ABY3 Multiplication and Array Shuffling

Shuwei Hu

March 17, 2025

Abstract

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

Contents

theory *Finite-Number-Type*
imports
HOL−Library.Cardinality
begin

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

consts *L* :: *nat*

specification (*L*) *L-gt-1*: *L* > *1*
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* ⇒ *natL* **is** $\lambda x.\ (-x)$ *mod int L*
by *simp*

lift-definition *plus-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** $\lambda x\ y.\ (x + y)$ *mod int L*
by *simp*

lift-definition *minus-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** $\lambda x\ y.\ (x - y)$ *mod int L*
by *simp*

lift-definition *times-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** $\lambda x\ y.\ (x * y)$ *mod 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* ⇒ *natL* ⇒ *natL* **is** *inf*
unfolding *inf-int-def* **by** *auto*

lift-definition *sup-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** *sup*
unfolding *sup-int-def* **by** *auto*

lift-definition *less-eq-natL* :: *natL* ⇒ *natL* ⇒ *bool* **is** *less-eq* **.**

lift-definition *less-natL* :: *natL* ⇒ *natL* ⇒ *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* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *sharing* $\Rightarrow 'b$ *sharing* **where**
map-sharing f x = *f* ∘ *x*

definition *get-party* :: *Role* $\Rightarrow 'a$ *sharing* $\Rightarrow 'a$ **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* ⟷ *False r* = *prev-role r* ⟷ *False*
by (*cases r*; *simp*)+

lemma *next-sharing-neq-prev*[*simp*]:
next-role r = *prev-role r* ⟷ *False prev-role r* = *next-role r* ⟷ *False*
by (*cases r*; *simp*)+

lemma *role-otherE*:
obtains *r* :: *Role* **where** *r0*≠*r r*≠*r1*
by (*cases r0*; *cases r1*) *auto*

lemma *make-sharing-sel-p2*:
shows
get-party (*prev-role r*) (*make-sharing′* (*prev-role r*) *r* (*next-role r*) *x1 x2 x3*) =
x1
get-party r (*make-sharing′* (*prev-role r*) *r* (*next-role r*) *x1 x2 x3*) = *x2*
get-party (*next-role r*) (*make-sharing′* (*prev-role r*) *r* (*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* = *make-sharing x1 x2 x3*
subgoal premises *P*
apply (*rule P*[*of s Party1 s Party2 s Party3*])
apply (*rule ext*)
subgoal for *p*
unfolding *make-sharing′-def* **by** (*cases p*; *simp*)
done
done

lemma *sharing-cases′*:
assumes *p1*≠*p2 p2*≠*p3 p3*≠*p1*
obtains *x1 x2 x3* **where** *s* = *make-sharing′ p1 p2 p3 x1 x2 x3*
subgoal premises *P*
apply (*rule P*[*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*]:
make-sharing (*get-party Party1 s*) (*get-party Party2 s*) (*get-party Party3 s*) = *s*
by (*cases s*) *simp*

lemma *sharing-eqI′*:
⟦*get-party Party1 x* = *get-party Party1 y*;
get-party Party2 x = *get-party Party2 y*;
get-party Party3 x = *get-party Party3 y*⟧

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

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

abbreviation *prod-sharing* :: *'a sharing* ⇒ *'b sharing* ⇒ (*'a* × *'b*) *sharing* **where**
prod-sharing ≡ *corec-prod*

abbreviation *map-sharing2* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a sharing* ⇒ *'b sharing* ⇒ *'c sharing* **where**
map-sharing2 f xs ys ≡ *map-sharing* (*case-prod f*) (*prod-sharing xs ys*)

lemma *prod-sharing-sel*[*simp*]:
get-party r (*prod-sharing x y*) = (*get-party r x*, *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* ⇒ *'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* ∘ (*make-sharing Party3 Party1 Party2*)
unfolding *shift-sharing-def make-sharing'-def comp-def get-party-def*
by (*auto*)

type-synonym *'a repsharing* = (*'a* × *'a*) *sharing*

definition *reshare* :: *'a sharing* ⇒ *'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* ⇒ *'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* ⇒ *'b*) ⇒ *'a repsharing* ⇒ *'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* ⇒ *bool* **where**
valid-repsharing s ⟷ *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 ⟹ *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 ⟹ *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* ⇒ *'a* ⇒ *'a* ⇒ *'a repsharing* **where**

make-repsharing x1 x2 x3 = *reshare* (*make-sharing x1 x2 x3*)

definition *reconstruct* :: *natL sharing* ⇒ *natL* **where**
reconstruct s = *get-party Party1 s* + *get-party Party2 s* + *get-party Party3 s*

definition *reconstruct-rep* :: *natL repsharing* ⇒ *natL* **where**
reconstruct-rep s = *reconstruct* (*derep-sharing s*)

lemma *reconstruct-share*[*simp*]:
reconstruct-rep (*reshare s*) = *reconstruct s*
unfolding *reconstruct-rep-def* **by** *simp*

