

ABY3 Multiplication and Array Shuffling

Shuwei Hu

April 18, 2024

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 $\{0::'a..<L\}$ 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:  $L > 1$ 
by auto
```

```
typedef  $natL = \{0..<int L\}$ 
using L-gt-1 by auto
```

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

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

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

```

using L-gt-1 by simp

lift-definition one-natL :: natL is 1
using L-gt-1 by simp

lift-definition uminus-natL :: natL  $\Rightarrow$  natL is  $\lambda x. (-x) \text{ mod } \text{int } L$ 
by simp

lift-definition plus-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  natL is  $\lambda x y. (x + y) \text{ mod } \text{int } L$ 
by simp

lift-definition minus-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  natL is  $\lambda x y. (x - y) \text{ mod } \text{int } L$ 
by simp

lift-definition times-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  natL is  $\lambda x y. (x * y) \text{ mod } \text{int } L$ 
by simp

instance
by (standard; (transfer, simp add: algebra-simps mod-simps))

end
typ int

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

lift-definition inf-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  natL is inf
unfolding inf-int-def by auto

lift-definition sup-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  natL is sup
unfolding sup-int-def by auto

lift-definition less-eq-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  bool is less-eq .

lift-definition less-natL :: natL  $\Rightarrow$  natL  $\Rightarrow$  bool is less .

lift-definition top-natL :: natL is int L - 1
using L-gt-1 by simp

lift-definition bot-natL :: natL is 0
using L-gt-1 by simp

instance
by (standard; (transfer, simp add: inf-int-def sup-int-def))+

end

instantiation natL :: semiring-modulo
begin

```

```

lift-definition divide-natL :: natL ⇒ natL ⇒ natL is divide
  apply (auto simp: div-int-pos-iff)
  by (smt (verit) div-by-0 div-by-1 zdiv-mono2)

lift-definition modulo-natL :: natL ⇒ natL ⇒ natL is modulo
  apply (auto simp: mod-int-pos-iff)
  by (smt (verit) zmod-le-nonneg-dividend)

instance
  by (standard; (transfer, simp add: mod-simps))

end

instance natL :: finite
  apply standard
  unfolding type-definition.univ[OF type-definition-natL]
  by simp

lemma natL-card[simp]:
  CARD(natL) = L
  unfolding type-definition.univ[OF type-definition-natL]
  apply (subst card-image)
  subgoal by (meson Abs-natL-inject inj-onI)
  subgoal by simp
  done

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

datatype Role = Party1 | Party2 | Party3

lemma Role-exhaust'[dest!]:
  r ≠ Party1 ⇒ r ≠ Party2 ⇒ r ≠ Party3 ⇒ False
  using Role.exhaust by blast

lemma Role-exhaust:
  (r1::Role)≠r2 ⇒ r2≠r3 ⇒ r3≠r1 ⇒ r=r1 ∨ r=r2 ∨ r=r3
  by (metis Role.exhaust)

type-synonym 'a sharing = Role ⇒ 'a

instance Role :: finite
  apply (standard)
  by (metis (full-types) Role.exhaust ex-new-if-finite finite.intros(1) finite-insert

```

insert-iff)

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)
unfolding *get-party-def map-sharing-def comp-def* ..

definition *make-sharing'* :: Role ⇒ Role ⇒ Role ⇒ 'a ⇒ 'a ⇒ 'a ⇒ 'a sharing
where
make-sharing' r1 r2 r3 x1 x2 x3 = *undefined*(r1:=x1, r2:=x2, r3:=x3)

abbreviation *make-sharing* ≡ *make-sharing'* Party1 Party2 Party3

lemma *make-sharing'-sel*:
assumes r1 ≠ r2 r2 ≠ r3 r3 ≠ 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
unfolding *make-sharing'-def get-party-def*
using *assms* **by** *simp-all*

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
by (*simp-all add: make-sharing'-sel*)

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 ⇔ False r = *next-role* r ⇔ False
by (*cases* r; *simp*)⁺

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

prev-role $r = r \longleftrightarrow \text{False}$ $r = \text{prev-role } r \longleftrightarrow \text{False}$
by (*cases* r ; *simp*)**+**

lemma *next-sharing-neq-prev*[*simp*]:
next-role $r = \text{prev-role } r \longleftrightarrow \text{False}$ $\text{prev-role } r = \text{next-role } r \longleftrightarrow \text{False}$
by (*cases* r ; *simp*)**+**

lemma *role-otherE*:
obtains $r :: \text{Role}$ **where** $r0 \neq r$ $r \neq r1$
by (*cases* $r0$; *cases* $r1$) *auto*

lemma *make-sharing-sel-p2*:
shows
 $\text{get-party } (\text{prev-role } r) (\text{make-sharing}' (\text{prev-role } r) r (\text{next-role } r) x1 x2 x3) =$
 $x1$
 $\text{get-party } r (\text{make-sharing}' (\text{prev-role } r) r (\text{next-role } r) x1 x2 x3) = x2$
 $\text{get-party } (\text{next-role } r) (\text{make-sharing}' (\text{prev-role } r) r (\text{next-role } r) x1 x2 x3) =$
 $x3$
using *make-sharing'-sel*[*of prev-role r r next-role r x1 x2 x3, simplified*] .

lemma *sharing-cases*[*cases type*]:
obtains $x1 x2 x3$ **where** $s = \text{make-sharing } x1 x2 x3$
subgoal premises P
apply (*rule* $P[\text{of } s \text{ Party1 } s \text{ Party2 } s \text{ Party3}]$)
apply (*rule ext*)
subgoal for p
unfolding *make-sharing'-def* **by** (*cases* p ; *simp*)
done
done

lemma *sharing-cases'*:
assumes $p1 \neq p2$ $p2 \neq p3$ $p3 \neq p1$
obtains $x1 x2 x3$ **where** $s = \text{make-sharing}' p1 p2 p3 x1 x2 x3$
subgoal premises P
apply (*rule* $P[\text{of } s p1 s p2 s p3]$)
apply (*rule ext*)
subgoal for p
unfolding *make-sharing'-def* **using** *assms Role-exhaust* **by** (*cases* p ; *auto*)
done
done

lemma *make-sharing-collapse*[*simp*]:
 $\text{make-sharing } (\text{get-party } \text{Party1 } s) (\text{get-party } \text{Party2 } s) (\text{get-party } \text{Party3 } s) = s$
by (*cases* s) *simp*

lemma *sharing-eqI'*:
 $\llbracket \text{get-party } \text{Party1 } x = \text{get-party } \text{Party1 } y;$
 $\text{get-party } \text{Party2 } x = \text{get-party } \text{Party2 } y;$
 $\text{get-party } \text{Party3 } x = \text{get-party } \text{Party3 } y \rrbracket$

$\implies x = y$
apply (*rule ext*)
subgoal for *r*
 unfolding *get-party-def* **by** (*cases r; simp*)
done

lemma *sharing-eqI*[*intro*]:
 $(\bigwedge r. \text{get-party } r \ x = \text{get-party } r \ y) \implies x = y$
apply (*rule ext*)
subgoal for *r*
 unfolding *get-party-def* **by** (*cases r; simp*)
done

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

abbreviation *map-sharing2* :: ('*a* \Rightarrow '*b* \Rightarrow '*c*) \Rightarrow '*a* sharing \Rightarrow '*b* sharing \Rightarrow '*c* sharing **where**
map-sharing2 *f* *xs* *ys* \equiv *map-sharing* (*case-prod f*) (*prod-sharing* *xs* *ys*)

lemma *prod-sharing-sel*[*simp*]:
 $\text{get-party } r \ (\text{prod-sharing } x \ y) = (\text{get-party } r \ x, \text{get-party } r \ y)$
unfolding *get-party-def* *corec-prod-apply* ..

lemma *prod-sharing-def-alt*:
prod-sharing *x* *y* = *make-sharing*
 (*get-party* *Party1* *x*, *get-party* *Party1* *y*)
 (*get-party* *Party2* *x*, *get-party* *Party2* *y*)
 (*get-party* *Party3* *x*, *get-party* *Party3* *y*)
by (*auto intro: sharing-eqI'*)

lemma *prod-sharing-map-sel*[*simp*]:
map-sharing *fst* (*prod-sharing* *x* *y*) = *x*
map-sharing *snd* (*prod-sharing* *x* *y*) = *y*
unfolding *map-sharing-def* *comp-def* **by** *simp-all*

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*)
unfolding *shift-sharing-def* *make-sharing'-def* *comp-def* *get-party-def*
by (*auto*)

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*:

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

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

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

unfolding *reshare-def shift-sharing-def*

by *simp-all*

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

$derep-sharing = map-sharing\ snd$

lemma *derep-sharing-sel*:

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

unfolding *derep-sharing-def* **by** *simp*

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

$derep-sharing\ (reshare\ s) = s$

unfolding *derep-sharing-def reshare-def* **by** *simp*

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*:

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

unfolding *map-repsharing-def* **by** *simp*

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

$valid-repsharing\ s \longleftrightarrow reshare\ (derep-sharing\ s) = s$

lemma *valid-repsharingE*:

assumes *valid-repsharing* *s*

obtains *p* **where** $s = reshare\ p$

using *assms* **unfolding** *valid-repsharing-def* **by** *metis*

lemma *map-repsharing-def-alt*:

$valid-repsharing\ s \Longrightarrow map-repsharing\ f\ s = map-sharing\ (map-prod\ f\ f)\ s$

by (*erule* *valid-repsharingE*) (*auto* *intro*: *sharing-eqI'* *simp*: *map-repsharing-reshare reshare-sel*)

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

$valid-repsharing\ s \Longrightarrow reshare\ (derep-sharing\ s) = s$

by (*erule* *valid-repsharingE*) *simp*

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

$valid-repsharing\ (reshare\ s)$

unfolding *valid-repsharing-def* **by** *simp*

definition *make-repsharing* :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a repsharing **where**

$make\text{-}repsharing\ x1\ x2\ x3 = reshare\ (make\text{-}sharing\ x1\ x2\ x3)$

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

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

lemma $reconstruct\text{-}share[simp]$:
 $reconstruct\text{-}rep\ (reshare\ s) = reconstruct\ s$
unfolding $reconstruct\text{-}rep\text{-}def$ **by** $simp$

lemma $reconstruct\text{-}def'$:
assumes $r1 \neq r2\ r2 \neq r3\ r3 \neq r1$
shows $reconstruct\ s = get\text{-}party\ r1\ s + get\text{-}party\ r2\ s + get\text{-}party\ r3\ s$
unfolding $reconstruct\text{-}def$
using $assms$
by $(cases\ r1; cases\ r2; cases\ r3; simp)$

lemma $reconstruct\text{-}make\text{-}sharing'[simp]$:
assumes $r1 \neq r2\ r2 \neq r3\ r3 \neq r1$
shows $reconstruct\ (make\text{-}sharing'\ r1\ r2\ r3\ x1\ x2\ x3) = x1 + x2 + x3$
unfolding $reconstruct\text{-}def$
unfolding $make\text{-}sharing'\text{-}def\ get\text{-}party\text{-}def$
using $assms$
by $(cases\ r1; cases\ r2; cases\ r3; simp)$

lemma $reconstruct\text{-}make\text{-}repsharing[simp]$:
 $reconstruct\text{-}rep\ (make\text{-}repsharing\ x1\ x2\ x3) = x1 + x2 + x3$
unfolding $make\text{-}repsharing\text{-}def$
unfolding $reconstruct\text{-}share$
by $simp$

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

lemma $comp\text{-}inj\text{-}on\text{-}iff'$:
 $inj\text{-}on\ (f' \circ f)\ A \iff inj\text{-}on\ f\ A \wedge inj\text{-}on\ f'\ (f\ ' A)$
using $comp\text{-}inj\text{-}on\text{-}iff\ inj\text{-}on\text{-}imageI\ inj\text{-}on\text{-}imageI2$ **by** $auto$

lemma $corec\text{-}prod\text{-}inject[simp]$:
 $corec\text{-}prod\ f\ g = corec\text{-}prod\ f'\ g' \iff f = f' \wedge g = g'$
unfolding $corec\text{-}prod\text{-}def$
by $(meson\ ext\ prod.inject)$

lemma $inj\text{-}on\text{-}corec\text{-}prodI$:
 $inj\text{-}on\ f\ A \vee inj\text{-}on\ g\ A \implies inj\text{-}on\ (\lambda x. corec\text{-}prod\ (f\ x)\ (g\ x))\ A$

by (*auto intro: inj-onI simp: inj-on-eq-iff*)

lemma *inj-on-reshare[simp]*:
inj-on reshare A
unfolding *reshare-def*
by (*rule inj-on-corec-prodI*) *simp*

lemma *inj-on-make-repsharing-eq-sharing*:
inj-on ($\lambda x. \text{make-repsharing } (f x) (g x) (h x)$) A \longleftrightarrow inj-on ($\lambda x. \text{make-sharing } (f x) (g x) (h x)$) A
unfolding *make-repsharing-def*
unfolding *comp-inj-on-iff'*
apply (*subst comp-inj-on-iff'[unfolded comp-def, where f'=reshare]*)
by *auto*

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*
using *assms by (metis make-sharing'-sel)*

lemma *make-sharing-inject[simp]*:
make-sharing x1 x2 x3 = make-sharing y1 y2 y3 \longleftrightarrow x1=y1 \wedge x2=y2 \wedge x3=y3
by *simp*

lemma *reshare-inject[simp]*:
reshare a = reshare b \longleftrightarrow a = b
unfolding *reshare-def*
by *auto*

lemma *make-repsharing-inject[simp]*:
make-repsharing x1 x2 x3 = make-repsharing y1 y2 y3 \longleftrightarrow x1=y1 \wedge x2=y2 \wedge x3=y3
unfolding *make-repsharing-def*
by *simp*

lemma *inj-on-make-sharingI*:
inj-on f A \vee inj-on g A \vee inj-on h A \implies inj-on ($\lambda x. \text{make-sharing } (f x) (g x) (h x)$) A
by (*auto intro: inj-onI simp: inj-on-eq-iff*)

lemma *inj-on-make-repsharingI*:
inj-on f A \vee inj-on g A \vee inj-on h A \implies inj-on ($\lambda x. \text{make-repsharing } (f x) (g x) (h x)$) A
unfolding *inj-on-make-repsharing-eq-sharing using inj-on-make-sharingI* .

lemma *finite'-card-0*: $\text{finite}' A \longleftrightarrow \text{card } A \neq 0$
and *card-0-finite'*: $\text{card } A = 0 \longleftrightarrow \neg \text{finite}' A$
unfolding *card-eq-0-iff* **by** *auto*

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 $\text{spmf-of-set } (A \gg\!\!\gg f) = \text{spmf-of-set } A \gg\!\!\gg (\lambda x. \text{spmf-of-set } (f x))$

proof –

have *card-bind*: $\text{card } (A \gg\!\!\gg f) = \text{card } A * c$

proof –

{ **assume** *c0*: $c = 0$
note *card-UN-le*[*OF fin, folded bind-UNION, of f*]
hence *?thesis*
using *card c0* **by** *simp*
}
moreover
{ **assume** *cn0*: $c \neq 0$
then have *finite* $(f a)$ **if** $a \in A$ **for** a
using *card*[*OF that*] *card-0-finite'* **by** *metis*
hence *?thesis*
unfolding *bind-UNION*
using *assms* **by** (*auto simp: card-UN-disjoint'*[*OF disj - fin*])
}

ultimately show *?thesis* **by** *blast*

qed

show *?thesis*

apply (*rule smpf-eqI*)
apply (*unfold smpf-bind*)
apply (*unfold integral-smpf-of-set*)
apply (*unfold smpf-of-set*)
apply (*unfold card-bind*)
apply (*unfold indicator-UN-disjoint*[*folded bind-UNION, OF fin disj*])
apply (*auto simp: card sum-divide-distrib*[*symmetric*])
done

qed

lemma *ap-set-singleton*:

ap-set $\{f\} A = f \text{ ' } A$

by *blast*

lemma *ap-set-Union*:

ap-set $F A = (\bigcup f \in F. f \text{ ' } A)$

unfolding *ap-set-def* **by** *auto*

lemma *ap-set-curry*:

ap-set (ap-set F A) B = ap-set (case-prod ' F) (A × B)

unfolding *ap-set-Union* **by** *auto*

definition *share-nat* :: *natL* ⇒ *natL* *sharing* *spmf* **where**

share-nat x = spmf-of-set (reconstruct - ' {x})

definition *zero-sharing* :: *natL* *sharing* *spmf* **where**

zero-sharing = share-nat 0

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

assumes [*simp*]: *r1 ≠ r2 r2 ≠ r3 r3 ≠ r1*

shows

share-nat x = (do {
 (x1,x2) ← pair-spmf (spmf-of-set UNIV) (spmf-of-set UNIV);
 let x3 = x - (x1 + x2);
 return-spmf (make-sharing' r1 r2 r3 x1 x2 x3)
})

apply (*unfold pair-spmf-of-set*)

apply (*unfold share-nat-def*)

apply (*unfold Let-def*)

apply (*unfold case-prod-unfold*)

apply (*fold map-spmf-conv-bind-spmf*)

apply (*subst map-spmf-of-set-inj-on*)

subgoal using *assms* **by** (*auto intro: inj-onI*)

apply (*rule arg-cong[where f=spmf-of-set]*)

apply (*rule Set.equalityI*)

subgoal

apply (*rule Set.subsetI*)

subgoal for *xa*

by (*cases xa rule: sharing-cases'[OF assms]*) (*auto simp: Set.image-iff*)

done

subgoal using *assms* **by** *auto*

done

lemma *share-nat-def-calc*:

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

using *share-nat-def-calc'[of Party1 Party2 Party3]* **by** (*simp add: pair-spmf-of-set*)

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)
  unfolding share-nat-def-calc repshare-nat-def
  unfolding map-bind-spmf comp-def map-return-pmf make-repsharing-def case-prod-unfold
  by simp

lemma repshare-nat-def-alt:
  repshare-nat x = spmf-of-set {reshare s | s. reconstruct s = x}
  apply (unfold repshare-nat-def-share)
  apply (unfold share-nat-def)
  by (auto intro: arg-cong[where f=spmf-of-set])

lemma valid-nat-repsharing-reshare[simp]:
  valid-nat-repsharing (reconstruct s) (reshare s)
  unfolding valid-nat-repsharing-def
  unfolding reconstruct-share
  unfolding derep-sharing-reshare
  by simp

lemma valid-nat-repsharingE:
  assumes valid-nat-repsharing x s
  obtains s' where s = reshare s' and reconstruct s' = x
  subgoal premises prems
    apply (rule prems[of derep-sharing s])
    using assms unfolding valid-nat-repsharing-def reconstruct-rep-def by auto
  done

lemma repshare-nat-def-alt':
  repshare-nat x = spmf-of-set (Collect (valid-nat-repsharing x))
  unfolding repshare-nat-def-alt
  apply (rule arg-cong[where f=spmf-of-set])
  by (auto elim: valid-nat-repsharingE)

lemma share-nat-valid:
  pred-spmf (valid-nat-repsharing x) (repshare-nat x)
  unfolding repshare-nat-def-alt' by simp

lemma prod-sharing-map-fst-snd[simp]:
  prod-sharing (map-sharing fst s) (map-sharing snd s) = s
  by auto

end
theory Spmf-Common
  imports CryptHOL.CryptHOL
begin

```

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

lemma *mk-lossless-back-eq*:
 scale-spmf (weight-spmf s) (mk-lossless s) = s
 unfolding *mk-lossless-def*
 unfolding *scale-scale-spmf*
 by (*auto simp: field-simps weight-spmf-eq-0*)

lemma *cond-spmf-enforce*:
 cond-spmf sx (Collect A) = mk-lossless (enforce-spmf A sx)
 unfolding *enforce-spmf-def*
 unfolding *cond-spmf-alt*
 unfolding *restrict-spmf-def*
 unfolding *enforce-option-alt-def*
 apply (*rule arg-cong[where f=mk-lossless]*)
 apply (*rule arg-cong[where f= $\lambda x. \text{map-pmf } x \text{ } sx$]*)
 apply (*intro ext*)
 apply (*rule arg-cong[where f=Option.bind -]*)
 apply *auto*
 done

definition *rel-scale-spmf s t* \longleftrightarrow (*mk-lossless s = mk-lossless t*)

lemma *rel-scale-spmf-refl*:
 rel-scale-spmf s s
 unfolding *rel-scale-spmf-def ..*

lemma *rel-scale-spmf-sym*:
 rel-scale-spmf s t \implies rel-scale-spmf t s
 unfolding *rel-scale-spmf-def by simp*

lemma *rel-scale-spmf-trans*:
 rel-scale-spmf s t \implies rel-scale-spmf t u \implies rel-scale-spmf s u
 unfolding *rel-scale-spmf-def by simp*

lemma *rel-scale-spmf-equiv*:
 equivp rel-scale-spmf
 using *rel-scale-spmf-refl rel-scale-spmf-sym*
 by (*auto intro!: equivpI reflpI symplI transpI dest: rel-scale-spmf-trans*)

lemma *spmf-eq-iff*: *p = q \longleftrightarrow ($\forall i. \text{spmf } p \ i = \text{spmf } q \ i$)*
 using *spmf-eqI by auto*

