf\ (sequence\text{-}spmf\ (map\ share\text{-}nat\ x))\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ (map\ share\text{-}nat\ x))\ g$
<proof>

lemma *map-reconstruct-comp-eq-iff*:

$(\bigwedge x. x \in \text{set } xs \implies \text{reconstruct } (f x) = \text{reconstruct } x) \implies \text{map } (\text{reconstruct } \circ f)$
 $xs = \text{map } \text{reconstruct } xs$
 ⟨proof⟩

lemma *permute-list-replicate*:

$p \text{ permutes } \{..<n\} \implies \text{permute-list } p (\text{replicate } n x) = \text{replicate } n x$
 ⟨proof⟩

lemma *map2-minus-zero*:

$\text{length } xs = \text{length } ys \implies (\bigwedge y::\text{natL}. y \in \text{set } ys \implies y = 0) \implies \text{map2 } (-) xs ys$
 $= xs$
 ⟨proof⟩

lemma *permute-comp-left-inj*:

$p \text{ permutes } \{..<n\} \implies \text{inj } (\lambda p'. p \circ p')$
 ⟨proof⟩

lemma *permute-comp-left-inj-on*:

$p \text{ permutes } \{..<n\} \implies \text{inj-on } (\lambda p'. p \circ p') A$
 ⟨proof⟩

lemma *permute-comp-right-inj*:

$p \text{ permutes } \{..<n\} \implies \text{inj } (\lambda p'. p' \circ p)$
 ⟨proof⟩

lemma *permute-comp-right-inj-on*:

$p \text{ permutes } \{..<n\} \implies \text{inj-on } (\lambda p'. p' \circ p) A$
 ⟨proof⟩

lemma *permutes-inv-comp-left*:

$p \text{ permutes } \{..<n\} \implies \text{inv } (\lambda p'. p \circ p') = (\lambda p'. \text{inv } p \circ p')$
 ⟨proof⟩

lemma *permutes-inv-comp-right*:

$p \text{ permutes } \{..<n\} \implies \text{inv } (\lambda p'. p' \circ p) = (\lambda p'. p' \circ \text{inv } p)$
 ⟨proof⟩

lemma *permutes-inv-comp-left-right*:

$\pi a \text{ permutes } \{..<n\} \implies \pi b \text{ permutes } \{..<n\} \implies \text{inv } (\lambda p'. \pi a \circ p' \circ \pi b) = (\lambda p'.$
 $\text{inv } \pi a \circ p' \circ \text{inv } \pi b)$
 ⟨proof⟩

lemma *permutes-inv-comp-left-left*:

$\pi a \text{ permutes } \{..<n\} \implies \pi b \text{ permutes } \{..<n\} \implies \text{inv } (\lambda p'. \pi a \circ \pi b \circ p') = (\lambda p'.$
 $\text{inv } \pi b \circ \text{inv } \pi a \circ p')$
 ⟨proof⟩

lemma *permutes-inv-comp-right-right*:

$\pi a \text{ permutes } \{..<n\} \implies \pi b \text{ permutes } \{..<n\} \implies \text{inv } (\lambda p'. p' \circ \pi a \circ \pi b) = (\lambda p'.$

$p' \circ \text{inv } \pi b \circ \text{inv } \pi a$
 ⟨proof⟩

lemma *image-compose-permutations-left-right:*

fixes S
assumes πa permutes S πb permutes S
shows $\{\pi a \circ \pi \circ \pi b \mid \pi. \pi \text{ permutes } S\} = \{\pi. \pi \text{ permutes } S\}$
 ⟨proof⟩

lemma *image-compose-permutations-left-left:*

fixes S
assumes πa permutes S πb permutes S
shows $\{\pi a \circ \pi b \circ \pi \mid \pi. \pi \text{ permutes } S\} = \{\pi. \pi \text{ permutes } S\}$
 ⟨proof⟩

lemma *image-compose-permutations-right-right:*

fixes S
assumes πa permutes S πb permutes S
shows $\{\pi \circ \pi a \circ \pi b \mid \pi. \pi \text{ permutes } S\} = \{\pi. \pi \text{ permutes } S\}$
 ⟨proof⟩