lemma *recontrust-def'*:
assumes *r1*≠*r2 r2*≠*r3 r3*≠*r1*
shows *reconstruct s* = *get-party r1 s* + *get-party r2 s* + *get-party r3 s*
unfolding *reconstruct-def*
using *assms*
by (*cases r1*; *cases r2*; *cases r3*; *simp*)

lemma *reconstruct-make-sharing'*[*simp*]:
assumes *r1* ≠ *r2 r2* ≠ *r3 r3* ≠ *r1*
shows *reconstruct* (*make-sharing' r1 r2 r3 x1 x2 x3*) = *x1* + *x2* + *x3*
unfolding *reconstruct-def*
unfolding *make-sharing'-def get-party-def*
using *assms*
by (*cases r1*; *cases r2*; *cases r3*; *simp*)

lemma *reconstruct-make-repsharing*[*simp*]:
reconstruct-rep (*make-repsharing x1 x2 x3*) = *x1* + *x2* + *x3*
unfolding *make-repsharing-def*
unfolding *reconstruct-share*
by *simp*

definition *valid-nat-repsharing* :: *natL* ⇒ *natL repsharing* ⇒ *bool* **where**
valid-nat-repsharing v s ⟷ *reconstruct-rep s* = *v* ∧ *reshare* (*derep-sharing s*) = *s*

lemma *comp-inj-on-iff'*:
inj-on (*f'* ∘ *f*) *A* ⟷ *inj-on f A* ∧ *inj-on f'* (*f* ‘ *A*)
using *comp-inj-on-iff inj-on-imageI inj-on-imageI2* **by** *auto*

lemma *corec-prod-inject*[*simp*]:
corec-prod f g = *corec-prod f' g'* ⟷ *f* = *f'* ∧ *g* = *g'*
unfolding *corec-prod-def*
by (*meson ext prod.inject*)

lemma *inj-on-corec-prodI*:
inj-on f A ∨ *inj-on g A* ⟹ *inj-on* (λ*x*. *corec-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 (*λx. make-repsharing* (*f x*) (*g x*) (*h x*)) *A* ⟷ *inj-on* (*λx. 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≠r2 r2≠r3 r3≠r1*
shows *make-sharing' r1 r2 r3 x1 x2 x3 = make-sharing' r1 r2 r3 y1 y2 y3* ⟷ *x1=y1* ∧ *x2=y2* ∧ *x3=y3*
using *assms* **by** (*metis make-sharing'-sel*)

lemma *make-sharing-inject*[*simp*]:
make-sharing x1 x2 x3 = make-sharing y1 y2 y3 ⟷ *x1=y1* ∧ *x2=y2* ∧ *x3=y3*
by *simp*

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

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

lemma *inj-on-make-sharingI*:
inj-on f A ∨ *inj-on g A* ∨ *inj-on h A* ⟹ *inj-on* (*λx. 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 ∨ *inj-on g A* ∨ *inj-on h A* ⟹ *inj-on* (*λx. 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*: *finite′ A* ⟷ *card A* ≠ *0*
  and *card-0-finite′*: *card A* = *0* ⟷ ¬*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*: ⋀*a*. *a*∈*A* ⟹ *card* (*f a*) = *c*
  shows *spmf-of-set* (*A* ≫= *f*) = *spmf-of-set A* ≫= (λ*x*. *spmf-of-set* (*f x*))

proof −
  have *card-bind*: *card* (*A* ≫= *f*) = *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* ≠ *0*
      then have *finite* (*f a*) **if** *a*∈*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 spmf-eqI*)
    apply (*unfold spmf-bind*)
    apply (*unfold integral-spmf-of-set*)
    apply (*unfold spmf-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* ‘ *A*
  by *blast*

lemma *ap-set-Union*:
  ap-set F A = (⋃*f*∈*F*. *f* ‘ *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*=λ*x*. *map-pmf x sx*])
  apply (*intro ext*)
  apply (*rule arg-cong*[**where** *f*=*Option.bind* -])
  apply *auto*
  done

definition *rel-scale-spmf s t* ⟷ (*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 ⟹ *rel-scale-spmf t s*
  unfolding *rel-scale-spmf-def* **by** *simp*

lemma *rel-scale-spmf-trans*:
  rel-scale-spmf s t ⟹ *rel-scale-spmf t u* ⟹ *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 sympI transpI dest*: *rel-scale-spmf-trans*)

lemma *spmf-eq-iff*: *p* = *q* ⟷ (∀ *i*. *spmf p i* = *spmf q i*)
  using *spmf-eqI* **by** *auto*

lemma *spmf-eq-iff-set*:
  set-spmf a = *set-spmf b* ⟹ (⋀*x*. *x* ∈ *set-spmf b* ⟹ *spmf a x* = *spmf b x*) ⟹
a = *b*