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

```

using in-set-spmf-iff-spmf-spmf-eq-iff
by (metis)

lemma rel-scale-spmf-None:
  rel-scale-spmf s t  $\implies$  s = return-pmf None  $\longleftrightarrow$  t = return-pmf None
unfolding rel-scale-spmf-def by auto

lemma rel-scale-spmf-def-alt:
  rel-scale-spmf s t  $\longleftrightarrow$   $(\exists k > 0. s = \text{scale-spmf } k \ t)$ 
proof
  assume rel: rel-scale-spmf s t
  then consider  $(\text{isNone}) \ s = \text{return-pmf None} \wedge t = \text{return-pmf None} \mid (\text{notNone})$ 
  weight-spmf s > 0  $\wedge$  weight-spmf t > 0
    using rel-scale-spmf-None weight-spmf-eq-0 zero-less-measure-iff by blast
    then show  $\exists k > 0. s = \text{scale-spmf } k \ t$ 
  proof cases
    case isNone
      show ?thesis
      apply (rule exI[of - 1])
      using isNone by simp
    next
      case notNone
      have scale-spmf (weight-spmf s) (mk-lossless s) = scale-spmf (weight-spmf s /
weight-spmf t) t
        unfolding rel[unfolded rel-scale-spmf-def]
        unfolding mk-lossless-def
        unfolding scale-scale-spmf
        by (auto simp: field-simps)
      then show  $\exists k > 0. s = \text{scale-spmf } k \ t$ 
      apply (unfold mk-lossless-back-eq)
      using notNone divide-pos-pos by blast
    qed
  next
    assume  $\exists k > 0. s = \text{scale-spmf } k \ t$ 
    then obtain k where kpos: k > 0 and st: s = scale-spmf k t by blast
    then consider  $(\text{isNone}) \ \text{weight-spmf } s = 0 \wedge \text{weight-spmf } t = 0 \mid (\text{notNone})$ 
    weight-spmf s > 0  $\wedge$  weight-spmf t > 0
      using zero-less-measure-iff mult-pos-pos zero-less-measure-iff by (fastforce simp:
weight-scale-spmf)
      then show rel-scale-spmf s t
    proof cases
      case isNone
        then show ?thesis
        unfolding rel-scale-spmf-def weight-spmf-eq-0 by simp
      next
        case notNone
        then show ?thesis
        unfolding rel-scale-spmf-def
        unfolding mk-lossless-def

```

```

unfolding st
by (cases k < inverse (weight-spmf t))
    (auto simp: weight-scale-spmf scale-scale-spmf field-simps)
qed
qed

lemma rel-scale-spmf-def-alt2:
  rel-scale-spmf s t  $\longleftrightarrow$ 
    (s = return-pmf None  $\wedge$  t = return-pmf None)
  | (weight-spmf s > 0  $\wedge$  weight-spmf t > 0  $\wedge$  s = scale-spmf (weight-spmf s /
weight-spmf t) t)
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume rel: ?lhs
  then consider (isNone) s = return-pmf None  $\wedge$  t = return-pmf None | (notNone)
weight-spmf s > 0  $\wedge$  weight-spmf t > 0
    using rel-scale-spmf-None weight-spmf-eq-0 zero-less-measure-iff by blast
  thus ?rhs
  proof cases
    case notNone
      have scale-spmf (weight-spmf s) (mk-lossless s) = scale-spmf (weight-spmf s /
weight-spmf t) t
        unfolding rel[unfolded rel-scale-spmf-def]
        unfolding mk-lossless-def
        unfolding scale-scale-spmf
        by (auto simp: field-simps)
      thus ?thesis
      apply (unfold mk-lossless-back-eq)
      using notNone by simp
    qed simp
  next
    assume ?rhs
    then show ?lhs
    proof cases
      case right
        then have gt0: weight-spmf s > 0 weight-spmf t > 0 and st: s = scale-spmf
(weight-spmf s / weight-spmf t) t
          by auto
        then have (1 / weight-spmf t)  $\geq$  (weight-spmf s / weight-spmf t)
          using weight-spmf-le-1 divide-le-cancel by fastforce
        then show ?thesis
          unfolding rel-scale-spmf-def mk-lossless-def
          apply (subst (3) st)
          using gt0 by (auto simp: scale-scale-spmf field-simps)
        qed (simp add: rel-scale-spmf-refl)
    qed

```

```

lemma rel-scale-spmf-scale:
  r > 0  $\implies$  rel-scale-spmf s t  $\implies$  rel-scale-spmf s (scale-spmf r t)

```

apply (*unfold rel-scale-spmf-def-alt*)
by (*metis rel-scale-spmf-def rel-scale-spmf-def-alt*)

lemma *rel-scale-spmf-mk-lossless*:
 $rel_scale_spmf\ s\ t \implies rel_scale_spmf\ s\ (mk_lossless\ t)$
unfolding *rel-scale-spmf-def-alt*
unfolding *mk-lossless-def*
apply (*cases weight-spmf t = 0*)
subgoal **by**(*simp add: weight-spmf-eq-0*)
subgoal
apply (*auto simp: weight-spmf-eq-0 field-simps scale-scale-spmf*)
using *rel-scale-spmf-def-alt rel-scale-spmf-def-alt2* **by** *blast*
done

lemma *rel-scale-spmf-eq-iff*:
 $rel_scale_spmf\ s\ t \implies weight_spmf\ s = weight_spmf\ t \implies s = t$
unfolding *rel-scale-spmf-def-alt2* **by** *auto*

lemma *rel-scale-spmf-set-restrict*:
 $finite\ A \implies rel_scale_spmf\ (restrict_spmf\ (spmf_of_set\ A)\ B)\ (spmf_of_set\ (A \cap B))$
apply (*unfold rel-scale-spmf-def*)
apply (*fold cond-spmf-alt*)
apply (*subst cond-spmf-spmf-of-set*)
subgoal .
apply (*unfold mk-lossless-spmf-of-set*)
..

lemma *spmf-of-set-restrict-empty*:
 $A \cap B = \{\} \implies restrict_spmf\ (spmf_of_set\ A)\ B = return_pmf\ None$
unfolding *spmf-of-set-def*
by *simp*

lemma *spmf-of-set-restrict-scale*:
 $finite\ A \implies restrict_spmf\ (spmf_of_set\ A)\ B = scale_spmf\ (card\ (A \cap B) / card\ A)\ (spmf_of_set\ (A \cap B))$
apply (*rule rel-scale-spmf-eq-iff*)
subgoal
apply (*cases A ∩ B = {}*)
subgoal
by (*auto simp: spmf-of-set-restrict-empty intro: rel-scale-spmf-refl*)
subgoal
apply (*rule rel-scale-spmf-scale*)
subgoal
by (*metis card-gt-0-iff divide-pos-pos finite-Int inf-bot-left of-nat-0-less-iff*)
subgoal **by** (*rule rel-scale-spmf-set-restrict*)
done
done
subgoal


```

apply (auto simp add: weight-scale-spmf measure-spmf-of-set)
by (smt (verit, best) card-gt-0-iff card-mono disjoint-notin1 divide-le-eq-1-pos
finite-Int inf-le1 of-nat-0-less-iff of-nat-le-iff)
done

```

```

lemma min-em2:
   $\min a b = c \implies a \neq c \implies b = c$ 
unfolding min-def by auto

```

```

lemma weight-0-spmf:
   $\text{weight-spmf } s = 0 \implies \text{spmfs } s \ i = 0$ 
using order-trans[OF spmf-le-weight, of s 0 i] by simp

```

```

lemma mk-lossless-scale-absorb:
   $r > 0 \implies \text{mk-lossless } (\text{scale-spmf } r \ s) = \text{mk-lossless } s$ 
apply (rule rel-scale-spmf-eq-iff)
subgoal
  apply (rule rel-scale-spmf-trans[where t=s])
subgoal
  apply (rule rel-scale-spmf-sym)
  apply (rule rel-scale-spmf-mk-lossless)
  apply (rule rel-scale-spmf-scale)
  subgoal .
  subgoal by (rule rel-scale-spmf-refl)
  done
subgoal
  apply (rule rel-scale-spmf-mk-lossless)
  apply (rule rel-scale-spmf-refl)
  done
done
subgoal
  unfolding weight-mk-lossless
  by (auto simp flip: weight-spmf-eq-0 simp: weight-scale-spmf dest: min-em2)
done

```

```

lemma scale-spmf-None-iff:
   $\text{scale-spmf } k \ s = \text{return-pmf } \text{None} \iff k \leq 0 \vee s = \text{return-pmf } \text{None}$ 
apply (auto simp: spmf-eq-iff spmf-scale-spmf)
using
  inverse-nonpositive-iff-nonpositive
  weight-0-spmf
  measure-le-0-iff
by (smt (verit))

```

```

lemma spmf-of-pmf-the:
   $\text{lossless-spmf } s \implies \text{spmfs-of-pmf } (\text{map-pmf } \text{the } s) = s$ 
unfolding lossless-spmf-conv-spmfs-of-pmf by auto

```

```

lemma lossless-mk-lossless:

```

$s \neq \text{return-pmf None} \implies \text{lossless-spmf (mk-lossless s)}$
unfolding *lossless-spmf-def*
unfolding *weight-mk-lossless*
by *simp*

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

lemma *scale-weight-spmf-of-pmf:*
 $p = \text{scale-spmf (weight-spmf } p) (\text{spmf-of-pmf (pmf-of-spmf } p))$
unfolding *pmf-of-spmf-def*
apply (*cases* $p = \text{return-pmf None}$)
subgoal by *simp*
subgoal
apply (*subst mk-lossless-back-eq[of p, symmetric]*)
apply (*rule arg-cong[where f=scale-spmf -]*)
apply (*rule spmf-of-pmf-the[symmetric]*)
by (*fact lossless-mk-lossless*)
done

lemma *pmf-spmf:*
 $\text{pmf-of-spmf (spmf-of-pmf } p) = p$
unfolding *pmf-of-spmf-def*
unfolding *lossless-spmf-spmf-of-spmf[THEN mk-lossless-lossless]*
unfolding *map-the-spmf-of-pmf*
..

lemma *spmf-pmf:*
 $\text{lossless-spmf } p \implies \text{spmf-of-pmf (pmf-of-spmf } p) = p$
unfolding *pmf-of-spmf-def*
by (*simp add: spmf-of-pmf-the*)

lemma *pmf-of-spmf-scale-spmf:*
 $r > 0 \implies \text{pmf-of-spmf (scale-spmf } r p) = \text{pmf-of-spmf } p$
unfolding *pmf-of-spmf-def*
by (*simp add: mk-lossless-scale-absorb*)

lemma *nonempty-spmf-weight:*
 $p \neq \text{return-pmf None} \iff \text{weight-spmf } p > 0$
apply (*fold weight-spmf-eq-0*)
using *dual-order.not-eq-order-implies-strict[OF - weight-spmf-nonneg[of p]]*
by *auto*

lemma *pmf-of-spmf-mk-lossless:*
 $\text{pmf-of-spmf (mk-lossless } p) = \text{pmf-of-spmf } p$
apply (*cases* $p = \text{return-pmf None}$)
subgoal by *auto*
apply (*unfold mk-lossless-def*)
apply (*subst pmf-of-spmf-scale-spmf*)

subgoal by (*simp add: nonempty-spmf-weight*)
 ..

lemma *spmf-pmf'*:
 $p \neq \text{return-pmf None} \implies \text{spmf-of-pmf (pmf-of-spmf } p) = \text{mk-lossless } p$
apply (*subst spmf-pmf[of mk-lossless p, symmetric]*)
apply (*unfold pmf-of-spmf-mk-lossless*)
subgoal using *lossless-mk-lossless* .
subgoal ..
done

lemma *rel-scale-spmf-cond-UNIV*:
 $\text{rel-scale-spmf } s \text{ (cond-spmf } s \text{ UNIV)}$
unfolding *cond-spmf-UNIV*
by (*rule rel-scale-spmf-mk-lossless*) (*rule rel-scale-spmf-refl*)

lemma *set-pmf* $p \cap g \neq \{\}$ $\implies \text{pmf-prob } p \text{ (} f \cap g) = \text{pmf-prob (cond-pmf } p \text{ } g) f$
 $* \text{pmf-prob } p \text{ } g$
using *measure-cond-pmf*
unfolding *pmf-prob-def*
by (*metis Int-commute divide-eq-eq measure-measure-pmf-not-zero*)

lemma *bayes*:
 $\text{set-pmf } p \cap A \neq \{\} \implies \text{set-pmf } p \cap B \neq \{\} \implies$
 $\text{measure-pmf.prob (cond-pmf } p \text{ } A) B$
 $= \text{measure-pmf.prob (cond-pmf } p \text{ } B) A * \text{measure-pmf.prob } p \text{ } B / \text{measure-pmf.prob } p \text{ } A$
unfolding *measure-cond-pmf*
by (*subst inf-commute*) (*simp add: measure-pmf-zero-iff*)

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

lemma *spmf-prob = measure-measure-spmf*
unfolding *spmf-prob-def measure-measure-spmf-def ..*

lemma *spmf-prob-pmf*:
 $\text{spmf-prob } p \text{ } A = \text{pmf-prob } p \text{ (Some ' } A)$
unfolding *spmf-prob-def pmf-prob-def*
unfolding *measure-measure-spmf-conv-measure-pmf*
 ..

lemma *bayes-spmf*:
 $\text{spmf-prob (cond-spmf } p \text{ } A) B$
 $= \text{spmf-prob (cond-spmf } p \text{ } B) A * \text{spmf-prob } p \text{ } B / \text{spmf-prob } p \text{ } A$
unfolding *spmf-prob-def*
unfolding *measure-cond-spmf*
by (*subst inf-commute*) (*auto simp: measure-spmf-zero-iff*)

lemma *spmf-prob-pmf-of-spmf*:
 $spmf\text{-}prob\ p\ A = weight\text{-}spmf\ p * pmf\text{-}prob\ (pmf\text{-}of\text{-}spmf\ p)\ A$
apply (*subst scale-weight-spmf-of-pmf*)
apply (*unfold spmf-prob-def*)
apply (*subst measure-spmf-scale-spmf'*)
subgoal using *weight-spmf-le-1* .
by (*simp add: pmf-prob-def*)

lemma *cond-spmf-Int*:
 $cond\text{-}spmf\ (cond\text{-}spmf\ p\ A)\ B = cond\text{-}spmf\ p\ (A \cap B)$
apply (*rule spmf-eqI*)
apply (*unfold spmf-cond-spmf*)
apply(*auto simp add: measure-cond-spmf split: if-split-asm*)
using *Int-lower1 [THEN measure-spmf.finite-measure-mono[simplified]]* *mea-*
sure-le-0-iff
by *metis*

lemma *cond-spmf-prob*:
 $spmf\text{-}prob\ p\ (A \cap B) = spmf\text{-}prob\ (cond\text{-}spmf\ p\ A)\ B * spmf\text{-}prob\ p\ A$
unfolding *spmf-prob-def measure-cond-spmf*
using *Int-lower1 [THEN measure-spmf.finite-measure-mono[simplified]]* *mea-*
sure-le-0-iff
by (*metis mult-eq-0-iff nonzero-eq-divide-eq*)

definition *empty-spmf = return-pmf None*

lemma *spmf-prob-empty*:
 $spmf\text{-}prob\ empty\text{-}spmf\ A = 0$
unfolding *spmf-prob-def empty-spmf-def*
by *simp*

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

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

lemma *class.order-bot empty-spmf le-spmf lt-spmf*
oops

lemma *spmf-prob-cond-Int*:
 $spmf\text{-}prob\ (cond\text{-}spmf\ p\ C)\ (A \cap B)$
 $= spmf\text{-}prob\ (cond\text{-}spmf\ p\ (B \cap C))\ A * spmf\text{-}prob\ (cond\text{-}spmf\ p\ C)\ B$
apply (*subst Int-commute[of B C]*)
apply (*subst Int-commute[of A B]*)
apply (*fold cond-spmf-Int*)
using *cond-spmf-prob* .

lemma *cond-spmf-mk-lossless*:

$cond\text{-}spmf\ (mk\text{-}lossless\ p)\ A = cond\text{-}spmf\ p\ A$
apply (fold cond-spmf-UNIV)
apply (unfold cond-spmf-Int)
by simp

primrec sequence-spmf :: 'a spmf list \Rightarrow 'a list spmf **where**
 sequence-spmf [] = return-spmf []
 | sequence-spmf (x#xs) = map-spmf (case-prod Cons) (pair-spmf x (sequence-spmf xs))

lemma set-sequence-spmf:
 $set\text{-}spmf\ (sequence\text{-}spmf\ xs) = \{l.\ list\text{-}all2\ (\lambda x\ s.\ x \in set\text{-}spmf\ s)\ l\ xs\}$
by (induction xs) (auto simp: list-all2-Cons2)

lemma map-spmf-map-sequence:
 $map\text{-}spmf\ (map\ f)\ (sequence\text{-}spmf\ xs) = sequence\text{-}spmf\ (map\ (map\text{-}spmf\ f)\ xs)$
apply (induction xs)
subgoal by simp
subgoal premises IH
 unfolding list.map
 unfolding sequence-spmf.simps
apply (fold IH)
apply (unfold pair-map-spmf)
apply (unfold spmf.map-comp)
by (simp add: comp-def case-prod-map-prod prod.case-distrib)
done

lemma sequence-map-return-spmf:
 $sequence\text{-}spmf\ (map\ return\text{-}spmf\ xs) = return\text{-}spmf\ xs$
by (induction xs) auto

lemma sequence-bind-cong:
 $\llbracket xs=ys; \bigwedge y.\ length\ y = length\ ys \implies f\ y = g\ y \rrbracket \implies bind\text{-}spmf\ (sequence\text{-}spmf\ xs)\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ ys)\ g$
apply (rule bind-spmf-cong)
subgoal by simp
subgoal unfolding set-sequence-spmf list-all2-iff **by** simp
done

lemma bind-spmf-sequence-replicate-cong:
 $(\bigwedge l.\ length\ l = n \implies f\ l = g\ l)$
 $\implies bind\text{-}spmf\ (sequence\text{-}spmf\ (replicate\ n\ x))\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ (replicate\ n\ x))\ g$
by (rule bind-spmf-cong[OF refl]) (simp add: set-spmf-of-set finite-permutations set-sequence-spmf[unfolded list-all2-iff])

lemma bind-spmf-sequence-map-cong:
 $(\bigwedge l.\ length\ l = length\ x \implies f\ l = g\ l)$
 $\implies bind\text{-}spmf\ (sequence\text{-}spmf\ (map\ m\ x))\ f = bind\text{-}spmf\ (sequence\text{-}spmf\ (map\ m\ x))\ g$