lemma *random-perm-middle:*

defines $\text{random-perm } n \equiv \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n::\text{nat}\}\}$
shows
 $\text{map-spmf } (\lambda(\pi a, \pi b, \pi c). ((\pi a, \pi b, \pi c), \pi a \circ \pi b \circ \pi c)) (\text{pair-spmf } (\text{random-perm } n) (\text{pair-spmf } (\text{random-perm } n) (\text{random-perm } n)))$
 $= \text{map-spmf } (\lambda(\pi, \pi a, \pi c). ((\pi a, \text{inv } \pi a \circ \pi \circ \text{inv } \pi c, \pi c), \pi)) (\text{pair-spmf } (\text{random-perm } n) (\text{pair-spmf } (\text{random-perm } n) (\text{random-perm } n)))$
 (is ?lhs = ?rhs)
 ⟨proof⟩

lemma *random-perm-right:*

defines $\text{random-perm } n \equiv \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n::\text{nat}\}\}$
shows
 $\text{map-spmf } (\lambda(\pi a, \pi b, \pi c). ((\pi a, \pi b, \pi c), \pi a \circ \pi b \circ \pi c)) (\text{pair-spmf } (\text{random-perm } n) (\text{pair-spmf } (\text{random-perm } n) (\text{random-perm } n)))$
 $= \text{map-spmf } (\lambda(\pi, \pi a, \pi b). ((\pi a, \pi b, \text{inv } \pi b \circ \text{inv } \pi a \circ \pi), \pi)) (\text{pair-spmf } (\text{random-perm } n) (\text{pair-spmf } (\text{random-perm } n) (\text{random-perm } n)))$
 (is ?lhs = ?rhs)
 ⟨proof⟩

lemma *random-perm-left:*

defines $\text{random-perm } n \equiv \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n::\text{nat}\}\}$
shows
 $\text{map-spmf } (\lambda(\pi a, \pi b, \pi c). ((\pi a, \pi b, \pi c), \pi a \circ \pi b \circ \pi c)) (\text{pair-spmf } (\text{random-perm } n) (\text{pair-spmf } (\text{random-perm } n) (\text{random-perm } n)))$
 $= \text{map-spmf } (\lambda(\pi, \pi b, \pi c). ((\pi \circ \text{inv } \pi c \circ \text{inv } \pi b, \pi b, \pi c), \pi)) (\text{pair-spmf } (\text{random-perm } n) (\text{pair-spmf } (\text{random-perm } n) (\text{random-perm } n)))$
 (is ?lhs = ?rhs)

<proof>

lemma *case-prod-return-spmf*:

$case\text{-}prod (\lambda a b. return\text{-}spmf (f a b)) = (\lambda x. return\text{-}spmf (case\text{-}prod f x))$

<proof>

lemma *sequence-share-nat-calc'*:

assumes $r1 \neq r2$ $r2 \neq r3$ $r3 \neq r1$

shows

$sequence\text{-}spmf (map\ share\text{-}nat\ xs) = (do \{$
 $let\ n = length\ xs;$
 $let\ random\text{-}seq = sequence\text{-}spmf (replicate\ n (spmf\text{-}of\text{-}set\ UNIV));$
 $(dp, dpn) \leftarrow (pair\text{-}spmf\ random\text{-}seq\ random\text{-}seq);$
 $return\text{-}spmf (map3 (\lambda x a b. make\text{-}sharing' r1 r2 r3 a b (x - (a + b))) xs dp$

$dpn)$

$\}) (is - = ?rhs)$

<proof>

lemma *reconstruct-stack-sharing-eq-reconstruct*:

$reconstruct \circ aby3\text{-}stack\text{-}sharing\ r = reconstruct$

<proof>

lemma *map2-ignore1*:

$length\ xs = length\ ys \implies map2 (\lambda \cdot. f) xs ys = map f ys$

<proof>

lemma *map2-ignore2*:

$length\ xs = length\ ys \implies map2 (\lambda a b. f a) xs ys = map f xs$

<proof>

lemma *map-sequence-share-nat-reconstruct*:

$map\text{-}spmf (\lambda x. (x, map\ reconstruct\ x)) (sequence\text{-}spmf (map\ share\text{-}nat\ y)) =$
 $map\text{-}spmf (\lambda x. (x, y)) (sequence\text{-}spmf (map\ share\text{-}nat\ y))$