$m\ x))\ g$
by (*rule* *bind-spmf-cong*[*OF refl*]) (*simp add: set-spmf-of-set finite-permutations set-sequence-spmf*[*unfolded list-all2-iff*])

lemma *lossless-pair-spmf-iff*:
 $lossless-spmf\ (pair-spmf\ a\ b) \longleftrightarrow lossless-spmf\ a \wedge lossless-spmf\ b$
unfolding *pair-spmf-alt-def*
by (*auto simp: set-spmf-eq-empty*)

lemma *lossless-sequence-spmf*:
 $(\bigwedge x. x \in set\ xs \implies lossless-spmf\ x) \implies lossless-spmf\ (sequence-spmf\ xs)$
by (*induction xs*) (*auto simp: lossless-pair-spmf-iff*)

end
theory *Sharing-Lemmas*
imports
Additive-Sharing
begin

lemma *share-nat-2party-uniform*:
 $p \neq q \implies map-spmf\ (\lambda s. (get-party\ p\ s, get-party\ q\ s))\ (share-nat\ x) = spmf-of-set\ UNIV$

proof –
assume *pq: p ≠ q*
obtain *r* **where** *r: q ≠ r r ≠ p*
using *role-otherE* .
show $map-spmf\ (\lambda s. (get-party\ p\ s, get-party\ q\ s))\ (share-nat\ x) = spmf-of-set\ UNIV$
apply (*unfold share-nat-def-calc'*[*OF pq r, simplified*])
apply (*unfold case-prod-unfold*)
apply (*fold map-spmf-conv-bind-spmf*)
apply (*unfold spmf.map-comp comp-def*)
apply (*unfold make-sharing'-sel*[*OF pq r*])
apply (*auto simp: pair-spmf-of-set*)
done

qed

lemma *share-nat-party-uniform*:
 $map-spmf\ (get-party\ p)\ (share-nat\ x) = spmf-of-set\ UNIV$
supply [*simp*] = *spmf.map-comp comp-def*
apply (*unfold share-nat-2party-uniform*[*of p next-role p, THEN arg-cong*[*where f=map-spmf fst, simplified*])
apply (*fold UNIV-Times-UNIV*)
apply (*fold pair-spmf-of-set*)
apply (*unfold map-fst-pair-spmf*)
by *simp*

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

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

definition $is\text{-uniform}\text{-sharing}2 :: (natL\ \text{sharing} \times natL\ \text{sharing})\ \text{spm}f \Rightarrow bool$
where

$is\text{-uniform}\text{-sharing}2\ s \iff (\exists xy :: (natL \times natL)\ \text{spm}f.$
 $s = \text{bind}\text{-spm}f\ xy\ (\lambda(x,y). \text{pair}\text{-spm}f\ (\text{share}\text{-nat } x)\ (\text{share}\text{-nat } y)))$

lemma $share\text{-nat}\text{-uniform}$:

$is\text{-uniform}\text{-sharing}\ (\text{share}\text{-nat } x)$
unfolding $is\text{-uniform}\text{-sharing}\text{-def}$
apply $(rule\ exI[\text{where } x = \text{return}\text{-spm}f\ x])$
by $simp$

lemma $share\text{-nat}\text{-lossless}$:

$lossless\text{-spm}f\ (\text{share}\text{-nat } x)$
unfolding $share\text{-nat}\text{-def}$
unfolding $lossless\text{-spm}f\text{-of}\text{-set}$
apply $rule$
subgoal
by $(rule\ finite\text{-subset}[\text{where } B = UNIV])\ auto$
subgoal
unfolding $vimage\text{-def}$
apply $simp$
apply $(rule\ exI[\text{where } x = \text{make}\text{-sharing } x\ 0\ 0])$
apply $simp$
done
done

lemma $uniform\text{-sharing}2$:

$is\text{-uniform}\text{-sharing } s \implies p1 \neq p2 \implies \text{map}\text{-spm}f\ (\lambda x. (\text{get}\text{-party } p1\ x, \text{get}\text{-party } p2\ x))\ s = \text{scale}\text{-spm}f\ (\text{weight}\text{-spm}f\ s)\ (\text{spm}f\text{-of}\text{-set } UNIV)$
unfolding $is\text{-uniform}\text{-sharing}\text{-def}$
apply $(erule\ exE)$
apply $simp$
unfolding $map\text{-bind}\text{-spm}f\ \text{comp}\text{-def}$
unfolding $share\text{-nat}\text{-2party}\text{-uniform}$
unfolding $bind\text{-spm}f\text{-const}$
unfolding $weight\text{-bind}\text{-spm}f$
apply $(subst\ Bochner\text{-Integration}\text{-integral}\text{-cong}[\text{where } g = \lambda\text{-}. 1])$
apply $(rule\ refl)$
subgoal **using** $share\text{-nat}\text{-lossless}$ **unfolding** $lossless\text{-spm}f\text{-def}$ **by** $simp$
subgoal **by** $simp$
done

lemma $uniform\text{-sharing}$:

$is\text{-uniform}\text{-sharing } s \implies \text{map}\text{-spm}f\ (\text{get}\text{-party } p)\ s = \text{scale}\text{-spm}f\ (\text{weight}\text{-spm}f\ s)$
 $(\text{spm}f\text{-of}\text{-set } UNIV)$
supply $[simp] = \text{spm}f.\text{map}\text{-comp}\ \text{comp}\text{-def}$

apply (*unfold uniform-sharing2*[**where** $?p1.0=p$ **and** $?p2.0=next\text{-}role\ p$, *THEN*
arg-cong[**where** $f=map\text{-}spmf\ fst$], *simplified*])
apply (*fold UNIV-Times-UNIV*)
apply (*fold pair-spmf-of-set*)
apply (*unfold map-scale-spmf*)
by *simp*

lemma *uniform-sharing'*:

$[[is\text{-}uniform\text{-}sharing\ s; lossless\text{-}spmf\ s]] \implies map\text{-}spmf\ (get\text{-}party\ p)\ s = spmf\text{-}of\text{-}set\ UNIV$
by (*simp add: uniform-sharing lossless-weight-spmfD*)

lemma *zero-masking-uniform*:

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

proof –

assume $pq: p \neq q$
obtain r **where** $r: q \neq r\ r \neq p$
using *role-otherE* .
note [*simp*] = *make-sharing'-sel[OF pq r]*
have $inj: inj\ (\lambda x. (get\text{-}party\ p\ s + fst\ x, get\text{-}party\ q\ s + snd\ x))$
unfolding *inj-def* **by** *simp*
have $surj: surj\ (\lambda x. (get\text{-}party\ p\ s + fst\ x, get\text{-}party\ q\ s + snd\ x))$
by (*rule finite-UNIV-inj-surj[OF - inj]*) *simp*
show $(map\text{-}spmf\ ((\lambda t. (get\text{-}party\ p\ t, get\text{-}party\ q\ t)) \circ map\text{-}sharing2\ (+)\ s)$
 $zero\text{-}sharing) = spmf\text{-}of\text{-}set\ UNIV$
unfolding *zero-sharing-def*
unfolding *share-nat-def-calc'[OF pq r]* *Let-def case-prod-unfold*
unfolding *map-spmf-conv-bind-spmf[symmetric]* *spmf.map-comp comp-def*
using $inj\ surj$ **by** (*auto simp: pair-spmf-of-set*)

qed

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

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

shows $s = t$

by (*smt (verit) assms Role.exhaust insert-commute insert-subset sharing-eqI' subset-insertI*)

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

assumes $p1 \neq p2\ p2 \neq p3\ p3 \neq p1$
shows $map\text{-}sharing\ f\ (make\text{-}sharing'\ p1\ p2\ p3\ x1\ x2\ x3) = make\text{-}sharing'\ p1\ p2\ p3\ (f\ x1)\ (f\ x2)\ (f\ x3)$
supply [*simp*] = *make-sharing'-sel[OF assms]*
apply (*rule sharing-eqI2[OF assms]*)
by *auto*

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


```

assumes  $p1 \neq p2$   $p2 \neq p3$   $p3 \neq p1$ 
shows prod-sharing (make-sharing'  $p1$   $p2$   $p3$   $x1$   $x2$   $x3$ ) (make-sharing'  $p1$   $p2$   $p3$ 
 $y1$   $y2$   $y3$ )
  = make-sharing'  $p1$   $p2$   $p3$  ( $x1, y1$ ) ( $x2, y2$ ) ( $x3, y3$ )
supply [simp] = make-sharing'-sel[OF assms]
apply (rule sharing-eqI2[OF assms])
by auto

```

```

lemma add-sharing-inj:
  inj (map-sharing2 (+) ( $s :: \text{natL sharing}$ ))
apply (rule injI)
subgoal for  $x$   $y$ 
  by (cases s; cases x; cases y) simp
done

```

```

lemma add-sharing-surj:
  surj (map-sharing2 (+) ( $s :: \text{natL sharing}$ ))
apply (rule finite-UNIV-inj-surj)
subgoal by simp
subgoal using add-sharing-inj .
done

```

```

lemma sharing-add-inv-eq-minus:
  Hilbert-Choice.inv (map-sharing2 (+) ( $s :: \text{natL sharing}$ ))  $t = \text{map-sharing2}$  (-)
 $t$   $s$ 
apply (rule inv-f-eq)
subgoal using add-sharing-inj .
by (cases s; cases t) simp

```

```

lemma zero-masking-eq-share-nat:
  map-spmf (map-sharing2 (+) ( $s :: \text{natL sharing}$ )) zero-sharing = share-nat
(reconstruct s)
unfolding zero-sharing-def share-nat-def
apply (subst map-spmf-of-set-inj-on)
subgoal
  apply (rule inj-on-subset[OF - subset-UNIV])
  apply (rule add-sharing-inj)
  done
apply (rule arg-cong[where  $f = \text{spm-f-of-set}$ ])
apply (rule antisym)
subgoal
  apply clarsimp
subgoal for  $t$ 
  apply (cases s; cases t)
  apply (auto simp: algebra-simps)
  done
done
subgoal
  apply clarsimp

```

```

subgoal for  $t$ 
  apply (cases  $s$ ; cases  $t$ )
  apply clarsimp
  apply (rule image-eqI)
  apply (rule surj-f-inv-f[OF add-sharing-surj, symmetric])
  unfolding sharing-add-inv-eq-minus
  apply (auto simp: algebra-simps)
  done
done
done

```

```

lemma inv-uniform':
  assumes  $ss: s \subseteq U$  and inj: inj-on  $f$   $U$ 
  shows  $\text{map-spmf } (\lambda x. (x, f x)) (\text{spmof-of-set } s) = \text{map-spmf } (\lambda y. (\text{Hilbert-Choice.inv-into } U f y, y)) (\text{spmof-of-set } (f \text{ ` } s))$ 
  apply (subst map-spmf-of-set-inj-on)
  subgoal using inj-on-convol-ident .
  apply (subst map-spmf-of-set-inj-on)
  subgoal using assms by (simp add: inj-on-def)
  apply (rule arg-cong[where  $f = \text{spmof-of-set}$ ])
  using assms by (auto simp: inv-into-f-f[OF inj] intro!: image-eqI)

```

```

lemma inv-uniform:
   $\text{inj } f \implies \text{map-spmf } (\lambda x. (x, f x)) (\text{spmof-of-set } s) = \text{map-spmf } (\lambda y. (\text{Hilbert-Choice.inv } f y, y)) (\text{spmof-of-set } (f \text{ ` } s))$ 
  using inv-uniform'[where  $U = UNIV$ , simplified] .

```

```

lemma reconstruct-plus:
   $\text{reconstruct } (\text{map-sharing2 } (+) x y) = \text{reconstruct } x + \text{reconstruct } y$ 
  by (cases  $x$ ; cases  $y$ ) simp

```

```

lemma reconstruct-minus:
   $\text{reconstruct } (\text{map-sharing2 } (-) x y) = \text{reconstruct } x - \text{reconstruct } y$ 
  by (cases  $x$ ; cases  $y$ ) simp

```

```

lemma plus-reconstruct:
   $\text{map-sharing2 } (+) x \text{ ` } \text{reconstruct } - \{y\} = \text{reconstruct } - \{\text{reconstruct } x + y\}$ 
  supply [simp] = reconstruct-plus reconstruct-minus
  apply rule
  subgoal by auto
  subgoal
    unfolding vimage-def image-def
    apply clarsimp
    subgoal for  $t$ 
      by (rule exI[where  $x = \text{map-sharing2 } (-) t x$ ]) auto
    done
  done

```

```

lemma inv-zero-sharing:

```

$map\text{-}spmf (\lambda\zeta. (\zeta, map\text{-}sharing2 (+) x \zeta)) zero\text{-}sharing = map\text{-}spmf (\lambda y. (map\text{-}sharing2 (-) y x, y)) (share\text{-}nat (reconstruct x))$
unfolding *zero-sharing-def share-nat-def*
apply (*subst inv-uniform*)
subgoal
using *add-sharing-inj[THEN inj-on-subset]* **by** *auto*
apply (*subst sharing-add-inv-eq-minus*)
apply (*subst plus-reconstruct*)
apply *simp*
done

lemma *hoist-map-spmf*:
 $(do \{x \leftarrow s; g x (f x)\}) = (do \{(x,y) \leftarrow map\text{-}spmf (\lambda x. (x, f x)) s; g x y\})$
unfolding *bind-map-spmf comp-def* **by** *simp*

lemma *hoist-map-spmf'*:
 $(do \{x \leftarrow s; g x (f x)\}) = (do \{(y,x) \leftarrow map\text{-}spmf (\lambda x. (f x, x)) s; g x y\})$
unfolding *bind-map-spmf comp-def* **by** *simp*

definition *HOIST-TAG* $x = x$
lemmas *hoist-tag = HOIST-TAG-def[symmetric]*

lemma *tagged-hoist-map-spmf*:
 $(do \{x \leftarrow s; g x (HOIST\text{-}TAG (f x))\}) = (do \{(x,y) \leftarrow map\text{-}spmf (\lambda x. (x, f x)) s; g x y\})$
unfolding *HOIST-TAG-def* **using** *hoist-map-spmf* .

lemma *get-prev-sharing[simp]*:
 $get\text{-}party p (shift\text{-}sharing s) = get\text{-}party (prev\text{-}role p) s$
unfolding *shift-sharing-def*
by (*cases p; simp*)

lemma *shift-sharing-make-sharing*:
 $shift\text{-}sharing (make\text{-}sharing x1 x2 x3) = make\text{-}sharing x3 x1 x2$
unfolding *shift-sharing-def*
by *simp*

lemma *reconstruct-share-nat*:
 $map\text{-}spmf (\lambda xs. (xs, reconstruct xs)) (share\text{-}nat x) = map\text{-}spmf (\lambda xs. (xs, x)) (share\text{-}nat x)$ **for** x
unfolding *share-nat-def* **by** (*auto cong: map-spmf-cong-simp*)

lemma *weight-share-nat*:
 $weight\text{-}spmf (share\text{-}nat x) = 1$
unfolding *share-nat-def weight-spmf-of-set vimage-def*
apply *clarsimp*
apply (*rule exI[where x=make-sharing x 0 0]*)
apply *simp*
done

```

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)

no-notation *m-inv* (*inv1* - [81] 80)

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

$$\begin{aligned}
\text{aby3-stack-sharing } r \ s &= \text{make-sharing}' \ r \ (\text{next-role } r) \ (\text{prev-role } r) \\
&\quad (\text{get-party } r \ s) \\
&\quad (\text{get-party } (\text{next-role } r) \ s + \text{get-party } (\text{prev-role } r) \ s) \\
&\quad 0
\end{aligned}$$

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

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

})

definition *aby3-shuffleR* :: *natL sharing list* \Rightarrow (*role-msg sharing* \times *natL sharing list*) *spmf* **where**

```

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

```

definition *aby3-shuffleF* :: *natL sharing list* \Rightarrow *natL sharing list* *spmf* **where**

```

aby3-shuffleF x = (do {
   $\pi$   $\leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {..length x}};
  let x1 = map reconstruct x;
  let x $\pi$  = permute-list  $\pi$  x1;
  y  $\leftarrow$  sequence-spmf (map share-nat x $\pi$ );
  return-spmf y
})

```

definition *S1* :: *natL list* \Rightarrow *natL list* \Rightarrow *role-msg* *spmf* **where**

```

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

   $\pi$ a  $\leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {..n}};
   $\pi$ c  $\leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {..n}};

  ya1::natL list  $\leftarrow$  sequence-spmf (replicate n (spmf-of-set UNIV));

  yb1::natL list  $\leftarrow$  sequence-spmf (replicate n (spmf-of-set UNIV));
  yb2::natL list  $\leftarrow$  sequence-spmf (replicate n (spmf-of-set UNIV));

```

— round 1

```

  let  $\zeta$ a1 = map2 (–) ya1 (permute-list  $\pi$ a x1);

```

— round 2

```

  let  $\zeta$ 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 $msg1 = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c)$;
 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 $msg2 = ((\zeta a2, \zeta b2, \zeta c2), x', \pi a, \pi b)$;
 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
 $\text{let } \zeta a3 = ya3;$

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

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

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

})

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

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

lemma $\text{case-prod-nesting-same}$:
 $\text{case-prod } (\lambda a \ b. f \text{ (case-prod } g \ x) \ a \ b) \ x = \text{case-prod } (\lambda a \ b. f \text{ (} g \ a \ b) \ a \ b) \ x$
by $(\text{cases } x) \ \text{simp}$

lemma zip-map-map-same :
 $\text{map } (\lambda x. (f \ x, g \ x)) \ xs = \text{zip } (\text{map } f \ xs) \ (\text{map } g \ xs)$
unfolding zip-map-map
unfolding zip-same-conv-map
by simp

lemma dup-map-eq :
 $\text{length } xs = \text{length } ys \implies (xs, \text{map2 } f \ ys \ xs) = (\lambda xys. (\text{map } \text{fst } xys, \text{map } \text{snd } xys))$
 $(\text{map2 } (\lambda x \ y. (y, f \ x \ y)) \ ys \ xs)$
by $(\text{auto simp: map-snd-zip[unfolded snd-def]})$

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

lemma map-spmf-cong2 :
assumes $p = \text{map-spmf } m \ q \ \wedge \ x. x \in \text{set-spmf } q \implies f \ (m \ x) = g \ x$
shows $\text{map-spmf } f \ p = \text{map-spmf } g \ q$
using assms **by** $(\text{simp add: spmf.map-comp cong: map-spmf-cong})$

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

shows $bind\text{-}spmf\ p\ f = bind\text{-}spmf\ q\ g$
using *assms* **by** (*simp* *add*: *map-spmf-conv-bind-spmf* *cong*: *bind-spmf-cong*)

lemma *map2-spmf-map2-sequence*:

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

apply (*induction* *xss* *yss* *rule*: *list-induct2*)

subgoal **by** *simp*

subgoal **premises** *IH* **for** *x* *xs* *y* *ys*

apply *simp*

apply (*fold* *IH*)

apply (*unfold* *pair-map-spmf*)

apply (*unfold* *spmf.map-comp*)

apply (*rule* *map-spmf-cong2*[**where** $m = \lambda((x,y),(xs,ys)). ((x,xs),(y,ys))$]])

subgoal

unfolding *pair-spmf-alt-def*

apply (*simp* *add*: *map-spmf-conv-bind-spmf*)

apply (*subst* *bind-commute-spmf*[**where** $q=y$]])

..

subgoal **by** *auto*

done

done

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*:

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

apply (*induction* *xss* *yss* *zss* *rule*: *list-induct3*)

subgoal **by** *simp*

subgoal **premises** *IH* **for** *x* *xs* *y* *ys* *z* *zs*

apply *simp*

apply (*fold* *IH*)

apply (*unfold* *pair-map-spmf*)

apply (*unfold* *spmf.map-comp*)

apply (*rule* *map-spmf-cong2*[**where** $m = \lambda((x,y,z),(xs,ys,zs)). ((x,xs),(y,ys),(z,zs))$]])

subgoal

unfolding *pair-spmf-alt-def*

apply (*simp* *add*: *map-spmf-conv-bind-spmf*)

apply (*subst* *bind-commute-spmf*[**where** $q=y$]])

apply (*subst* *bind-commute-spmf*[**where** $q=z$]])

apply (*subst* *bind-commute-spmf*[**where** $q=z$]])

..

subgoal **by** *auto*

done

done

lemma *in-pairD2*:

$x \in A \times B \implies \text{snd } x \in B$

by *auto*

lemma *list-map-cong2*:

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

unfolding *simp-implies-def*

by *simp*

lemma *map-swap-zip*:

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

apply (*induction xs arbitrary: ys*)

subgoal by *simp*

subgoal for $x \ xs \ ys$

by (*cases ys*) *auto*

done

lemma *inv-zero-sharing-sequence*:

$n = \text{length } x \implies$

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

$=$

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

proof –

assume $n: n = \text{length } x$

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

$=$

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

unfolding *sequence-map-return-spmf*

apply (*rule map-spmf-cong2* [**where** $m = \text{fst}$])

subgoal by *simp*

subgoal by (*auto simp: case-prod-unfold dest: in-pairD2*)

done

also have $\dots = \text{map-spmf} \ (\lambda \zeta xs. (\text{map } \text{fst } \zeta xs, \text{map } \text{snd } \zeta xs)) \ (\text{map2-spmf} \ (\text{map2} \ (\lambda \zeta \ x. (\zeta, \text{map-sharing2} \ (+)) \ x \ \zeta))) \ (\text{sequence-spmf} \ (\text{replicate } n \ \text{zero-sharing})) \ (\text{sequence-spmf} \ (\text{map } \text{return-spmf } x))$

apply (*unfold spmf.map-comp*)

apply (*rule map-spmf-cong* [*OF refl*])

using n **by** (*auto simp: case-prod-unfold comp-def set-sequence-spmf list-all2-iff map-swap-zip intro: list-map-cong2* [**where** $m = \text{prod.swap}$])

also have $\dots = \text{map-spmf} \ (\lambda \zeta xs. (\text{map } \text{fst } \zeta xs, \text{map } \text{snd } \zeta xs)) \ (\text{map2-spmf} \ (\text{map2}$

```

( $\lambda y x. (map-sharing2 (-) y x, y)) (sequence-spmf (map (share-nat \circ reconstruct)
x)) (sequence-spmf (map return-spmf x)))
  apply (rule arg-cong[where f=map-spmf -])
  using n   apply (simp add: map2-spmf-map2-sequence)
  apply (rule arg-cong[where f=sequence-spmf])
  apply (unfold list-eq-iff-nth-eq)
  apply safe
  subgoal by simp
  apply (simp add: )
  apply (subst map-spmf-cong2[where p=pair-spmf - (return-spmf -)])
    apply (rule pair-spmf-return-spmf2)
    apply simp
  apply (subst map-spmf-cong2[where p=pair-spmf - (return-spmf -)])
    apply (rule pair-spmf-return-spmf2)
    apply simp
  using inv-zero-sharing .

also have ... = map2-spmf ( $\lambda ys x. (map2 (map-sharing2 (-)) ys x, ys) (sequence-spmf
(map (share-nat \circ reconstruct) x)) (sequence-spmf (map return-spmf x))
  apply (unfold spmf.map-comp)
  apply (rule map-spmf-cong[OF refl])
  using n by (auto simp: case-prod-unfold comp-def set-sequence-spmf list-all2-iff
map-swap-zip intro: list-map-cong2[where m=prod.swap])

also have ... = map-spmf ( $\lambda ys. (map2 (map-sharing2 (-)) ys x, ys) (sequence-spmf
(map (share-nat \circ reconstruct) x))
  unfolding sequence-map-return-spmf
  apply (rule map-spmf-cong2[where m=fst, symmetric])
  subgoal by simp
  subgoal by (auto simp: case-prod-unfold dest: in-pairD2)
  done

finally show ?thesis .
qed$$$ 
```

```

lemma get-party-map-sharing2:
  get-party p \circ (case-prod (map-sharing2 f)) = case-prod f \circ map-prod (get-party
p) (get-party p)
  by auto

```

```

lemma map-map-prod-zip:
  map (map-prod f g) (zip xs ys) = zip (map f xs) (map g ys)
  by (simp add: map-prod-def zip-map-map)

```

```

lemma map-map-prod-zip':
  map (h \circ map-prod f g) (zip xs ys) = map h (zip (map f xs) (map g ys))
  by (simp add: map-prod-def zip-map-map)

```

```

lemma eq-map-spmf-conv:

```

assumes $\bigwedge x. f (f' x) = x f' = \text{inv } f \text{ map-spmf } f' x = y$
shows $x = \text{map-spmf } f y$
proof –
have *surj*: *surj* *f*
apply (*rule surjI*) **using** *assms(1)* .
have $\text{map-spmf } f (\text{map-spmf } f' x) = \text{map-spmf } f y$
unfolding *assms(3)* ..
thus *?thesis*
using *assms(1)* **by** (*simp add: spmf.map-comp surj-iff comp-def*)
qed

lemma *lift-map-spmf-pairs*:
 $\text{map2-spmf } f = F \implies \text{map-spmf } (\text{case-prod } f) (\text{pair-spmf } A B) = F A B$
by *auto*

lemma *measure-pair-spmf-times'*:
 $C = A \times B \implies \text{measure } (\text{measure-spmf } (\text{pair-spmf } p q)) C = \text{measure } (\text{measure-spmf } p) A * \text{measure } (\text{measure-spmf } q) B$
by (*simp add: measure-pair-spmf-times*)

lemma *map-spmf-pairs-tmp*:
 $\text{map-spmf } (\lambda(a,b,c,d,e,f,g). (a,e,b,f,c,g,d)) (\text{pair-spmf } A (\text{pair-spmf } B (\text{pair-spmf } C (\text{pair-spmf } D (\text{pair-spmf } E (\text{pair-spmf } F G))))))$
 $= (\text{pair-spmf } A (\text{pair-spmf } E (\text{pair-spmf } B (\text{pair-spmf } F (\text{pair-spmf } C (\text{pair-spmf } G D))))))$
apply (*rule spmf-eqI*)
apply (*clarsimp simp add: spmf-map*)
subgoal for *a e b f c g d*
apply (*subst measure-pair-spmf-times'* [**where** $A=\{a\}$]) **defer**
apply (*subst measure-pair-spmf-times'* [**where** $A=\{b\}$]) **defer**
apply (*subst measure-pair-spmf-times'* [**where** $A=\{c\}$]) **defer**
apply (*subst measure-pair-spmf-times'* [**where** $A=\{d\}$]) **defer**
apply (*subst measure-pair-spmf-times'* [**where** $A=\{e\}$]) **defer**
apply (*subst measure-pair-spmf-times'* [**where** $A=\{f\}$ and $B=\{g\}$]) **defer**
apply (*auto simp: spmf-conv-measure-spmf*)
done
done

lemma *case-case-prods-tmp*:
 $(\text{case case } x \text{ of } (a, b, c, d, e, f, g) \Rightarrow (a, e, b, f, c, g, d) \text{ of } (ya, yb, yc, yd, ye, yf, yg) \Rightarrow F ya yb yc yd ye yf yg)$
 $= (\text{case } x \text{ of } (a,b,c,d,e,f,g) \Rightarrow F a e b f c g d)$
by (*cases x*) *simp*

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

by (rule bind-spmf-cong[OF refl]) (simp add: set-spmf-of-set finite-permutations set-sequence-spmf[unfolded list-all2-iff])

lemma map-eq-iff-list-all2:
 $map\ f\ xs = map\ g\ ys \longleftrightarrow list\ all2\ (\lambda x\ y.\ f\ x = g\ y)\ xs\ ys$
apply (induction xs arbitrary: ys)
subgoal by auto
subgoal for x xs ys **by** (cases ys) (auto)
done

lemma bind-spmf-sequence-map-share-nat-cong:
 $(\bigwedge l.\ map\ reconstruct\ l = x \implies f\ l = g\ l)$
 $\implies bind\ spmf\ (sequence\ spmf\ (map\ share\ nat\ x))\ f = bind\ spmf\ (sequence\ spmf\ (map\ share\ nat\ x))\ g$
subgoal premises prems
apply (rule bind-spmf-cong[OF refl])
apply (rule prems)
unfolding set-sequence-spmf mem-Collect-eq
apply (simp add: map-eq-iff-list-all2[**where** g=id, simplified])
apply (simp add: list-all2-map2)
apply (erule list-all2-mono)
unfolding share-nat-def
by simp
done

lemma map-reconstruct-comp-eq-iff:
 $(\bigwedge x.\ x \in set\ xs \implies reconstruct\ (f\ x) = reconstruct\ x) \implies map\ (reconstruct \circ f)$
 $xs = map\ reconstruct\ xs$
by (induction xs) auto

lemma permute-list-replicate:
 $p\ permutes\ \{..
apply (fold map-replicate-const[**where** lst=[0..
apply (simp add: permute-list-map)
unfolding map-replicate-const
by simp$

lemma map2-minus-zero:
 $length\ xs = length\ ys \implies (\bigwedge y::natL.\ y \in set\ ys \implies y = 0) \implies map2\ (-)\ xs\ ys = xs$
by (induction xs ys rule: list-induct2) auto

lemma permute-comp-left-inj:
 $p\ permutes\ \{..
by (rule fun.inj-map) (rule permutes-inj-on)$

lemma permute-comp-left-inj-on:
 $p\ permutes\ \{..
using permute-comp-left-inj inj-on-subset **by** blast$

lemma *permute-comp-right-inj*:

p permutes $\{..<n\} \implies \text{inj } (\lambda p'. p' \circ p)$
using *inj-onI comp-id o-assoc permutes-surj surj-iff*
by (*smt (verit)*)

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

p permutes $\{..<n\} \implies \text{inj-on } (\lambda p'. p' \circ p) A$
using *permute-comp-right-inj inj-on-subset* **by** *blast*

lemma *permutes-inv-comp-left*:

p permutes $\{..<n\} \implies \text{inv } (\lambda p'. p \circ p') = (\lambda p'. \text{inv } p \circ p')$
by (*rule inv-unique-comp; rule ext, simp add: permutes-inv-o comp-assoc[symmetric]*)

lemma *permutes-inv-comp-right*:

p permutes $\{..<n\} \implies \text{inv } (\lambda p'. p' \circ p) = (\lambda p'. p' \circ \text{inv } p)$
by (*rule inv-unique-comp; rule ext, simp add: permutes-inv-o comp-assoc*)

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

πa permutes $\{..<n\} \implies \pi b$ permutes $\{..<n\} \implies \text{inv } (\lambda p'. \pi a \circ p' \circ \pi b) = (\lambda p'. \text{inv } \pi a \circ p' \circ \text{inv } \pi b)$
by (*rule inv-unique-comp; rule ext, simp add: permutes-inv-o comp-assoc, simp add: permutes-inv-o comp-assoc[symmetric]*)

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

πa permutes $\{..<n\} \implies \pi b$ permutes $\{..<n\} \implies \text{inv } (\lambda p'. \pi a \circ \pi b \circ p') = (\lambda p'. \text{inv } \pi b \circ \text{inv } \pi a \circ p')$
by (*rule inv-unique-comp; rule ext, simp add: permutes-inv-o comp-assoc, simp add: permutes-inv-o comp-assoc[symmetric]*)

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

πa permutes $\{..<n\} \implies \pi b$ permutes $\{..<n\} \implies \text{inv } (\lambda p'. p' \circ \pi a \circ \pi b) = (\lambda p'. p' \circ \text{inv } \pi b \circ \text{inv } \pi a)$
by (*rule inv-unique-comp; rule ext, simp add: permutes-inv-o comp-assoc, simp add: permutes-inv-o comp-assoc[symmetric]*)

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 –

have $*$: $(\lambda \pi. \pi a \circ \pi \circ \pi b) = (\lambda \pi'. \pi a \circ \pi') \circ (\lambda \pi. \pi \circ \pi b)$

by (*simp add: comp-def*)

then show *?thesis*

apply (*fold image-Collect*)

apply (*unfold **)

apply (*fold image-comp*)

apply (*subst image-Collect*)

apply (*unfold image-compose-permutations-right[OF assms(2)]*)

apply (*subst image-Collect*)
apply (*unfold image-compose-permutations-left[OF assms(1)]*)
 ..
qed

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

fixes S
assumes πa *permutes* S πb *permutes* S
shows $\{\pi a \circ \pi b \circ \pi \mid \pi. \pi \text{ permutes } S\} = \{\pi. \pi \text{ permutes } S\}$
using *image-compose-permutations-left image-compose-permutations-right*
proof –
have $*$: $(\lambda \pi. \pi a \circ \pi b \circ \pi) = (\lambda \pi'. \pi a \circ \pi') \circ (\lambda \pi. \pi b \circ \pi)$
by (*simp add: comp-def*)
show *?thesis*
apply (*fold image-Collect*)
apply (*unfold **)
apply (*fold image-comp*)
apply (*subst image-Collect*)
apply (*unfold image-compose-permutations-left[OF assms(2)]*)
apply (*subst image-Collect*)
apply (*unfold image-compose-permutations-left[OF assms(1)]*)
 ..
qed

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

fixes S
assumes πa *permutes* S πb *permutes* S
shows $\{\pi \circ \pi a \circ \pi b \mid \pi. \pi \text{ permutes } S\} = \{\pi. \pi \text{ permutes } S\}$
using *image-compose-permutations-left image-compose-permutations-right*
proof –
have $*$: $(\lambda \pi. \pi \circ \pi a \circ \pi b) = (\lambda \pi. \pi \circ \pi b) \circ (\lambda \pi'. \pi' \circ \pi a)$
by (*simp add: comp-def*)
show *?thesis*
apply (*fold image-Collect*)
apply (*unfold **)
apply (*fold image-comp*)
apply (*subst image-Collect*)
apply (*unfold image-compose-permutations-right[OF assms(1)]*)
apply (*subst image-Collect*)
apply (*unfold image-compose-permutations-right[OF assms(2)]*)
 ..
qed

lemma *random-perm-middle*:

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

```

n) (pair-spmf (random-perm n) (random-perm n))
  (is ?lhs = ?rhs)
proof -
  have ?lhs = (do { $\pi a \leftarrow$  random-perm n;  $\pi c \leftarrow$  random-perm n;  $(\pi b, p) \leftarrow$ 
map-spmf ( $\lambda \pi b. (\pi b, \pi a \circ \pi b \circ \pi c)$ ) (random-perm n); return-spmf (( $\pi a, \pi b, \pi c$ ),
p)}})
  unfolding map-spmf-conv-bind-spmf pair-spmf-alt-def
  apply simp
  apply (subst (4) bind-commute-spmf)
  ..
also
  have ... = (do { $\pi a \leftarrow$  random-perm n;  $\pi c \leftarrow$  random-perm n; map-spmf ( $\lambda p.
((\pi a, inv \pi a \circ p \circ inv \pi c, \pi c), p)$ ) (random-perm n)}})
  unfolding random-perm-def
  supply [intro!] =
    bind-spmf-permutes-cong
  apply rule+
  apply (subst inv-uniform)
  subgoal for  $\pi a \pi c$ 
    apply (rule inj-compose[unfolded comp-def, where  $f = \lambda p. p \circ \pi c$ ])
    subgoal by (rule permute-comp-right-inj-on)
    subgoal by (rule permute-comp-left-inj-on)
    done
  apply (simp add: permutes-inv-comp-left-right map-spmf-conv-bind-spmf im-
age-Collect image-compose-permutations-left-right)
  done
also
  have ... = ?rhs
  unfolding map-spmf-conv-bind-spmf pair-spmf-alt-def
  apply (subst (2) bind-commute-spmf)
  apply (subst (1) bind-commute-spmf)
  apply simp
  done
finally show ?thesis .
qed

```

lemma random-perm-right:

```

defines random-perm n  $\equiv$  spmf-of-set { $\pi. \pi$  permutes {.. $n::nat$ }}
shows
  map-spmf ( $\lambda(\pi a, \pi b, \pi c). ((\pi a, \pi b, \pi c), \pi a \circ \pi b \circ \pi c)$ ) (pair-spmf (random-perm
n) (pair-spmf (random-perm n) (random-perm n)))
  = map-spmf ( $\lambda(\pi, \pi a, \pi b). ((\pi a, \pi b, inv \pi b \circ inv \pi a \circ \pi), \pi)$ ) (pair-spmf (random-perm
n) (pair-spmf (random-perm n) (random-perm n)))
  (is ?lhs = ?rhs)
proof -
  have ?lhs = (do { $\pi a \leftarrow$  random-perm n;  $\pi b \leftarrow$  random-perm n;  $(\pi c, \pi) \leftarrow$ 
map-spmf ( $\lambda \pi c. (\pi c, \pi a \circ \pi b \circ \pi c)$ ) (random-perm n); return-spmf (( $\pi a, \pi b, \pi c$ ),
 $\pi$ )}})
  unfolding map-spmf-conv-bind-spmf pair-spmf-alt-def

```

```

  by simp
also
  have ... = (do { $\pi a \leftarrow \text{random-perm } n$ ;  $\pi b \leftarrow \text{random-perm } n$ ;  $\text{map-spmf } (\lambda \pi. ((\pi a, \pi b, \text{inv } \pi b \circ \text{inv } \pi a \circ \pi), \pi)) (\text{random-perm } n)$ })
    unfolding random-perm-def
    supply [intro!] =
      bind-spmf-permutes-cong
    apply rule+
    apply (subst inv-uniform)
  subgoal
    apply (rule permute-comp-left-inj-on)
    using permutes-compose .
    apply (simp add: permutes-inv-comp-left-left map-spmf-conv-bind-spmf image-Collect image-compose-permutations-left-left)
  done
also
  have ... = ?rhs
    unfolding map-spmf-conv-bind-spmf pair-spmf-alt-def
    apply (subst (2) bind-commute-spmf)
    apply (subst (1) bind-commute-spmf)
    apply simp
  done
  finally show ?thesis .
qed

```

lemma *random-perm-left*:

```

  defines random-perm  $n \equiv \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n::\text{nat}\}\}$ 
  shows
    map-spmf ( $\lambda(\pi a, \pi b, \pi c). ((\pi a, \pi b, \pi c), \pi a \circ \pi b \circ \pi c)$ ) (pair-spmf (random-perm  $n$ ) (pair-spmf (random-perm  $n$ ) (random-perm  $n$ )))
    = map-spmf ( $\lambda(\pi, \pi b, \pi c). ((\pi \circ \text{inv } \pi c \circ \text{inv } \pi b, \pi b, \pi c), \pi)$ ) (pair-spmf (random-perm  $n$ ) (pair-spmf (random-perm  $n$ ) (random-perm  $n$ )))
    (is ?lhs = ?rhs)
  proof -
    have ?lhs = (do { $\pi b \leftarrow \text{random-perm } n$ ;  $\pi c \leftarrow \text{random-perm } n$ ;  $(\pi a, \pi) \leftarrow \text{map-spmf } (\lambda \pi a. (\pi a, \pi a \circ \pi b \circ \pi c)) (\text{random-perm } n)$ ;  $\text{return-spmf } ((\pi a, \pi b, \pi c), \pi)$ })
      unfolding map-spmf-conv-bind-spmf pair-spmf-alt-def
      apply simp
      apply (subst (4) bind-commute-spmf)
      apply (subst (3) bind-commute-spmf)
    ..
  also
    have ... = (do { $\pi b \leftarrow \text{random-perm } n$ ;  $\pi c \leftarrow \text{random-perm } n$ ;  $\text{map-spmf } (\lambda \pi. ((\pi \circ \text{inv } \pi c \circ \text{inv } \pi b, \pi b, \pi c), \pi)) (\text{random-perm } n)$ })
      unfolding random-perm-def
      supply [intro!] =
        bind-spmf-permutes-cong
      apply rule+

```



```

apply (subst inv-uniform)
subgoal
  apply (unfold comp-assoc)
  apply (rule permute-comp-right-inj-on)
  using permutes-compose .
  apply (simp add: permutes-inv-comp-right-right map-spmf-conv-bind-spmf im-
age-Collect image-compose-permutations-right-right)
  done
also
have ... = ?rhs
  unfolding map-spmf-conv-bind-spmf pair-spmf-alt-def
  apply (subst (2) bind-commute-spmf)
  apply (subst (1) bind-commute-spmf)
  apply simp
  done
finally show ?thesis .
qed

```

```

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

```

```

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

```

```

proof -
have
  sequence-spmf (map share-nat xs) = (do {
    let  $n = \text{length } xs$ ;
    let random-seq = sequence-spmf (replicate  $n$  (spmof-of-set UNIV));
    (xs, dp, dpn)  $\leftarrow$  pair-spmf (sequence-spmf (map return-spmf xs)) (pair-spmf
random-seq random-seq);
    return-spmf (map3 ( $\lambda x a b. \text{make-sharing}' r1 r2 r3 a b (x - (a + b))$ )) xs dp
dpn)
  })
  apply (unfold Let-def)
  apply (unfold case-prod-return-spmf)
  apply (fold map-spmf-conv-bind-spmf)
  apply (subst map3-spmf-map3-sequence)
  subgoal by simp
  subgoal by simp
  apply (rule arg-cong[where  $f = \text{sequence-spmf}$ ])

```

```

apply (unfold map-eq-iff-list-all2)
apply (rule list-all2-all-nthI)
subgoal by simp
unfolding share-nat-def-calc'[OF assms]
apply (auto simp: map-spmf-conv-bind-spmf pair-spmf-alt-def)
done
also have ... = ?rhs
  by (auto simp: pair-spmf-alt-def sequence-map-return-spmf)
finally show ?thesis .
qed