<proof>

theorem *shuffle-secrecy*:

assumes

$is\text{-}uniform\text{-}sharing\text{-}list\ x\text{-}dist$

shows

$(do \{$
 $x \leftarrow x\text{-}dist;$
 $(msg, y) \leftarrow aby3\text{-}shuffleR\ x;$
 $return\text{-}spmf (map (get\text{-}party\ r) x,$
 $get\text{-}party\ r\ msg,$
 $y)$
 $\})$

=
 (do {
 $x \leftarrow x\text{-dist}$;
 $y \leftarrow \text{aby3-shuffleF } x$;
 let $xr = \text{map } (\text{get-party } r) x$;
 let $yr = \text{map } (\text{get-party } r) y$;
 $\text{msg} \leftarrow S r xr yr$;
 return-spmf (xr, msg, y)
 })
 (is ?lhs = ?rhs)
 <proof>

lemma *Collect-case-prod*:
 $\{f x y \mid x y. P x y\} = (\text{case-prod } f) \text{ ` } (\text{Collect } (\text{case-prod } P))$
 <proof>

lemma *inj-split-Cons'*: $\text{inj-on } (\lambda(n, xs). n\#xs) X$
 <proof>

lemma *finite-indicator-eq-sum*:
 $\text{finite } A \implies \text{indicat-real } A x = \text{sum } (\text{indicat-real } \{x\}) A$
 <proof>

lemma *spmf-of-set-Cons*:
 $\text{spmf-of-set } (\text{set-Cons } A B) = \text{map2-spmf } (\#) (\text{spmf-of-set } A) (\text{spmf-of-set } B)$
 <proof>

lemma *sequence-spmf-replicate*:
 $\text{sequence-spmf } (\text{replicate } n (\text{spmf-of-set } A)) = \text{spmf-of-set } (\text{listset } (\text{replicate } n A))$
 <proof>

lemma *listset-replicate*:
 $\text{listset } (\text{replicate } n A) = \{l. \text{length } l = n \wedge \text{set } l \subseteq A\}$
 <proof>

lemma *map2-map2-map3*:
 $\text{map2 } f (\text{map2 } g x y) z = \text{map3 } (\lambda x y. f (g x y)) x y z$
 <proof>

lemma *inv-add-sequence*:
assumes $n = \text{length } x$
shows
 $\text{map-spmf } (\lambda\zeta::\text{natL list. } (\zeta, \text{map2 } (+) \zeta x)) (\text{sequence-spmf } (\text{replicate } n (\text{spmf-of-set } \text{UNIV})))$
 =
 $\text{map-spmf } (\lambda y. (\text{map2 } (-) y x, y)) (\text{sequence-spmf } (\text{replicate } n (\text{spmf-of-set } \text{UNIV})))$
 <proof>

lemma *S1-def-simplified*:

```

S1 x1 yc1 = (do {
  let n = length x1;

  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};
  ζa1::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
  yb1::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
  yb2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

  let ζc1 = map2 (-) (yc1) (permute-list πc (map2 (+) yb1 yb2));
  return-spmf ((ζa1, yb1, ζc1), yb2, πa, πc)
})

```

<proof>

lemma *S2-def-simplified*:

```

S2 x2 yc2 = (do {
  let n = length x2;

  x3 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  ζa2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
  ζb2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

  let msg2 = ((ζa2, ζb2, yc2), x3, πa, πb);
  return-spmf msg2
})

```

<proof>

lemma *S3-def-simplified*:

```

S3 x3 yc3 = (do {
  let n = length x3;

  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};
  ya3::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
  ya1::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
  ζb3::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

  let ζc3 = map2 (-) yc3 (permute-list πc (map2 (+) ζb3 (permute-list πb
(map2 (+) ya3 ya1))));
  return-spmf ((ya3, ζb3, ζc3), ya1, πb, πc)
})

```

<proof>

```

end
theory Multiplication
  imports
    Additive-Sharing
    Spmf-Common
    Sharing-Lemmas
begin

```

This is the formalisation of ABY3's multiplication protocol. We manually book-keep the messages to be simulated, and we manually define the simulator used in the simulation proof.

```

definition do-mul :: (natL × natL) ⇒ (natL × natL) ⇒ natL where
  do-mul xy1 xy2 = fst xy1 * snd xy1 + fst xy1 * snd xy2 + fst xy2 * snd xy1