```

```

lemma reconstruct-stack-sharing-eq-reconstruct:
  reconstruct  $\circ$  aby3-stack-sharing  $r =$  reconstruct
unfolding aby3-stack-sharing-def reconstruct-def
by (cases  $r$ ) (auto simp: make-sharing'-sel)

```

```

lemma map2-ignore1:
  length  $xs =$  length  $ys \implies$  map2 ( $\lambda$ -.  $f$ )  $xs$   $ys =$  map  $f$   $ys$ 
apply (unfold map-eq-iff-list-all2)
apply (rule list-all2-all-nthI)
by auto

```

```

lemma map2-ignore2:
  length  $xs =$  length  $ys \implies$  map2 ( $\lambda$   $a$   $b$ .  $f$   $a$ )  $xs$   $ys =$  map  $f$   $xs$ 
apply (unfold map-eq-iff-list-all2)
apply (rule list-all2-all-nthI)
by auto

```

```

lemma map-sequence-share-nat-reconstruct:
  map-spmf ( $\lambda$   $x$ . ( $x$ , map reconstruct  $x$ )) (sequence-spmf (map share-nat  $y$ )) =
  map-spmf ( $\lambda$   $x$ . ( $x$ ,  $y$ )) (sequence-spmf (map share-nat  $y$ ))
apply (unfold map-spmf-conv-bind-spmf)
apply (rule bind-spmf-cong[OF refl])
apply (auto simp: set-sequence-spmf list-eq-iff-nth-eq list-all2-conv-all-nth share-nat-def)
done

```

```

theorem shuffle-secrecy:
assumes
  is-uniform-sharing-list  $x$ -dist
shows
  (do {
     $x \leftarrow$   $x$ -dist;
    ( $msg$ ,  $y$ )  $\leftarrow$  aby3-shuffleR  $x$ ;
    return-spmf (map (get-party  $r$ )  $x$ ,
      get-party  $r$   $msg$ ,
       $y$ )
  })

```

```

})
=
(do {
  x ← x-dist;
  y ← aby3-shuffleF x;
  let xr = map (get-party r) x;
  let yr = map (get-party r) y;
  msg ← S r xr yr;
  return-spmf (xr, msg, y)
})
(is ?lhs = ?rhs)

```

proof –

obtain xs **where** $xs: x\text{-dist} = xs \ggg (\text{sequence-spmf} \circ \text{map share-nat})$
using *assms unfolding is-uniform-sharing-list-def* **by** *auto*

have *left-unfolded*:

```

(do {
  x ← x-dist;
  (msg, y) ← aby3-shuffleR x;
  return-spmf (map (get-party r) x, get-party r msg, y)})
=
(do {
  xs ← xs;
  x ← sequence-spmf (map share-nat xs);

```

— round 1

```

let n = length x;
ζa ← sequence-spmf (replicate n zero-sharing);
πa ← spmf-of-set {π. π permutes {.. $n$ }};
let x' = map (get-party (prev-role Party1)) x;
let y' = map (aby3-stack-sharing Party1) x;
let a = map2 (map-sharing2 (+)) (permute-list πa y') ζa;

```

— round 2

```

let n = length a;
ζb ← sequence-spmf (replicate n zero-sharing);
πb ← spmf-of-set {π. π permutes {.. $n$ }};
let a' = map (get-party (prev-role Party2)) a;
let y' = map (aby3-stack-sharing Party2) a;
let b = map2 (map-sharing2 (+)) (permute-list πb y') ζb;

```

— round 3

```

let n = length b;
ζc ← sequence-spmf (replicate n zero-sharing);
πc ← spmf-of-set {π. π permutes {.. $n$ }};
let b' = map (get-party (prev-role Party3)) b;
let y' = map (aby3-stack-sharing Party3) b;
let c = map2 (map-sharing2 (+)) (permute-list πc y') ζc;

```

```

    let msg1 = ((map (get-party Party1) ζa, map (get-party Party1) ζb, map
(get-party Party1) ζc), b', πa, πc);
    let msg2 = ((map (get-party Party2) ζa, map (get-party Party2) ζb, map
(get-party Party2) ζc), x', πa, πb);
    let msg3 = ((map (get-party Party3) ζa, map (get-party Party3) ζb, map
(get-party Party3) ζc), a', πb, πc);
    let msg = make-sharing msg1 msg2 msg3;
    return-spmf (map (get-party r) x, get-party r msg, c)
  })
unfolding xs aby3-shuffleR-def aby3-do-permute-def
by (auto simp: case-prod-unfold Let-def)

```

also have clarify-length:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

```

— round 1

```

ζa ← sequence-spmf (replicate n zero-sharing);
πa ← spmf-of-set {π. π permutes {.. $n$ }};
let x' = map (get-party (prev-role Party1)) x;
let y' = map (aby3-stack-sharing Party1) x;
let a = map2 (map-sharing2 (+)) (permute-list πa y') ζa;

```

— round 2

```

ζb ← sequence-spmf (replicate n zero-sharing);
πb ← spmf-of-set {π. π permutes {.. $n$ }};
let a' = map (get-party (prev-role Party2)) a;
let y' = map (aby3-stack-sharing Party2) a;
let b = map2 (map-sharing2 (+)) (permute-list πb y') ζb;

```

— round 3

```

ζc ← sequence-spmf (replicate n zero-sharing);
πc ← spmf-of-set {π. π permutes {.. $n$ }};
let b' = map (get-party (prev-role Party3)) b;
let y' = map (aby3-stack-sharing Party3) b;
let c = map2 (map-sharing2 (+)) (permute-list πc y') ζc;

```

```

    let msg1 = ((map (get-party Party1) ζa, map (get-party Party1) ζb, map
(get-party Party1) ζc), b', πa, πc);
    let msg2 = ((map (get-party Party2) ζa, map (get-party Party2) ζb, map
(get-party Party2) ζc), x', πa, πb);
    let msg3 = ((map (get-party Party3) ζa, map (get-party Party3) ζb, map
(get-party Party3) ζc), a', πb, πc);
    let msg = make-sharing msg1 msg2 msg3;
    return-spmf (map (get-party r) x, get-party r msg, c)
  })
supply [intro!] =

```

```

bind-spmf-cong[OF refl]
let-cong[OF refl]
bind-spmf-sequence-map-cong
bind-spmf-sequence-replicate-cong
bind-spmf-permutes-cong
by (auto simp: Let-def)

```

also have *hoist-permutations*:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};

```

— round 1

```

let x' = map (get-party (prev-role Party1)) x;
let y' = map (aby3-stack-sharing Party1) x;
ζa ← sequence-spmf (replicate n zero-sharing);
let a = map2 (map-sharing2 (+)) (permute-list πa y') ζa;

```

— round 2

```

let a' = map (get-party (prev-role Party2)) a;
let y' = map (aby3-stack-sharing Party2) a;
ζb ← sequence-spmf (replicate n zero-sharing);
let b = map2 (map-sharing2 (+)) (permute-list πb y') ζb;

```

— round 3

```

let b' = map (get-party (prev-role Party3)) b;
let y' = map (aby3-stack-sharing Party3) b;
ζc ← sequence-spmf (replicate n zero-sharing);
let c = map2 (map-sharing2 (+)) (permute-list πc y') ζc;

let msg1 = ((map (get-party Party1) ζa, map (get-party Party1) ζb, map
(get-party Party1) ζc), b', πa, πc);
let msg2 = ((map (get-party Party2) ζa, map (get-party Party2) ζb, map
(get-party Party2) ζc), x', πa, πb);
let msg3 = ((map (get-party Party3) ζa, map (get-party Party3) ζb, map
(get-party Party3) ζc), a', πb, πc);
let msg = make-sharing msg1 msg2 msg3;
return-spmf (map (get-party r) x, get-party r msg, c)
})
apply (simp add: Let-def)
apply (subst (1) bind-commute-spmf[where q=spmf-of-set -])
apply (subst (2) bind-commute-spmf[where q=spmf-of-set -])
apply (subst (2) bind-commute-spmf[where q=spmf-of-set -])
apply (subst (3) bind-commute-spmf[where q=spmf-of-set -])

```

```

apply (subst (3) bind-commute-spmf[where q=spmf-of-set -])
apply (subst (3) bind-commute-spmf[where q=spmf-of-set -])
by simp

```

also have *hoist-permutations*:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};

```

— round 1

```

  let x' = map (get-party (prev-role Party1)) x;
  let y' = map (aby3-stack-sharing Party1) x;
  a ← sequence-spmf (map (share-nat ∘ reconstruct) (permute-list πa y'));
  let ζa = map2 (map-sharing2 (-)) a (permute-list πa y');

```

— round 2

```

  let a' = map (get-party (prev-role Party2)) a;
  let y' = map (aby3-stack-sharing Party2) a;
  b ← sequence-spmf (map (share-nat ∘ reconstruct) (permute-list πb y'));
  let ζb = map2 (map-sharing2 (-)) b (permute-list πb y');

```

— round 3

```

  let b' = map (get-party (prev-role Party3)) b;
  let y' = map (aby3-stack-sharing Party3) b;
  c ← sequence-spmf (map (share-nat ∘ reconstruct) (permute-list πc y'));
  let ζc = map2 (map-sharing2 (-)) c (permute-list πc y');

  let msg1 = ((map (get-party Party1) ζa, map (get-party Party1) ζb, map
(get-party Party1) ζc), b', πa, πc);
  let msg2 = ((map (get-party Party2) ζa, map (get-party Party2) ζb, map
(get-party Party2) ζc), x', πa, πb);
  let msg3 = ((map (get-party Party3) ζa, map (get-party Party3) ζb, map
(get-party Party3) ζc), a', πb, πc);
  let msg = make-sharing msg1 msg2 msg3;
  return-spmf (map (get-party r) x, get-party r msg, c)
})
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

```

```

apply rule+
apply (subst hoist-map-spmf[where s=sequence-spmf (replicate - -) and f =
map2 (map-sharing2 (+)) -])
apply (subst hoist-map-spmf'[where s=sequence-spmf (map - -) and f = λys.
map2 (map-sharing2 (-)) ys -])
apply (subst inv-zero-sharing-sequence)
subgoal by simp
apply (unfold map-spmf-conv-bind-spmf)
apply (unfold bind-spmf-assoc)
apply (unfold return-bind-spmf)
apply rule+
apply (subst (1 12) Let-def)
apply rule+

```

```

apply (subst hoist-map-spmf[where s=sequence-spmf (replicate - -) and f =
map2 (map-sharing2 (+)) -])
apply (subst hoist-map-spmf'[where s=sequence-spmf (map - -) and f = λys.
map2 (map-sharing2 (-)) ys -])
apply (subst inv-zero-sharing-sequence)
subgoal by simp
apply (unfold map-spmf-conv-bind-spmf)
apply (unfold bind-spmf-assoc)
apply (unfold return-bind-spmf)
apply rule+
apply (subst (1 9) Let-def)
apply rule+

```

```

apply (subst hoist-map-spmf[where s=sequence-spmf (replicate - -) and f =
map2 (map-sharing2 (+)) -])
apply (subst hoist-map-spmf'[where s=sequence-spmf (map - -) and f = λys.
map2 (map-sharing2 (-)) ys -])
apply (subst inv-zero-sharing-sequence)
subgoal by simp
apply (unfold map-spmf-conv-bind-spmf)
apply (unfold bind-spmf-assoc)
apply (unfold return-bind-spmf)
apply rule+
apply (subst (1 6) Let-def)
apply rule+

```

done

also have reconstruct:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  π a ← spmf-of-set {π. π permutes {.. $n$ }};

```

```

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

— round 1
  let  $x' = \text{map } (\text{get-party } (\text{prev-role } \text{Party1})) x;$ 
  let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party1}) x;$ 
   $a \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } \pi a \text{ } xs));$ 
  let  $\zeta a = \text{map2 } (\text{map-sharing2 } (-)) a (\text{permute-list } \pi a \text{ } y');$ 

— round 2
  let  $a' = \text{map } (\text{get-party } (\text{prev-role } \text{Party2})) a;$ 
  let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party2}) a;$ 
   $b \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b) \text{ } xs));$ 
  let  $\zeta b = \text{map2 } (\text{map-sharing2 } (-)) b (\text{permute-list } \pi b \text{ } y');$ 

— round 3
  let  $b' = \text{map } (\text{get-party } (\text{prev-role } \text{Party3})) b;$ 
  let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party3}) b;$ 
   $c \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b \circ \pi c) \text{ } xs));$ 
  let  $\zeta c = \text{map2 } (\text{map-sharing2 } (-)) c (\text{permute-list } \pi c \text{ } y');$ 

  let  $\text{msg1} = ((\text{map } (\text{get-party } \text{Party1}) \zeta a, \text{map } (\text{get-party } \text{Party1}) \zeta b, \text{map}$ 
   $(\text{get-party } \text{Party1}) \zeta c), b', \pi a, \pi c);$ 
  let  $\text{msg2} = ((\text{map } (\text{get-party } \text{Party2}) \zeta a, \text{map } (\text{get-party } \text{Party2}) \zeta b, \text{map}$ 
   $(\text{get-party } \text{Party2}) \zeta c), x', \pi a, \pi b);$ 
  let  $\text{msg3} = ((\text{map } (\text{get-party } \text{Party3}) \zeta a, \text{map } (\text{get-party } \text{Party3}) \zeta b, \text{map}$ 
   $(\text{get-party } \text{Party3}) \zeta c), a', \pi b, \pi c);$ 
  let  $\text{msg} = \text{make-sharing } \text{msg1 } \text{msg2 } \text{msg3};$ 
  return-spmf (map (get-party r) x, get-party r msg, c)
})

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong
  bind-spmf-sequence-map-share-nat-cong

apply rule+
apply (subst list.map-comp[symmetric])
apply (rule bind-spmf-cong)
subgoal by (auto simp:permute-list-map[symmetric] map-reconstruct-comp-eq-iff
reconstruct-def set-sequence-spmf[unfolded list-all2-iff] make-sharing'-sel reconstruct-stack-sharing-eq-reconstru
comp-assoc)
apply rule+
apply (subst list.map-comp[symmetric])
apply (rule bind-spmf-cong)

```



```

subgoal for  $x\ l\ xa\ \pi a\ \pi b\ \pi c\ xb\ xc\ xd\ xe\ xf\ xg$ 
  apply (subst permute-list-map[symmetric] )
  subgoal by (auto simp add: set-sequence-spmf[unfolded list-all2-iff])
  apply simp
  apply (subst map-reconstruct-comp-eq-iff)
subgoal by (simp add: reconstruct-def make-sharing'-sel aby3-stack-sharing-def)
  unfolding set-sequence-spmf mem-Collect-eq
  unfolding list-all2-map2
  apply (subst map-eq-iff-list-all2[where  $f=\text{reconstruct}$  and  $g=id$  and  $xs=xd$ 
and  $ys=\text{permute-list } \pi a\ x,$  simplified, THEN iffD2])
  subgoal by (erule list-all2-mono) (simp add: share-nat-def)
  apply (subst permute-list-compose)
  subgoal by auto
  ..
apply rule+
apply (subst list.map-comp[symmetric])
apply (rule bind-spmf-cong)
subgoal for  $x\ l\ xa\ \pi a\ \pi b\ \pi c\ xb\ xc\ xd\ xe\ xf\ xg\ xh\ xi\ xj\ xk$ 
  apply (subst permute-list-map[symmetric] )
  subgoal by (auto simp add: set-sequence-spmf[unfolded list-all2-iff])
  apply simp
  apply (subst map-reconstruct-comp-eq-iff)
subgoal by (simp add: reconstruct-def make-sharing'-sel aby3-stack-sharing-def)
  unfolding set-sequence-spmf mem-Collect-eq
  unfolding list-all2-map2
  apply (subst map-eq-iff-list-all2[where  $f=\text{reconstruct}$  and  $g=id$  and  $xs=xh$ 
and  $ys=\text{permute-list } (\pi a \circ \pi b)\ x,$  simplified, THEN iffD2])
  subgoal by (erule list-all2-mono) (simp add: share-nat-def)
  apply (subst permute-list-compose[symmetric])
  subgoal by auto
  ..
  ..

```

also have *hoist*:

```

... = (do {
   $xs \leftarrow xs;$ 
   $let\ n = \text{length } xs;$ 
   $x \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } xs);$ 

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

   $a \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } \pi a\ xs));$ 
   $b \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b)\ xs));$ 
   $c \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b \circ \pi c)\ xs));$ 

```

— round 1

```

   $let\ x' = \text{map } (\text{get-party } (\text{prev-role } \text{Party1}))\ x;$ 

```

```

    let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party1}) \ x;$ 
    let  $\zeta a = \text{map2 } (\text{map-sharing2 } (-)) \ a \ (\text{permute-list } \pi a \ y');$ 

— round 2
    let  $a' = \text{map } (\text{get-party } (\text{prev-role } \text{Party2})) \ a;$ 
    let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party2}) \ a;$ 
    let  $\zeta b = \text{map2 } (\text{map-sharing2 } (-)) \ b \ (\text{permute-list } \pi b \ y');$ 

— round 3
    let  $b' = \text{map } (\text{get-party } (\text{prev-role } \text{Party3})) \ b;$ 
    let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party3}) \ b;$ 
    let  $\zeta c = \text{map2 } (\text{map-sharing2 } (-)) \ c \ (\text{permute-list } \pi c \ y');$ 

    let  $\text{msg1} = ((\text{map } (\text{get-party } \text{Party1}) \ \zeta a, \text{map } (\text{get-party } \text{Party1}) \ \zeta b, \text{map}$ 
     $(\text{get-party } \text{Party1}) \ \zeta c), b', \pi a, \pi c);$ 
    let  $\text{msg2} = ((\text{map } (\text{get-party } \text{Party2}) \ \zeta a, \text{map } (\text{get-party } \text{Party2}) \ \zeta b, \text{map}$ 
     $(\text{get-party } \text{Party2}) \ \zeta c), x', \pi a, \pi b);$ 
    let  $\text{msg3} = ((\text{map } (\text{get-party } \text{Party3}) \ \zeta a, \text{map } (\text{get-party } \text{Party3}) \ \zeta b, \text{map}$ 
     $(\text{get-party } \text{Party3}) \ \zeta c), a', \pi b, \pi c);$ 
    let  $\text{msg} = \text{make-sharing } \text{msg1 } \text{msg2 } \text{msg3};$ 
    return-spmf (map (get-party r) x, get-party r msg, c)
  })
unfolding Let-def ..
finally have hoisted-save: ?lhs = ... .
let ?hoisted = ...

{ assume  $r: r = \text{Party1}$ 
  have project-to-Party1:
  ?hoisted = (do {
     $xs \leftarrow xs;$ 
     $let \ n = \text{length } xs;$ 
     $x \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } xs);$ 

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

     $a \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } \pi a \ xs));$ 
     $b \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b) \ xs));$ 
     $c \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b \circ \pi c) \ xs));$ 

— round 1
    let  $x' = \text{map } (\text{get-party } \text{Party3}) \ x;$ 
    let  $y' = \text{map } (\text{aby3-stack-sharing } \text{Party1}) \ x;$ 
    let  $\zeta a1 = \text{map } (\text{case-prod } (-)) \ (\text{zip } (\text{map } (\text{get-party } \text{Party1}) \ a) \ (\text{map}$ 
     $(\text{get-party } \text{Party1}) \ (\text{permute-list } \pi a \ y')));$ 

— round 2
    let  $a' = \text{map } (\text{get-party } \text{Party1}) \ a;$ 

```

$let\ y' = map\ (aby3-stack-sharing\ Party2)\ a;$
 $let\ \zeta b1 = map\ (case-prod\ (-))\ (zip\ (map\ (get-party\ Party1)\ b)\ (map\ (get-party\ Party1)\ (permutate-list\ \pi b\ y')));$

— round 3

$let\ b' = map\ (get-party\ Party2)\ b;$
 $let\ y' = map\ (aby3-stack-sharing\ Party3)\ b;$
 $let\ \zeta c1 = map\ (case-prod\ (-))\ (zip\ (map\ (get-party\ Party1)\ c)\ (map\ (get-party\ Party1)\ (permutate-list\ \pi c\ y')));$

$let\ msg1 = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c);$
 $return-spmf\ (map\ (get-party\ Party1)\ x, msg1, c)$

})

by (*simp add: r Let-def get-party-map-sharing2 map-map-prod-zip'*)

also have *project-to-Party1*:

$\dots = (do\ \{$
 $xs \leftarrow xs;$
 $let\ n = length\ xs;$
 $x \leftarrow sequence-spmf\ (map\ share-nat\ xs);$

$\pi a \leftarrow spmf-of-set\ \{\pi. \pi\ permutates\ \{..<n\}\};$
 $\pi b \leftarrow spmf-of-set\ \{\pi. \pi\ permutates\ \{..<n\}\};$
 $\pi c \leftarrow spmf-of-set\ \{\pi. \pi\ permutates\ \{..<n\}\};$

$a \leftarrow sequence-spmf\ (map\ share-nat\ (permutate-list\ \pi a\ xs));$
 $b \leftarrow sequence-spmf\ (map\ share-nat\ (permutate-list\ (\pi a \circ \pi b)\ xs));$
 $c \leftarrow sequence-spmf\ (map\ share-nat\ (permutate-list\ (\pi a \circ \pi b \circ \pi c)\ xs));$

— round 1

$let\ x' = map\ (get-party\ Party3)\ x;$
 $let\ y' = map\ (aby3-stack-sharing\ Party1)\ x;$
 $let\ \zeta a1 = map2\ (-)\ (map\ (get-party\ Party1)\ a)\ (permutate-list\ \pi a\ (map\ (get-party\ Party1)\ y')));$

— round 2

$let\ a' = map\ (get-party\ Party1)\ a;$
 $let\ y' = map\ (aby3-stack-sharing\ Party2)\ a;$
 $let\ \zeta b1 = map2\ (-)\ (map\ (get-party\ Party1)\ b)\ (permutate-list\ \pi b\ (map\ (get-party\ Party1)\ y')));$

— round 3

$let\ b' = map\ (get-party\ Party2)\ b;$
 $let\ y' = map\ (aby3-stack-sharing\ Party3)\ b;$
 $let\ \zeta c1 = map2\ (-)\ (map\ (get-party\ Party1)\ c)\ (permutate-list\ \pi c\ (map\ (get-party\ Party1)\ y')));$

$let\ msg1 = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c);$
 $return-spmf\ (map\ (get-party\ Party1)\ x, msg1, c)$

```

})
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp
..

also have reduce-Lets:
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  πa ← spmf-of-set {π. π permutes {..

```

```

    let msg1 = ((ζ a1, ζ b1, ζ c1), b', π a, π c);
    return-spmf (map (get-party Party1) x, msg1, c)
  })
  unfolding Let-def
  unfolding aby3-stack-sharing-def
by (simp add: comp-def make-sharing'-sel map-replicate-const zip-map-map-same[symmetric])

also have simplify-minus-zero:
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  π a ← spmf-of-set {π. π permutes {.. $n$ }};
  π b ← spmf-of-set {π. π permutes {.. $n$ }};
  π c ← spmf-of-set {π. π permutes {.. $n$ }};

  a ← sequence-spmf (map share-nat (permute-list π a xs));
  b ← sequence-spmf (map share-nat (permute-list (π a ∘ π b) xs));
  c ← sequence-spmf (map share-nat (permute-list (π a ∘ π b ∘ π c) xs));

— round 1
  let ζ a1 = map2 (−) (map (get-party Party1) a) (permute-list π a (map
(get-party Party1) x));

— round 2
  let ζ b1 = (map (get-party Party1) b);

— round 3
  let b' = map (get-party Party2) b;
  let ζ c1 = map2 (−) (map (get-party Party1) c) (permute-list π c (map2
(+) (map (get-party Party1) b) (map (get-party Party2) b)));

  let msg1 = ((ζ a1, ζ b1, ζ c1), b', π a, π c);
  return-spmf (map (get-party Party1) x, msg1, c)
})
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

apply rule+
apply (subst permute-list-replicate)
subgoal by simp

apply (subst map2-minus-zero)

```

subgoal by simp
subgoal by simp

..

also have break-perms-1:

```
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);
```

```
  (πa,πb,πc) ← pair-spmf (spmof-of-set {π. π permutes {.. $n$ }}) (pair-spmf
(spmof-of-set {π. π permutes {.. $n$ }}) (spmof-of-set {π. π permutes {.. $n$ }}));
```

```
  a ← sequence-spmf (map share-nat (permute-list πa xs));
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
  c ← sequence-spmf (map share-nat (permute-list (πa ∘ πb ∘ πc) xs));
```

— round 1

```
  let ζa1 = map2 (-) (map (get-party Party1) a) (permute-list πa (map
(get-party Party1) x));
```

— round 2

```
  let ζb1 = (map (get-party Party1) b);
```

— round 3

```
  let b' = map (get-party Party2) b;
  let ζc1 = map2 (-) (map (get-party Party1) c) (permute-list πc (map2
(+) (map (get-party Party1) b) (map (get-party Party2) b)));
```

```
  let msg1 = ((ζa1, ζb1, ζc1), b', πa, πc);
  return-spmf (map (get-party Party1) x, msg1, c)
```

})

unfolding pair-spmf-alt-def by simp

also have break-perms-2:

```
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);
```

```
  ((πa,πb,πc),π) ← map-spmf (λ(πa,πb,πc). ((πa,πb,πc), πa ∘ πb ∘ πc))
(pair-spmf (spmof-of-set {π. π permutes {.. $n$ }}) (pair-spmf (spmof-of-set {π. π
permutes {.. $n$ }}) (spmof-of-set {π. π permutes {.. $n$ }})));
```

```
  a ← sequence-spmf (map share-nat (permute-list πa xs));
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
  c ← sequence-spmf (map share-nat (permute-list π xs));
```

```

— round 1
  let  $\zeta a1 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party1}) a) (\text{permute-list } \pi a (\text{map } (\text{get-party } \text{Party1}) x));$ 

— round 2
  let  $\zeta b1 = (\text{map } (\text{get-party } \text{Party1}) b);$ 

— round 3
  let  $b' = \text{map } (\text{get-party } \text{Party2}) b;$ 
  let  $\zeta c1 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party1}) c) (\text{permute-list } \pi c (\text{map2 } (+) (\text{map } (\text{get-party } \text{Party1}) b) (\text{map } (\text{get-party } \text{Party2}) b)));$ 

  let  $\text{msg1} = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c);$ 
  return-spmf (map (get-party Party1) x, msg1, c)
})
unfolding pair-spmf-alt-def map-spmf-conv-bind-spmf by simp

```

also have *break-perms-3*:

```

... = (do {
  xs  $\leftarrow$  xs;
  let n = length xs;
  x  $\leftarrow$  sequence-spmf (map share-nat xs);

   $\pi \leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {.. $n$ }};
   $\pi a \leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {.. $n$ }};
   $\pi c \leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {.. $n$ }};
  let  $\pi b = \text{inv } \pi a \circ \pi \circ \text{inv } \pi c;$ 

  a  $\leftarrow$  sequence-spmf (map share-nat (permute-list  $\pi a$  xs));
  b  $\leftarrow$  sequence-spmf (map share-nat (permute-list ( $\pi a \circ \pi b$ ) xs));
  c  $\leftarrow$  sequence-spmf (map share-nat (permute-list  $\pi$  xs));

```

```

— round 1
  let  $\zeta a1 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party1}) a) (\text{permute-list } \pi a (\text{map } (\text{get-party } \text{Party1}) x));$ 

— round 2
  let  $\zeta b1 = (\text{map } (\text{get-party } \text{Party1}) b);$ 

— round 3
  let  $b' = \text{map } (\text{get-party } \text{Party2}) b;$ 
  let  $\zeta c1 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party1}) c) (\text{permute-list } \pi c (\text{map2 } (+) (\text{map } (\text{get-party } \text{Party1}) b) (\text{map } (\text{get-party } \text{Party2}) b)));$ 

  let  $\text{msg1} = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c);$ 
  return-spmf (map (get-party Party1) x, msg1, c)
})
apply (unfold random-perm-middle)
apply (unfold map-spmf-conv-bind-spmf pair-spmf-alt-def)

```

```

by simp

also have break-seqs-3:
... = (do {
  xs ← xs;
  let n = length xs;

  x ← sequence-spmf (map share-nat xs);

  π ← spmf-of-set {π. π permutes {.. $n$ }};
  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};
  let πb = inv πa ∘ π ∘ inv πc;

  a1 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  a2 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  let a = map3 (λa b c. make-sharing b c (a - (b + c))) (permute-list πa xs)
a1 a2;
  b1 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  b2 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  let b = map3 (λa b c. make-sharing b c (a - (b + c))) (permute-list (πa ∘
πb) xs) b1 b2;
  c ← sequence-spmf (map share-nat (permute-list π xs));

— round 1
  let ζa1 = map2 (-) (map (get-party Party1) a) (permute-list πa (map
(get-party Party1) x));

— round 2
  let ζb1 = (map (get-party Party1) b);

— round 3
  let b' = map (get-party Party2) b;
  let ζc1 = map2 (-) (map (get-party Party1) c) (permute-list πc (map2
(+) (map (get-party Party1) b) (map (get-party Party2) b)));

  let msg1 = ((ζa1, ζb1, ζc1), b', πa, πc);
  return-spmf (map (get-party Party1) x, msg1, c)
})
apply (unfold sequence-share-nat-calc[of Party1 Party2 Party3, simplified])
apply (simp add: pair-spmf-alt-def Let-def)
done

also have break-seqs-3:
... = (do {
  xs ← xs;
  let n = length xs;

  x ← sequence-spmf (map share-nat xs);

```



```

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

     $a1::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 
     $a2::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 

     $b1::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 
     $b2::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 
     $c \leftarrow \text{sequence-spmf } (\text{map share-nat } (\text{permute-list } \pi \text{ } xs));$ 

— round 1
     $\text{let } \zeta a1 = \text{map2 } (-) \text{ } a1 \text{ (permute-list } \pi a \text{ (map (get-party Party1) } x));$ 

— round 2
     $\text{let } \zeta b1 = b1;$ 

— round 3
     $\text{let } b' = b2;$ 
     $\text{let } \zeta c1 = \text{map2 } (-) \text{ (map (get-party Party1) } c) \text{ (permute-list } \pi c \text{ (map2$ 
(+ )  $b1 \text{ } b2));$ 

     $\text{let } msg1 = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c);$ 
     $\text{return-spmf } (\text{map (get-party Party1) } x, msg1, c)$ 
  })
supply [intro!] =
   $\text{bind-spmf-cong}[OF \text{ refl}]$ 
   $\text{let-cong}[OF \text{ refl}]$ 
   $\text{prod.case-cong}[OF \text{ refl}]$ 
   $\text{bind-spmf-sequence-map-cong}$ 
   $\text{bind-spmf-sequence-replicate-cong}$ 
   $\text{bind-spmf-permutes-cong}$ 
  unfolding Let-def
  apply rule+
  apply (auto simp: map2-ignore1 map2-ignore2 comp-def prod.case-distrib
bind-spmf-const)
  done

also have
   $\dots = (\text{do } \{x \leftarrow x\text{-dist}; y \leftarrow \text{aby3-shuffleF } x; \text{let } xr = \text{map (get-party } r) \text{ } x; \text{let}$ 
 $yr = \text{map (get-party } r) \text{ } y; msg \leftarrow S \text{ } r \text{ } xr \text{ } yr; \text{return-spmf } (xr, msg, y)\})$ 
  unfolding xs
  unfolding aby3-shuffleF-def
  apply (simp add: bind-spmf-const map-spmf-conv-bind-spmf)
  apply (subst lossless-sequence-spmf[unfolded lossless-spmf-def])
  subgoal by simp
  apply simp
  apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat

```

```

-))
apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))
apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))
apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))
apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))

apply (subst ( $\exists$ ) hoist-map-spmf[where s=sequence-spmf (map share-nat -)
and f=map reconstruct])
apply (subst map-sequence-share-nat-reconstruct)
apply (simp add: map-spmf-conv-bind-spmf)
apply (subst Let-def)

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

supply [simp] = finite-permutations

apply rule
apply rule
apply simp
apply rule
apply rule
unfolding S-def S1-def r
apply (simp add: Let-def)
done

finally have ?hoisted = ... .
} note simulate-party1 = this

{ assume r: r = Party2
have project-to-Party2:
  ?hoisted = (do {
    xs  $\leftarrow$  xs;
    let n = length xs;
    x  $\leftarrow$  sequence-spmf (map share-nat xs);

     $\pi a \leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {.. $n$ }};
     $\pi b \leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {.. $n$ }};
     $\pi c \leftarrow$  spmf-of-set { $\pi$ .  $\pi$  permutes {.. $n$ }};

```

```

    a ← sequence-spmf (map share-nat (permute-list  $\pi a$  xs));
    b ← sequence-spmf (map share-nat (permute-list ( $\pi a \circ \pi b$ ) xs));
    c ← sequence-spmf (map share-nat (permute-list ( $\pi a \circ \pi b \circ \pi c$ ) xs));

— round 1
    let x' = map (get-party Party3) x;
    let y' = map (aby3-stack-sharing Party1) x;
    let  $\zeta a2$  = map (case-prod (-)) (zip (map (get-party Party2) a) (map
(get-party Party2) (permute-list  $\pi a$  y')));

— round 2
    let a' = map (get-party Party1) a;
    let y' = map (aby3-stack-sharing Party2) a;
    let  $\zeta b2$  = map (case-prod (-)) (zip (map (get-party Party2) b) (map
(get-party Party2) (permute-list  $\pi b$  y')));

— round 3
    let b' = map (get-party Party2) b;
    let y' = map (aby3-stack-sharing Party3) b;
    let  $\zeta c2$  = map (case-prod (-)) (zip (map (get-party Party2) c) (map
(get-party Party2) (permute-list  $\pi c$  y')));

    let msg2 = (( $\zeta a2$ ,  $\zeta b2$ ,  $\zeta c2$ ), x',  $\pi a$ ,  $\pi b$ );
    return-spmf (map (get-party Party2) x, msg2, c)
  })
  by (simp add: r Let-def get-party-map-sharing2 map-map-prod-zip')

also have project-to-Party2:
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

   $\pi a$  ← spmf-of-set { $\pi$ .  $\pi$  permutes  $\{..<n\}$ };
   $\pi b$  ← spmf-of-set { $\pi$ .  $\pi$  permutes  $\{..<n\}$ };
   $\pi c$  ← spmf-of-set { $\pi$ .  $\pi$  permutes  $\{..<n\}$ };

  a ← sequence-spmf (map share-nat (permute-list  $\pi a$  xs));
  b ← sequence-spmf (map share-nat (permute-list ( $\pi a \circ \pi b$ ) xs));
  c ← sequence-spmf (map share-nat (permute-list ( $\pi a \circ \pi b \circ \pi c$ ) xs));

— round 1
    let x' = map (get-party Party3) x;
    let y' = map (aby3-stack-sharing Party1) x;
    let  $\zeta a2$  = map2 (-) (map (get-party Party2) a) (permute-list  $\pi a$  (map
(get-party Party2) y')));

— round 2
    let a' = map (get-party Party1) a;

```

$let\ y' = map\ (aby3\text{-stack-sharing}\ Party2)\ a;$
 $let\ \zeta b2 = map2\ (-)\ (map\ (get\text{-party}\ Party2)\ b)\ (permute\text{-list}\ \pi b\ (map\ (get\text{-party}\ Party2)\ y'));$

— round 3

$let\ b' = map\ (get\text{-party}\ Party2)\ b;$
 $let\ y' = map\ (aby3\text{-stack-sharing}\ Party3)\ b;$
 $let\ \zeta c2 = map2\ (-)\ (map\ (get\text{-party}\ Party2)\ c)\ (permute\text{-list}\ \pi c\ (map\ (get\text{-party}\ Party2)\ y'));$

$let\ msg2 = ((\zeta a2, \zeta b2, \zeta c2), x', \pi a, \pi b);$
 $return\text{-spmf}\ (map\ (get\text{-party}\ Party2)\ x, msg2, c)$
 $\})$

supply [intro!] =
 $bind\text{-spmf}\text{-cong}[OF\ refl]$
 $let\text{-cong}[OF\ refl]$
 $prod.\text{case}\text{-cong}[OF\ refl]$
 $bind\text{-spmf}\text{-sequence}\text{-map}\text{-cong}$
 $bind\text{-spmf}\text{-sequence}\text{-replicate}\text{-cong}$
 $bind\text{-spmf}\text{-permutes}\text{-cong}$

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

..

also have reduce-Lets:

$\dots = (do\ \{$
 $xs \leftarrow xs;$
 $let\ n = length\ xs;$
 $x \leftarrow sequence\text{-spmf}\ (map\ share\text{-nat}\ xs);$

$\pi a \leftarrow spmf\text{-of}\text{-set}\ \{\pi.\ \pi\ \text{permutes}\ \{..\lt n\}\};$
 $\pi b \leftarrow spmf\text{-of}\text{-set}\ \{\pi.\ \pi\ \text{permutes}\ \{..\lt n\}\};$
 $\pi c \leftarrow spmf\text{-of}\text{-set}\ \{\pi.\ \pi\ \text{permutes}\ \{..\lt n\}\};$

$a \leftarrow sequence\text{-spmf}\ (map\ share\text{-nat}\ (permute\text{-list}\ \pi a\ xs));$
 $b \leftarrow sequence\text{-spmf}\ (map\ share\text{-nat}\ (permute\text{-list}\ (\pi a \circ \pi b)\ xs));$
 $c \leftarrow sequence\text{-spmf}\ (map\ share\text{-nat}\ (permute\text{-list}\ (\pi a \circ \pi b \circ \pi c)\ xs));$

— round 1

```

    let x' = map (get-party Party3) x;
    let ζa2 = map2 (-) (map (get-party Party2) a) (permute-list πa (map2
(+) (map (get-party Party2) x) (map (get-party Party3) x)));

— round 2
    let ζb2 = map2 (-) (map (get-party Party2) b) (permute-list πb (map
(get-party Party2) a));

— round 3
    let ζc2 = map2 (-) (map (get-party Party2) c) (permute-list πc (replicate
(length b) 0));

    let msg2 = ((ζa2, ζb2, ζc2), x', πa, πb);

    return-spmf (map (get-party Party2) x, msg2, c)
  })
unfolding Let-def
unfolding aby3-stack-sharing-def
by (simp add: comp-def make-sharing'-sel map-replicate-const zip-map-map-same[symmetric])

also have simplify-minus-zero:
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  πa ← spmf-of-set {π. π permutes {..