```

```

type-synonym mul-in = natL × natL
type-synonym mul-msg = (natL × natL) × natL
type-synonym mul-view = mul-in × mul-msg
type-synonym mul-out = natL

```

```

definition aby3-mulR :: mul-in sharing ⇒ (mul-msg sharing × mul-out sharing)
spmf where
  aby3-mulR xy = (do {
    let xy-shift = shift-sharing xy;
    let z-raw = map-sharing2 do-mul xy xy-shift;
    ζ ← zero-sharing;
    let z = map-sharing2 (+) z-raw ζ;
    let msg = prod-sharing xy-shift ζ;
    return-spmf (msg, z)
  })

```

```

definition aby3-mulF :: mul-in sharing ⇒ mul-out sharing spmf where
  aby3-mulF xy = (
    let x = reconstruct (map-sharing fst xy);
        y = reconstruct (map-sharing snd xy);
    in share-nat (x * y)
  )

```

```

definition S :: mul-in ⇒ mul-out ⇒ mul-msg spmf where
  S inp outp = (do {
    let (x1, y1) = inp;
    (x2, y2) ← spmf-of-set UNIV;
    let ζ = outp - do-mul (x1, y1) (x2, y2);
    return-spmf ((x2, y2), ζ)
  })

```

```

lemma reconstruct-do-mul:
  reconstruct (map-sharing2 do-mul xys (shift-sharing xys)) = reconstruct (map-sharing
fst xys) * reconstruct (map-sharing snd xys)

```

<proof>

theorem *mul-spec*:

fixes *x-dist y-dist* :: *natL sharing spmf*

assumes *is-uniform-sharing x-dist is-uniform-sharing y-dist*

shows

secure:

```
(do {  
  xs ← x-dist;  
  ys ← y-dist;  
  let inps = prod-sharing xs ys;  
  (msgs, outps) ← (aby3-mulR inps);  
  let view = (get-party p inps, get-party p msgs);  
  return-spmf (view, outps)  
})
```

=

```
(do {  
  xs ← x-dist;  
  ys ← y-dist;  
  let inps = prod-sharing xs ys;  
  outps ← aby3-mulF inps;  
  msg ← S (get-party p inps) (get-party p outps);  
  let view = (get-party p inps, msg);  
  return-spmf (view, outps)  
}) (is ?secure)
```

and

correct:

```
is-uniform-sharing (do {  
  xs ← x-dist;  
  ys ← y-dist;  
  aby3-mulF (prod-sharing xs ys)  
}) (is ?uniform)
```

<proof>

end

theory *Multiplication-Synthesization*

imports

Multiplication

begin

This is an experimental re-formalization of the multiplication protocol, which differs from the original one in three aspects: 1) We use the writer transformer for automatic bookkeeping of simulation obligations in the privacy proof. Since monad transformers are hard to deal with in HOL, we combine (writer transformer + spmf monad) into `writer_spmf`. To ease the modelling, we allow heterogeneous message types in the binding operation,

a technicality that might disqualify it as a monad but, luckily, does not stop us from using the built-in do-notation. 2) We wraps the adding of zero-sharing into a new operation called “sharing flattening”. The proof for the sharing flattening is then “composed” into the larger proof for multiplication. 3) The simulator is not manually defined but synthesized through **schematic-goal**.

type-synonym $(\text{'val}, \text{'msg}) \text{writer-spmf} = (\text{'val} \times \text{'msg}) \text{spm}$

definition $\text{bind-writer-spmf} :: (\text{'val1}, \text{'msg1}) \text{writer-spmf} \Rightarrow (\text{'val1} \Rightarrow (\text{'val2}, \text{'msg2}) \text{writer-spmf}) \Rightarrow (\text{'val2}, (\text{'msg1} \times \text{'msg2})) \text{writer-spmf}$ **where**
 $\text{bind-writer-spmf } x f = \text{bind-spmf } x (\lambda(\text{val1}, \text{msg1}). \text{map-spmf } (\lambda(\text{val2}, \text{msg2}). (\text{val2}, (\text{msg1}, \text{msg2})))) (f \text{val1}))$

adhoc-overloading $\text{Monad-Syntax.bind} \Rightarrow \text{bind-writer-spmf}$

definition $\text{flatten-sharingF} :: \text{natL sharing} \Rightarrow \text{natL sharing spmf}$ **where**
 $\text{flatten-sharingF } s = \text{share-nat } (\text{reconstruct } s)$