```

```

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

apply rule+
apply (subst permute-list-replicate)
subgoal by simp

apply (subst map2-minus-zero)
subgoal by simp
subgoal by simp

..

also have break-perms-1:
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  (πa,πb,πc) ← pair-spmf (spmof-of-set {π. π permutes {..unfolding pair-spmf-alt-def by simp

also have break-perms-2:

```

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  ((πa,πb,πc),π) ← map-spmf (λ(πa,πb,πc). ((πa,πb,πc), πa ∘ πb ∘ πc))
(pair-spmf (spmj-of-set {π. π permutes {..<n}}) (pair-spmf (spmj-of-set {π. π
permutes {..<n}}) (spmj-of-set {π. π permutes {..<n}})))));

  a ← sequence-spmf (map share-nat (permute-list πa xs));
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
  c ← sequence-spmf (map share-nat (permute-list π xs));

— round 1
  let x' = map (get-party Party3) x;
  let ζa2 = map2 (-) (map (get-party Party2) a) (permute-list πa (map2
(+)) (map (get-party Party2) x) (map (get-party Party3) x)));

— round 2
  let ζb2 = map2 (-) (map (get-party Party2) b) (permute-list πb (map
(get-party Party2) a));

— round 3
  let ζc2 = (map (get-party Party2) c);

  let msg2 = ((ζa2, ζb2, ζc2), x', πa, πb);
  return-spmf (map (get-party Party2) x, msg2, c)
})
unfolding pair-spmf-alt-def map-spmf-conv-bind-spmf by simp

```

also have break-perms-3:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  π ← spmj-of-set {π. π permutes {..<n}};
  πa ← spmj-of-set {π. π permutes {..<n}};
  πb ← spmj-of-set {π. π permutes {..<n}};
  let πc = inv πb ∘ inv πa ∘ π;

  a ← sequence-spmf (map share-nat (permute-list πa xs));
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
  c ← sequence-spmf (map share-nat (permute-list π xs));

— round 1
  let x' = map (get-party Party3) x;
  let ζa2 = map2 (-) (map (get-party Party2) a) (permute-list πa (map2
(+)) (map (get-party Party2) x) (map (get-party Party3) x)));

```

— round 2
 $\text{let } \zeta b2 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party2}) b) (\text{permute-list } \pi b (\text{map } (\text{get-party } \text{Party2}) a));$

— round 3
 $\text{let } \zeta c2 = (\text{map } (\text{get-party } \text{Party2}) c);$

$\text{let } \text{msg2} = ((\zeta a2, \zeta b2, \zeta c2), x', \pi a, \pi b);$
 $\text{return-spmf } (\text{map } (\text{get-party } \text{Party2}) x, \text{msg2}, c)$
 $\}})$
apply (*unfold random-perm-right*)
apply (*unfold map-spmf-conv-bind-spmf pair-spmf-alt-def*)
by *simp*

also have *break-seqs-3*:
 $\dots = (\text{do } \{$
 $\text{xs} \leftarrow \text{xs};$
 $\text{let } n = \text{length } \text{xs};$

 $x2 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spm-f-of-set } \text{UNIV}));$
 $x3 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spm-f-of-set } \text{UNIV}));$
 $\text{let } x = \text{map3 } (\lambda a b c. \text{make-sharing}' \text{Party2 Party3 Party1 } b c (a - (b +$
 $c))) \text{xs } x2 \text{ } x3;$

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

 $a2 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spm-f-of-set } \text{UNIV}));$
 $a3 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spm-f-of-set } \text{UNIV}));$
 $\text{let } a = \text{map3 } (\lambda a b c. \text{make-sharing}' \text{Party2 Party3 Party1 } b c (a - (b +$
 $c))) (\text{permute-list } \pi a \text{xs}) a2 a3;$
 $b2 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spm-f-of-set } \text{UNIV}));$
 $b3 \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spm-f-of-set } \text{UNIV}));$
 $\text{let } b = \text{map3 } (\lambda a b c. \text{make-sharing}' \text{Party2 Party3 Party1 } b c (a - (b +$
 $c))) (\text{permute-list } (\pi a \circ \pi b) \text{xs}) b2 b3;$
 $c \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } \pi \text{xs}));$

— round 1
 $\text{let } x' = \text{map } (\text{get-party } \text{Party3}) x;$
 $\text{let } \zeta a2 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party2}) a) (\text{permute-list } \pi a (\text{map2 } (+) (\text{map } (\text{get-party } \text{Party2}) x) (\text{map } (\text{get-party } \text{Party3}) x)));$

— round 2
 $\text{let } \zeta b2 = \text{map2 } (-) (\text{map } (\text{get-party } \text{Party2}) b) (\text{permute-list } \pi b (\text{map } (\text{get-party } \text{Party2}) a));$


```

— round 3
  let  $\zeta c2 = (\text{map } (\text{get-party } \text{Party2}) c)$ ;

  let  $\text{msg2} = ((\zeta a2, \zeta b2, \zeta c2), x', \pi a, \pi b)$ ;
  return-spmf (map (get-party Party2) x, msg2, c)
})
apply (unfold sequence-share-nat-calc'[of Party2 Party3 Party1, simplified])
apply (simp add: pair-spmf-alt-def Let-def)
done

also have break-seqs-3:
... = (do {
  xs  $\leftarrow$  xs;
  let n = length xs;

  x2  $\leftarrow$  sequence-spmf (replicate n (spm-f-of-set UNIV));
  x3  $\leftarrow$  sequence-spmf (replicate n (spm-f-of-set UNIV));

   $\pi \leftarrow$  spm-f-of-set { $\pi$ .  $\pi$  permutes {.. $n$ } };
   $\pi a \leftarrow$  spm-f-of-set { $\pi$ .  $\pi$  permutes {.. $n$ } };
   $\pi b \leftarrow$  spm-f-of-set { $\pi$ .  $\pi$  permutes {.. $n$ } };

  a2::natL list  $\leftarrow$  sequence-spmf (replicate n (spm-f-of-set UNIV));
  a3::natL list  $\leftarrow$  sequence-spmf (replicate n (spm-f-of-set UNIV));

  b2::natL list  $\leftarrow$  sequence-spmf (replicate n (spm-f-of-set UNIV));
  b3::natL list  $\leftarrow$  sequence-spmf (replicate n (spm-f-of-set UNIV));

  c  $\leftarrow$  sequence-spmf (map share-nat (permute-list  $\pi$  xs));

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

— round 2
  let  $\zeta b2 = \text{map2 } (-) b2 (\text{permute-list } \pi b a2)$ ;

— round 3
  let  $\zeta c2 = (\text{map } (\text{get-party } \text{Party2}) c)$ ;

  let  $\text{msg2} = ((\zeta a2, \zeta b2, \zeta c2), x', \pi a, \pi b)$ ;
  return-spmf (x2, msg2, c)

})
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong

```

```

bind-spmf-sequence-replicate-cong
bind-spmf-permutes-cong
unfolding Let-def
apply rule+
  apply (auto simp: map2-ignore1 map2-ignore2 comp-def prod.case-distrib
bind-spmf-const make-sharing'-sel)
done

```

also have

```

... = (do {x ← x-dist; y ← aby3-shuffleF x; let xr = map (get-party r) x; let
yr = map (get-party r) y; msg ← S r xr yr; return-spmf (xr, msg, y)})
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

```

unfolding *xs*

unfolding *aby3-shuffleF-def*

apply (*simp add: bind-spmf-const map-spmf-conv-bind-spmf*)

apply (*subst lossless-sequence-spmf[unfolded lossless-spmf-def]*)

subgoal by *simp*

apply (*subst lossless-sequence-spmf[unfolded lossless-spmf-def]*)

subgoal by *simp*

apply *simp*

apply *rule*

apply (*subst (2) sequence-share-nat-calc'[of Party2 Party3 Party1, simplified]*)

apply (*subst (2) Let-def*)

apply (*simp add: pair-spmf-alt-def*)

apply (*subst Let-def*)

unfolding *S-def r*

apply (*simp add: pair-spmf-alt-def comp-def prod.case-distrib map2-ignore2*
make-sharing'-sel)

apply (*rule trans[rotated]*)

apply (*rule bind-spmf-sequence-replicate-cong*)

apply (*rule bind-spmf-sequence-replicate-cong*)

apply (*simp add: map2-ignore1 map2-ignore2*)

apply (*subst bind-spmf-const*)

apply (*subst lossless-sequence-spmf[unfolded lossless-spmf-def]*)

subgoal by *simp*

apply *simp*

apply (*subst (2) bind-commute-spmf[where p=sequence-spmf (replicate - -)]*)

apply (*subst bind-commute-spmf[where q=sequence-spmf (map share-nat*

```

-))
  apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))
  apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))
  apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))
  apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-))

```

```

  apply rule
  apply rule
  apply rule

```

```

  unfolding S2-def Let-def
  apply simp
  done

```

```

  finally have ?hoisted = ... .
} note simulate-party2 = this

```

```

{ assume r: r = Party3
  have project-to-Party3:
    ?hoisted = (do {
      xs ← xs;
      let n = length xs;
      x ← sequence-spmf (map share-nat xs);

      πa ← spmf-of-set {π. π permutes {.. $n$ }};
      πb ← spmf-of-set {π. π permutes {.. $n$ }};
      πc ← spmf-of-set {π. π permutes {.. $n$ }};

      a ← sequence-spmf (map share-nat (permute-list πa xs));
      b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
      c ← sequence-spmf (map share-nat (permute-list (πa ∘ πb ∘ πc) xs));

```

— round 1

```

  let x' = map (get-party Party3) x;
  let y' = map (aby3-stack-sharing Party1) x;
  let ζa3 = map (case-prod (-)) (zip (map (get-party Party3) a) (map
(get-party Party3) (permute-list πa y')));

```

— round 2

```

  let a' = map (get-party Party1) a;
  let y' = map (aby3-stack-sharing Party2) a;
  let ζb3 = map (case-prod (-)) (zip (map (get-party Party3) b) (map
(get-party Party3) (permute-list πb y')));

```

— round 3

```

    let b' = map (get-party Party2) b;
    let y' = map (aby3-stack-sharing Party3) b;
    let ζc3 = map (case-prod (-)) (zip (map (get-party Party3) c) (map
(get-party Party3) (permute-list πc y')));

    let msg3 = ((ζa3, ζb3, ζc3), a', πb, πc);
    return-spmf (map (get-party Party3) x, msg3, c)
  })
  by (simp add: r Let-def get-party-map-sharing2 map-map-prod-zip')

```

also have *project-to-Party1*:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};

  a ← sequence-spmf (map share-nat (permute-list πa xs));
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
  c ← sequence-spmf (map share-nat (permute-list (πa ∘ πb ∘ πc) xs));

```

— round 1

```

    let x' = map (get-party Party3) x;
    let y' = map (aby3-stack-sharing Party1) x;
    let ζa3 = map2 (-) (map (get-party Party3) a) (permute-list πa (map
(get-party Party3) y'));

```

— round 2

```

    let a' = map (get-party Party1) a;
    let y' = map (aby3-stack-sharing Party2) a;
    let ζb3 = map2 (-) (map (get-party Party3) b) (permute-list πb (map
(get-party Party3) y'));

```

— round 3

```

    let b' = map (get-party Party2) b;
    let y' = map (aby3-stack-sharing Party3) b;
    let ζc3 = map2 (-) (map (get-party Party3) c) (permute-list πc (map
(get-party Party3) y'));

```

```

    let msg3 = ((ζa3, ζb3, ζc3), a', πb, πc);
    return-spmf (map (get-party Party3) x, msg3, c)

```

```

  })
  supply [intro!] =
    bind-spmf-cong[OF refl]

```

```

let-cong[OF refl]
prod.case-cong[OF refl]
bind-spmf-sequence-map-cong
bind-spmf-sequence-replicate-cong
bind-spmf-permutes-cong

```

```

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

```

```

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

```

```

apply rule+
apply (subst permute-list-map[symmetric])
subgoal by simp

```

```

..

```

also have *reduce-Lets*:

```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  πa ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};

  a ← sequence-spmf (map share-nat (permute-list πa xs));
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));
  c ← sequence-spmf (map share-nat (permute-list (πa ∘ πb ∘ πc) xs));

```

— round 1

```

  let ζa3 = map2 (−) (map (get-party Party3) a) (permute-list πa (replicate
(length x) 0));

```

— round 2

```

  let a' = map (get-party Party1) a;
  let ζb3 = map2 (−) (map (get-party Party3) b) (permute-list πb (map2
(+) (map (get-party Party3) a) (map (get-party Party1) a)));

```

— round 3

```

  let ζc3 = map2 (−) (map (get-party Party3) c) (permute-list πc (map
(get-party Party3) b));

```

```

  let msg3 = ((ζa3, ζb3, ζc3), a', πb, πc);
  return-spmf (map (get-party Party3) x, msg3, c)
})

```

```

unfolding Let-def
unfolding aby3-stack-sharing-def
by (simp add: comp-def make-sharing'-sel map-replicate-const zip-map-map-same[symmetric])

also have simplify-minus-zero:
... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  π a ← spmf-of-set {π. π permutes {..<n}};
  π b ← spmf-of-set {π. π permutes {..<n}};
  π c ← spmf-of-set {π. π permutes {..<n}};

  a ← sequence-spmf (map share-nat (permute-list π a xs));
  b ← sequence-spmf (map share-nat (permute-list (π a ∘ π b) xs));
  c ← sequence-spmf (map share-nat (permute-list (π a ∘ π b ∘ π c) xs));

— round 1
  let ζ a3 = (map (get-party Party3) a);

— round 2
  let a' = map (get-party Party1) a;
  let ζ b3 = map2 (-) (map (get-party Party3) b) (permute-list π b (map2
(+) (map (get-party Party3) a) (map (get-party Party1) a))));

— round 3
  let ζ c3 = map2 (-) (map (get-party Party3) c) (permute-list π c (map
(get-party Party3) b));

  let msg3 = ((ζ a3, ζ b3, ζ c3), a', π b, π c);
  return-spmf (map (get-party Party3) x, msg3, c)
})
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

apply rule+
apply (subst permute-list-replicate)
subgoal by simp

apply (subst map2-minus-zero)
subgoal by simp
subgoal by simp

```

..

also have *break-perms-1*:

```
... = (do {  
  xs ← xs;  
  let n = length xs;  
  x ← sequence-spmf (map share-nat xs);
```

```
  (πa,πb,πc) ← pair-spmf (spmof-of-set {π. π permutes {..<n}}) (pair-spmf  
(spmof-of-set {π. π permutes {..<n}}) (spmof-of-set {π. π permutes {..<n}}));
```

```
  a ← sequence-spmf (map share-nat (permute-list πa xs));  
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));  
  c ← sequence-spmf (map share-nat (permute-list (πa ∘ πb ∘ πc) xs));
```

— round 1

```
  let ζa3 = (map (get-party Party3) a);
```

— round 2

```
  let a' = map (get-party Party1) a;  
  let ζb3 = map2 (-) (map (get-party Party3) b) (permute-list πb (map2  
(+) (map (get-party Party3) a) (map (get-party Party1) a)));
```

— round 3

```
  let ζc3 = map2 (-) (map (get-party Party3) c) (permute-list πc (map  
(get-party Party3) b));
```

```
  let msg3 = ((ζa3, ζb3, ζc3), a', πb, πc);  
  return-spmf (map (get-party Party3) x, msg3, c)  
})
```

unfolding *pair-spmf-alt-def* **by** *simp*

also have *break-perms-2*:

```
... = (do {  
  xs ← xs;  
  let n = length xs;  
  x ← sequence-spmf (map share-nat xs);
```

```
  ((πa,πb,πc),π) ← map-spmf (λ(πa,πb,πc). ((πa,πb,πc), πa ∘ πb ∘ πc))  
(pair-spmf (spmof-of-set {π. π permutes {..<n}}) (pair-spmf (spmof-of-set {π. π  
permutes {..<n}}) (spmof-of-set {π. π permutes {..<n}})));
```

```
  a ← sequence-spmf (map share-nat (permute-list πa xs));  
  b ← sequence-spmf (map share-nat (permute-list (πa ∘ πb) xs));  
  c ← sequence-spmf (map share-nat (permute-list π xs));
```

— round 1

```
  let ζa3 = (map (get-party Party3) a);
```

— round 2
 $\text{let } a' = \text{map } (\text{get-party Party1}) a;$
 $\text{let } \zeta b3 = \text{map2 } (-) (\text{map } (\text{get-party Party3}) b) (\text{permute-list } \pi b (\text{map2}$
 $(+) (\text{map } (\text{get-party Party3}) a) (\text{map } (\text{get-party Party1}) a));$

— round 3
 $\text{let } \zeta c3 = \text{map2 } (-) (\text{map } (\text{get-party Party3}) c) (\text{permute-list } \pi c (\text{map}$
 $(\text{get-party Party3}) b));$

$\text{let } \text{msg3} = ((\zeta a3, \zeta b3, \zeta c3), a', \pi b, \pi c);$
 $\text{return-spmf } (\text{map } (\text{get-party Party3}) x, \text{msg3}, c)$
 $\})$
unfolding *pair-spmf-alt-def map-spmf-conv-bind-spmf by simp*

also have *break-perms-3*:

$\dots = (\text{do } \{$
 $xs \leftarrow xs;$
 $\text{let } n = \text{length } xs;$
 $x \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } xs);$

$\pi \leftarrow \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$
 $\pi b \leftarrow \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$
 $\pi c \leftarrow \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$
 $\text{let } \pi a = \pi \circ \text{inv } \pi c \circ \text{inv } \pi b;$

$a \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } \pi a \text{ } xs));$
 $b \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } (\pi a \circ \pi b) \text{ } xs));$
 $c \leftarrow \text{sequence-spmf } (\text{map } \text{share-nat } (\text{permute-list } \pi \text{ } xs));$

— round 1
 $\text{let } \zeta a3 = (\text{map } (\text{get-party Party3}) a);$

— round 2
 $\text{let } a' = \text{map } (\text{get-party Party1}) a;$
 $\text{let } \zeta b3 = \text{map2 } (-) (\text{map } (\text{get-party Party3}) b) (\text{permute-list } \pi b (\text{map2}$
 $(+) (\text{map } (\text{get-party Party3}) a) (\text{map } (\text{get-party Party1}) a));$

— round 3
 $\text{let } \zeta c3 = \text{map2 } (-) (\text{map } (\text{get-party Party3}) c) (\text{permute-list } \pi c (\text{map}$
 $(\text{get-party Party3}) b));$

$\text{let } \text{msg3} = ((\zeta a3, \zeta b3, \zeta c3), a', \pi b, \pi c);$
 $\text{return-spmf } (\text{map } (\text{get-party Party3}) x, \text{msg3}, c)$
 $\})$
apply (*unfold random-perm-left*)
apply (*unfold map-spmf-conv-bind-spmf pair-spmf-alt-def*)
by (*simp add: Let-def*)

also have *break-seqs-3*:


```

... = (do {
  xs ← xs;
  let n = length xs;
  x ← sequence-spmf (map share-nat xs);

  π ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};
  let πa = π ∘ inv πc ∘ inv πb;

  a3 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  a1 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  let a = map3 (λa b c. make-sharing' Party3 Party1 Party2 b c (a - (b +
c))) (permute-list πa xs) a3 a1;
  b3 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  b1 ← sequence-spmf (replicate n (spmf-of-set UNIV));
  let b = map3 (λa b c. make-sharing' Party3 Party1 Party2 b c (a - (b +
c))) (permute-list (πa ∘ πb) xs) b3 b1;
  c ← sequence-spmf (map share-nat (permute-list π xs));

— round 1
  let ζa3 = (map (get-party Party3) a);

— round 2
  let a' = map (get-party Party1) a;
  let ζb3 = map2 (-) (map (get-party Party3) b) (permute-list πb (map2
(+)) (map (get-party Party3) a) (map (get-party Party1) a)));

— round 3
  let ζc3 = map2 (-) (map (get-party Party3) c) (permute-list πc (map
(get-party Party3) b));

  let msg3 = ((ζa3, ζb3, ζc3), a', πb, πc);
  return-spmf (map (get-party Party3) x, msg3, c)
})
apply (unfold sequence-share-nat-calc'[of Party3 Party1 Party2, simplified])
apply (simp add: pair-spmf-alt-def Let-def)
done

```

also have *break-seqs-3*:

```

... = (do {
  xs ← xs;
  let n = length xs;

  x ← sequence-spmf (map share-nat xs);

  π ← spmf-of-set {π. π permutes {.. $n$ }};
  πb ← spmf-of-set {π. π permutes {.. $n$ }};
  πc ← spmf-of-set {π. π permutes {.. $n$ }};

```

```

    let  $\pi a = \pi \circ \text{inv } \pi c \circ \text{inv } \pi b$ ;

     $a3::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmof-of-set UNIV)});$ 
     $a1::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmof-of-set UNIV)});$ 

     $b3::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmof-of-set UNIV)});$ 
     $b1::\text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmof-of-set UNIV)});$ 

     $c \leftarrow \text{sequence-spmf } (\text{map share-nat } (\text{permute-list } \pi \text{ } xs));$ 

— round 1
    let  $\zeta a3 = a3$ ;

— round 2
    let  $a' = a1$ ;
    let  $\zeta b3 = \text{map2 } (-) \text{ } b3 \text{ (permute-list } \pi b \text{ (map2 } (+) \text{ } a3 \text{ } a1));$ 

— round 3
    let  $\zeta c3 = \text{map2 } (-) \text{ (map (get-party Party3) } c \text{ (permute-list } \pi c \text{ } b3));$ 

    let  $\text{msg3} = ((\zeta a3, \zeta b3, \zeta c3), a', \pi b, \pi c)$ ;
    return-spmf (map (get-party Party3)  $x$ ,  $\text{msg3}$ ,  $c$ )
  })
supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong
  unfolding Let-def
  apply rule+
  apply (auto simp: map2-ignore1 map2-ignore2 comp-def prod.case-distrib
bind-spmf-const make-sharing'-sel)
  done

also have
  ... = (do { $x \leftarrow x\text{-dist}$ ;  $y \leftarrow \text{aby3-shuffleF } x$ ; let  $xr = \text{map } (\text{get-party } r) \text{ } x$ ; let
 $yr = \text{map } (\text{get-party } r) \text{ } y$ ;  $\text{msg} \leftarrow S \text{ } r \text{ } xr \text{ } yr$ ; return-spmf ( $xr$ ,  $\text{msg}$ ,  $y$ )})
  unfolding  $xs$ 
  unfolding aby3-shuffleF-def
  apply (subst bind-spmf-const)
  apply (subst lossless-sequence-spmf[unfolded lossless-spmf-def])
  subgoal by simp
  apply (simp add: map-spmf-conv-bind-spmf)
  apply (subst bind-commute-spmf[where  $q=\text{sequence-spmf } (\text{map share-nat } -)$ ]])
  apply (subst bind-commute-spmf[where  $q=\text{sequence-spmf } (\text{map share-nat } -)$ ]])

```

```

    apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-)])
    apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-)])
    apply (subst bind-commute-spmf[where q=sequence-spmf (map share-nat
-)])

    apply (subst (3) hoist-map-spmf[where s=sequence-spmf (map share-nat -)
and f=map reconstruct])
    apply (subst map-sequence-share-nat-reconstruct)
    apply (simp add: map-spmf-conv-bind-spmf)
    apply (subst Let-def)

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong

supply [simp] = finite-permutations

    apply rule
    apply rule
    apply simp
    apply rule
    apply rule
    unfolding S-def S3-def r
    apply (simp add: Let-def)
    done

    finally have ?hoisted = ... .
  } note simulate-party3 = this

show ?thesis
  unfolding hoisted-save
  apply (cases r)
  subgoal using simulate-party1 .
  subgoal using simulate-party2 .
  subgoal using simulate-party3 .
  done
qed

lemma Collect-case-prod:
  {f x y | x y. P x y} = (case-prod f) ‘ (Collect (case-prod P))
  by auto

```

lemma *inj-split-Cons'*: *inj-on* ($\lambda(n, xs). n\#xs$) X
by (*auto intro!*: *inj-onI*)

lemma *finite-indicator-eq-sum*:
 $finite\ A \implies indicat-real\ A\ x = sum\ (indicat-real\ \{x\})\ A$
by (*induction rule*: *finite-induct*) (*auto simp*: *indicator-def*)

lemma *spmf-of-set-Cons*:
 $spmf-of-set\ (set-Cons\ A\ B) = map2-spmf\ (\#)\ (spmf-of-set\ A)\ (spmf-of-set\ B)$
unfolding *set-Cons-def pair-spmf-of-set*
apply (*rule spmf-eq-iff-set*)
subgoal unfolding *Collect-case-prod* **apply** (*auto simp*: *set-spmf-of-set*)
apply (*subst (asm) finite-image-iff*)
subgoal by (*rule inj-split-Cons'*)
subgoal by (*auto simp*: *finite-cartesian-product-iff*)
done
subgoal unfolding *Collect-case-prod*
apply (*auto simp*: *spmf-of-set map-spmf-conv-bind-spmf smpf-bind integral-spmf-of-set*)
apply (*subst card-image*)
subgoal by (*rule inj-split-Cons'*)
apply (*auto simp*: *card-eq-0-iff indicator-single-Some*)
apply (*subst (asm) finite-indicator-eq-sum*)
subgoal by (*simp add*: *finite-image-iff inj-split-Cons'*)
apply (*subst (asm) sum.reindex*)
subgoal by (*simp add*: *finite-image-iff inj-split-Cons'*)
apply (*auto*)
done
done

lemma *sequence-spmf-replicate*:
 $sequence-spmf\ (replicate\ n\ (spmf-of-set\ A)) = smpf-of-set\ (listset\ (replicate\ n\ A))$
apply (*induction n*)
subgoal by (*auto simp*: *spmf-of-set-singleton*)
subgoal by (*auto simp*: *spmf-of-set-Cons*)
done

lemma *listset-replicate*:
 $listset\ (replicate\ n\ A) = \{l. length\ l = n \wedge set\ l \subseteq A\}$
apply (*induction n*)
apply (*auto simp*: *set-Cons-def*)
subgoal for $n\ x$
by (*cases x*; *simp*)
done

lemma *map2-map2-map3*:
 $map2\ f\ (map2\ g\ x\ y)\ z = map3\ (\lambda x\ y. f\ (g\ x\ y))\ x\ y\ z$
by (*auto simp*: *zip-assoc map-zip-map*)

lemma *inv-add-sequence*:

```

assumes  $n = \text{length } x$ 
shows
 $\text{map-spmf } (\lambda \zeta :: \text{natL list. } (\zeta, \text{map2 } (+) \zeta x)) (\text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } \text{UNIV})))$ 
=
 $\text{map-spmf } (\lambda y. (\text{map2 } (-) y x, y)) (\text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } \text{UNIV})))$ 
unfolding sequence-spmf-replicate
apply (subst map-spmf-of-set-inj-on)
subgoal unfolding inj-on-def by simp
apply (subst map-spmf-of-set-inj-on)
subgoal unfolding inj-on-def by simp
apply (rule arg-cong[where f=spmof-of-set])
using assms apply (auto simp: image-def listset-replicate map2-map2-map3 zip-same-conv-map map-zip-map2 map2-ignore2)
done

```

lemma *S1-def-simplified*:

```

 $S1 \ x1 \ yc1 = (\text{do } \{$ 
   $\text{let } n = \text{length } x1;$ 

   $\pi a \leftarrow \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$ 
   $\pi c \leftarrow \text{spmof-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$ 
   $\zeta a1 :: \text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } \text{UNIV}));$ 
   $yb1 :: \text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } \text{UNIV}));$ 
   $yb2 :: \text{natL list} \leftarrow \text{sequence-spmf } (\text{replicate } n (\text{spmof-of-set } \text{UNIV}));$ 

   $\text{let } \zeta c1 = \text{map2 } (-) (yc1) (\text{permute-list } \pi c (\text{map2 } (+) yb1 yb2));$ 
   $\text{return-spmf } ((\zeta a1, yb1, \zeta c1), yb2, \pi a, \pi c)$ 
 $\}$ 

```

unfolding *S1-def*

```

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong
  bind-spmf-sequence-map-share-nat-cong

```

apply *rule*

apply *rule*

apply *rule*

apply (*subst hoist-map-spmf'[where s=sequence-spmf - and f= $\lambda x. \text{map2 } (-) x$ -]*)

apply (*subst inv-add-sequence[symmetric]*)

subgoal *by simp*

unfolding *map-spmf-conv-bind-spmf*
apply *simp*
done

lemma *S2-def-simplified:*

```

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

  x3 ← sequence-spmf (replicate n (spmof-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 (spmof-of-set UNIV));
  ζb2::natL list ← sequence-spmf (replicate n (spmof-of-set UNIV));

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

```

unfolding *S2-def*

```

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong
  bind-spmf-sequence-map-share-nat-cong

```

apply *rule*
apply *rule*
apply *rule*
apply *rule*
apply (*rule trans*)
apply (*rule bind-spmf-sequence-replicate-cong*)

```

apply (subst hoist-map-spmf'[where s=sequence-spmf - and f=λx. map2 (-) x
-])
apply (subst inv-add-sequence[symmetric])
subgoal by simp
  apply (rule refl)
apply (unfold map-spmf-conv-bind-spmf)
apply simp
apply (subst hoist-map-spmf'[where s=sequence-spmf - and f=λx. map2 (-) x
-])
apply (subst inv-add-sequence[symmetric])
subgoal by simp
apply (unfold map-spmf-conv-bind-spmf)
apply simp
done

```

lemma *S3-def-simplified*:

```

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

   $\pi b \leftarrow \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$ 
   $\pi c \leftarrow \text{spmf-of-set } \{\pi. \pi \text{ permutes } \{..<n\}\};$ 
  ya3::natL list  $\leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 
  ya1::natL list  $\leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 
  zb3::natL list  $\leftarrow \text{sequence-spmf } (\text{replicate } n \text{ (spmf-of-set UNIV)});$ 

  let  $\zeta c3 = \text{map2 } (-) \text{ } yc3 \text{ (permute-list } \pi c \text{ (map2 } (+) \zeta b3 \text{ (permute-list } \pi b$ 
  (map2 (+) ya3 ya1)));
  return-spmf ((ya3, zb3,  $\zeta c3$ ), ya1,  $\pi b$ ,  $\pi c$ )
})

```

unfolding *S3-def*

```

supply [intro!] =
  bind-spmf-cong[OF refl]
  let-cong[OF refl]
  prod.case-cong[OF refl]
  bind-spmf-sequence-map-cong
  bind-spmf-sequence-replicate-cong
  bind-spmf-permutes-cong
  bind-spmf-sequence-map-share-nat-cong

apply rule
apply rule
apply rule
apply rule
apply rule
apply (subst hoist-map-spmf' [where s=sequence-spmf - and f= $\lambda x. \text{map2 } (-) x$ 
-])
apply (subst inv-add-sequence[symmetric])
subgoal by simp
apply (unfold map-spmf-conv-bind-spmf)
apply simp
done

```

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 $do\text{-}mul :: (natL \times natL) \Rightarrow (natL \times natL) \Rightarrow natL$ **where**
 $do\text{-}mul\ xy1\ xy2 = fst\ xy1 * snd\ xy1 + fst\ xy1 * snd\ xy2 + fst\ xy2 * snd\ xy1$

type-synonym $mul\text{-}in = natL \times natL$

type-synonym $mul\text{-}msg = (natL \times natL) \times natL$

type-synonym $mul\text{-}view = mul\text{-}in \times mul\text{-}msg$

type-synonym $mul\text{-}out = natL$

definition $aby3\text{-}mulR :: mul\text{-}in\ sharing \Rightarrow (mul\text{-}msg\ sharing \times mul\text{-}out\ sharing)$
 $spmf$ **where**

$aby3\text{-}mulR\ xy = (do\ \{$
 $\quad let\ xy\text{-}shift = shift\text{-}sharing\ xy;$
 $\quad let\ z\text{-}raw = map\text{-}sharing2\ do\text{-}mul\ xy\ xy\text{-}shift;$
 $\quad \zeta \leftarrow zero\text{-}sharing;$
 $\quad let\ z = map\text{-}sharing2\ (+)\ z\text{-}raw\ \zeta;$
 $\quad let\ msg = prod\text{-}sharing\ xy\text{-}shift\ \zeta;$
 $\quad return\text{-}spmf\ (msg,\ z)$
 $\})$

definition $aby3\text{-}mulF :: mul\text{-}in\ sharing \Rightarrow mul\text{-}out\ sharing\ spmf$ **where**

$aby3\text{-}mulF\ xy = ($
 $\quad let\ x = reconstruct\ (map\text{-}sharing\ fst\ xy);$
 $\quad \quad y = reconstruct\ (map\text{-}sharing\ snd\ xy)$
 $\quad in\ share\text{-}nat\ (x * y))$

definition $S :: mul\text{-}in \Rightarrow mul\text{-}out \Rightarrow mul\text{-}msg\ spmf$ **where**

$S\ inp\ outp = (do\ \{$
 $\quad let\ (x1,\ y1) = inp;$
 $\quad (x2,\ y2) \leftarrow spmf\text{-}of\text{-}set\ UNIV;$
 $\quad let\ \zeta = outp - do\text{-}mul\ (x1,\ y1)\ (x2,\ y2);$
 $\quad return\text{-}spmf\ ((x2,\ y2),\ \zeta)$
 $\})$

lemma $reconstruct\text{-}do\text{-}mul$:

$reconstruct\ (map\text{-}sharing2\ do\text{-}mul\ xys\ (shift\text{-}sharing\ xys)) = reconstruct\ (map\text{-}sharing\ fst\ xys) * reconstruct\ (map\text{-}sharing\ snd\ xys)$

supply $[simp] = shift\text{-}sharing\text{-}make\text{-}sharing\ do\text{-}mul\text{-}def$

by $(cases\ xys)\ (auto\ simp:\ algebra\text{-}simps)$

theorem $mul\text{-}spec$:

fixes $x\text{-}dist\ y\text{-}dist :: natL\ sharing\ spmf$

assumes $is\text{-}uniform\text{-}sharing\ x\text{-}dist\ is\text{-}uniform\text{-}sharing\ y\text{-}dist$

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 –
  obtain xd yd where xd: x-dist = bind-spmf xd share-nat and yd: y-dist =
  bind-spmf yd share-nat
  using assms unfolding is-uniform-sharing-def by auto
  show ?secure
  unfolding aby3-mulR-def xd yd
  apply (subst (3) hoist-map-spmf[where f=map-sharing2 (+) -])
  apply (subst inv-zero-sharing)
  apply (unfold bind-map-spmf comp-def prod.case)
  unfolding Let-def
  apply (subst reconstruct-do-mul)
  unfolding bind-spmf-assoc
  apply (unfold prod-sharing-map-sel)
  apply (subst hoist-map-spmf[where f=reconstruct])
  apply (subst hoist-map-spmf[where f=reconstruct])
  apply (unfold reconstruct-share-nat)
  apply (unfold bind-map-spmf comp-def prod.case)
  unfolding return-bind-spmf prod.case
  unfolding prod-sharing-sel get-prev-sharing map-sharing-sel prod.case
  apply (subst (1 2) share-nat-def-calc'[of prev-role p p next-role p, unfolded
  case-prod-unfold, simplified])
  unfolding bind-spmf-assoc return-bind-spmf
  unfolding make-sharing-sel-p2

```

```

apply (subst (2 3) share-nat-def-calc'[of prev-role p p next-role p, simplified,
abs-def])
unfolding case-prod-unfold Let-def
unfolding aby3-mulF-def S-def
apply (clarsimp simp add: bind-commute-spmf[where q=yd] bind-commute-spmf[where
q=share-nat -])
supply [simp del] = UNIV-Times-UNIV
apply (fold UNIV-Times-UNIV )
apply (fold pair-spmf-of-set)
apply (unfold pair-spmf-alt-def)
apply clarsimp
unfolding make-sharing-sel-p2
apply (subst bind-spmf-const, simp)
apply (subst bind-spmf-const, simp)
apply (rule arg-cong[where f=bind-spmf -], rule)
apply (rule arg-cong[where f=bind-spmf -], rule)
apply (rule arg-cong[where f=bind-spmf -], rule)
apply (subst (3) bind-commute-spmf)
apply (subst (1) bind-commute-spmf)
apply (subst (2) bind-commute-spmf)
apply simp

done

show ?uniform
unfolding xd yd aby3-mulF-def
unfolding prod-sharing-map-sel
unfolding bind-spmf-assoc
apply (subst (1 2) hoist-map-spmf[where f=reconstruct])
apply (unfold reconstruct-share-nat)
unfolding bind-map-spmf comp-def prod.case Let-def
unfolding bind-commute-spmf[where q=yd]
apply (fold pair-spmf-alt-def[of yd xd, THEN arg-cong[where f= $\lambda x$ . bind-spmf
x (case-prod -)], simplified])
unfolding bind-spmf-const weight-share-nat scale-spmf-1
unfolding case-prod-unfold
apply (fold bind-map-spmf[where f= $\lambda x$ . snd x * fst x, unfolded comp-def])
unfolding is-uniform-sharing-def
by auto
qed

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 $(\text{'val}, \text{'msg}) \text{writer_spmf} = (\text{'val} \times \text{'msg}) \text{spmf}$

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 $\text{Monad-Syntax.bind } \text{bind_writer_spmf}$

definition $\text{flatten_sharingF} :: \text{natL } \text{sharing} \Rightarrow \text{natL } \text{sharing } \text{spmf}$ **where**
 $\text{flatten_sharingF } s = \text{share_nat } (\text{reconstruct } s)$

definition $\text{flatten_sharingR} :: \text{Role} \Rightarrow \text{natL } \text{sharing} \Rightarrow (\text{natL } \text{sharing}, \text{natL}) \text{writer_spmf}$
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 $\text{flatten_sharingS} :: \text{natL} \Rightarrow \text{natL} \Rightarrow \text{natL } \text{spmf}$ **where**
 $\text{flatten_sharingS } \text{inp } \text{outp} = \text{return_spmf } (\text{outp} - \text{inp})$

lemma $\text{flatten_sharing_spec}$:
 $\text{flatten_sharingR } p \ x = \text{do } \{$
 $\quad y \leftarrow \text{flatten_sharingF } x;$
 $\quad \text{msg} \leftarrow \text{flatten_sharingS } (\text{get_party } p \ x) \ (\text{get_party } p \ y);$
 $\quad \text{return_spmf } (y, \text{msg})$
 $\}$
unfolding $\text{flatten_sharingR-def}$
unfolding $\text{flatten_sharingF-def}$
unfolding $\text{flatten_sharingS-def}$
apply simp
apply $(\text{fold } \text{zero_masking_eq_share_nat})$
unfolding $\text{map_spmf_conv_bind_spmf}$
apply simp
done

definition $aby3\text{-mulR}' :: \text{Role} \Rightarrow \text{natL} \Rightarrow \text{natL} \Rightarrow (\text{natL} \text{ sharing}, ((\text{natL} \times \text{natL}) \times (\text{natL} \times \text{natL})) \times \text{natL}) \text{ writer-spmf}$ **where**
 $aby3\text{-mulR}' p x y = \text{do}$ {
 $xs \leftarrow \text{share-nat } x;$
 $ys \leftarrow \text{share-nat } y;$
 $\text{let } xy = \text{prod-sharing } xs \text{ } ys;$
 $\text{let } xy\text{-shift} = \text{shift-sharing } xy;$
 $\text{let } z\text{-raw} = \text{map-sharing2 do-mul } xy \text{ } xy\text{-shift};$
 $(z, \zeta\text{-msg}) \leftarrow \text{flatten-sharingR } p \text{ } z\text{-raw};$
 $\text{return-spmf } (z, (((\text{get-party } p \text{ } xs, \text{get-party } p \text{ } ys), \text{get-party } p \text{ } xy\text{-shift}), \zeta\text{-msg}))$
}

definition $aby3\text{-mulF}' :: \text{natL} \Rightarrow \text{natL} \Rightarrow \text{natL} \text{ sharing spmf}$ **where**
 $aby3\text{-mulF}' x y = \text{share-nat } (x * y)$

definition $aby3\text{-mulS}' :: \text{natL} \Rightarrow (((\text{natL} \times \text{natL}) \times (\text{natL} \times \text{natL})) \times \text{natL}) \text{ spmf}$ **where**
 $aby3\text{-mulS}' z = \text{do}$ {
 $x1 \leftarrow \text{spmf-of-set UNIV};$
 $x2 \leftarrow \text{spmf-of-set UNIV};$
 $y1 \leftarrow \text{spmf-of-set UNIV};$
 $y2 \leftarrow \text{spmf-of-set UNIV};$
 $\text{let } \zeta = z - (x1 * y1 + x1 * y2 + x2 * y1);$
 $\text{return-spmf } (((x1, y1), (x2, y2)), \zeta)$
}

lemma $\text{set-spmf-share-nat}$:
 $\text{set-spmf } (\text{share-nat } x) = \{s. \text{reconstruct } s = x\}$
unfolding share-nat-def
apply simp
unfolding vimage-def
apply simp
done

lemma $\text{reconstruct-share-nat}'$:
 $\text{pred-spmf } (\lambda s. \text{reconstruct } s = x) (\text{share-nat } x)$
unfolding pred-spmf-def
apply auto
unfolding $\text{set-spmf-share-nat}$
apply simp
done

lemma share-nat-cong :
 $x = y \implies (\bigwedge s. \text{reconstruct } s = x \implies f s = g s) \implies \text{bind-spmf } (\text{share-nat } x) f$
 $= \text{bind-spmf } (\text{share-nat } y) g$

```

apply (rule bind-spmf-cong)
subgoal by simp
subgoal unfolding set-spmf-share-nat by simp
done

```

lemma *return-ResSim*:

```

return-spmf (r, s) = bind-spmf (return-spmf s) (λmsg. return-spmf (r, msg))
by simp

```

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)))
unfolding aby3-mulR'-def
unfolding flatten-sharing-spec
unfolding aby3-mulF'-def
unfolding flatten-sharingF-def
apply simp
unfolding reconstruct-do-mul
apply simp
unfolding do-mul-def
apply simp
apply (simp cong: share-nat-cong)
unfolding share-nat-def-calc'[of p prev-role p next-role p x, simplified]
unfolding share-nat-def-calc'[of p prev-role p next-role p y, simplified]
unfolding pair-spmf-alt-def
apply simp
unfolding make-sharing'-sel[of p prev-role p next-role p, simplified]
unfolding bind-commute-spmf[where q=share-nat -]
apply (rule share-nat-cong[OF refl])

apply (subst return-ResSim)
apply (fold bind-spmf-assoc)
apply (rule refl)
done

```

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.