definition $\text{flatten-sharingR} :: \text{Role} \Rightarrow \text{natL sharing} \Rightarrow (\text{natL sharing}, \text{natL}) \text{writer-spmf}$ **where**

where
 $\text{flatten-sharingR } p s = \text{do } \{$
 $\quad \zeta \leftarrow \text{zero-sharing};$
 $\quad \text{let } r = \text{map-sharing2 } (+) s \zeta;$
 $\quad \text{return-spmf } (r, (\text{get-party } p \zeta))$
 $\}$

definition $\text{flatten-sharingS} :: \text{natL} \Rightarrow \text{natL} \Rightarrow \text{natL spmf}$ **where**
 $\text{flatten-sharingS } \text{inp } \text{outp} = \text{return-spmf } (\text{outp} - \text{inp})$

lemma $\text{flatten-sharing-spec}$:

$\text{flatten-sharingR } p x = \text{do } \{$
 $\quad y \leftarrow \text{flatten-sharingF } x;$
 $\quad \text{msg} \leftarrow \text{flatten-sharingS } (\text{get-party } p x) (\text{get-party } p y);$
 $\quad \text{return-spmf } (y, \text{msg})$
 $\}$
 $\langle \text{proof} \rangle$

definition $\text{aby3-mulR}' :: \text{Role} \Rightarrow \text{natL} \Rightarrow \text{natL} \Rightarrow (\text{natL sharing}, ((\text{natL} \times \text{natL}) \times (\text{natL} \times \text{natL})) \times \text{natL}) \text{writer-spmf}$ **where**

$\text{aby3-mulR}' p x y = \text{do } \{$
 $\quad xs \leftarrow \text{share-nat } x;$
 $\quad ys \leftarrow \text{share-nat } y;$
 $\quad \text{let } xy = \text{prod-sharing } xs \text{ } ys;$
 $\quad \text{let } xy\text{-shift} = \text{shift-sharing } xy;$
 $\quad \text{let } z\text{-raw} = \text{map-sharing2 } \text{do-mul } xy \text{ } xy\text{-shift};$
 $\quad (z, \zeta\text{-msg}) \leftarrow \text{flatten-sharingR } p \text{ } z\text{-raw};$
 $\}$

return-spmf (z, (((get-party p xs, get-party p ys), get-party p xy-shift), ζ-msg))
}

definition *aby3-mulF'* :: natL ⇒ natL ⇒ natL sharing spmf **where**
aby3-mulF' x y = share-nat (x * y)

definition *aby3-mulS'* :: natL ⇒ (((natL × natL) × (natL × natL)) × natL) spmf
where

aby3-mulS' z = do {
x1 ← spmf-of-set UNIV;
x2 ← spmf-of-set UNIV;
y1 ← spmf-of-set UNIV;
y2 ← spmf-of-set UNIV;
let ζ = z - (x1 * y1 + x1 * y2 + x2 * y1);
return-spmf (((x1, y1), (x2, y2)), ζ)
}

lemma *set-spmf-share-nat*:
set-spmf (share-nat x) = {s. reconstruct s = x}
⟨proof⟩

lemma *reconstruct-share-nat'*:
pred-spmf (λs. reconstruct s = x) (share-nat x)
⟨proof⟩

lemma *share-nat-cong*:
x = y ⇒ (∧s. reconstruct s = x ⇒ f s = g s) ⇒ bind-spmf (share-nat x) f
= bind-spmf (share-nat y) g
⟨proof⟩

lemma *return-ResSim*:
return-spmf (r, s) = bind-spmf (return-spmf s) (λmsg. return-spmf (r, msg))
⟨proof⟩

schematic-goal *aby3-mul-spec*:
aby3-mulR' p x y =
bind-spmf (*aby3-mulF'* x y) (λz.
bind-spmf (?*aby3-mulS'* (get-party p z)) (λmsg.
return-spmf (z, msg)))
⟨proof⟩

end

References

- [1] P. Laud and M. Pettai. Secure multiparty sorting protocols with covert privacy. *Nordic Conference on Secure IT Systems*, pages 216–231, 2016.
- [2] P. Mohassel and P. Rindal. Aby3: A mixed protocol framework for machine learning. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.