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

lemma *rel-scale-spmf-None*:
rel-scale-spmf s t $\Longrightarrow$ *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* = *scale-spmf k t*)
proof
assume *rel*: *rel-scale-spmf s t*
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*
then show $\exists$ *k*>*0*. *s* = *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* = *scale-spmf k t*
apply (*unfold mk-lossless-back-eq*)
using *notNone divide-pos-pos* **by** *blast*
qed
next
assume $\exists$ *k*>*0*. *s* = *scale-spmf k t*
then obtain *k* **where** *kpos*: *k*>*0* **and** *st*: *s* = *scale-spmf k t* **by** *blast*
then consider (*isNone*) *weight-spmf s* = *0* $\wedge$ *weight-spmf t* = *0* | (*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 ⟷
(*s* = *return-pmf None* ∧ *t* = *return-pmf None*)
| (*weight-spmf s* > *0* ∧ *weight-spmf t* > *0* ∧ *s* = *scale-spmf* (*weight-spmf s* / *weight-spmf t*) *t*)
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *rel*: *?lhs*
then consider (*isNone*) *s* = *return-pmf None* ∧ *t* = *return-pmf None* | (*notNone*) *weight-spmf s* > *0* ∧ *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*) ≥ (*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* ⟹ *rel-scale-spmf s t* ⟹ *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 $\Longrightarrow$ *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 $\Longrightarrow$ *weight-spmf s* = *weight-spmf t* $\Longrightarrow$ *s* = *t*
unfolding *rel-scale-spmf-def-alt2* **by** *auto*

lemma *rel-scale-spmf-set-restrict*:
finite A $\Longrightarrow$ *rel-scale-spmf* (*restrict-spmf* (*spmf-of-set A*) *B*) (*spmf-of-set* (*A* ∩ *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 ∩ *B* = {} $\Longrightarrow$ *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 $\Longrightarrow$ *restrict-spmf* (*spmf-of-set A*) *B* = *scale-spmf* (*card* (*A*∩*B*) / *card A*) (*spmf-of-set* (*A*∩*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 ⟹ *a≠c* ⟹ *b=c*
unfolding *min-def* **by** *auto*

lemma *weight-0-spmf*:
weight-spmf s = 0 ⟹ *spmf s i = 0*
using *order-trans*[*OF spmf-le-weight*, *of s 0 i*] **by** *simp*

lemma *mk-lossless-scale-absorb*:
r > 0 ⟹ *mk-lossless* (*scale-spmf r s*) = *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*:
scale-spmf k s = return-pmf None ⟷ *k≤0* ∨ *s=return-pmf 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*:
lossless-spmf s ⟹ *spmf-of-pmf* (*map-pmf the s*) = *s*
unfolding *lossless-spmf-conv-spmf-of-pmf* **by** *auto*

lemma *lossless-mk-lossless*:

s ≠ *return-pmf None* ⟹ *lossless-spmf* (*mk-lossless s*)
unfolding *lossless-spmf-def*
unfolding *weight-mk-lossless*
by *simp*

definition *pmf-of-spmf* **where**
pmf-of-spmf p = *map-pmf the* (*mk-lossless p*)

lemma *scale-weight-spmf-of-pmf*:
p = *scale-spmf* (*weight-spmf p*) (*spmf-of-pmf* (*pmf-of-spmf p*))
unfolding *pmf-of-spmf-def*
apply (*cases p* = *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*:
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*:
lossless-spmf p ⟹ *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* ⟹ *pmf-of-spmf* (*scale-spmf r p*) = *pmf-of-spmf p*
unfolding *pmf-of-spmf-def*
by (*simp add*: *mk-lossless-scale-absorb*)

lemma *nonempty-spmf-weight*:
p ≠ *return-pmf None* ⟷ *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*:
pmf-of-spmf (*mk-lossless p*) = *pmf-of-spmf p*
apply (*cases p* = *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 ≠ *return-pmf None* ⟹ *spmf-of-pmf* (*pmf-of-spmf p*) = *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*:
rel-scale-spmf s (*cond-spmf s UNIV*)
unfolding *cond-spmf-UNIV*
by (*rule rel-scale-spmf-mk-lossless*) (*rule rel-scale-spmf-refl*)

lemma *set-pmf p* ∩ *g* ≠ {} ⟹ *pmf-prob p* (*f* ∩ *g*) = *pmf-prob* (*cond-pmf p g*) *f* ∗ *pmf-prob p g*
using *measure-cond-pmf*
unfolding *pmf-prob-def*
by (*metis Int-commute divide-eq-eq measure-measure-pmf-not-zero*)

lemma *bayes*:
set-pmf p ∩ *A* ≠ {} ⟹ *set-pmf p* ∩ *B* ≠ {} ⟹
measure-pmf.prob (*cond-pmf p A*) *B*
= *measure-pmf.prob* (*cond-pmf p B*) *A* ∗ *measure-pmf.prob p B* / *measure-pmf.prob p A*
unfolding *measure-cond-pmf*
by (*subst inf-commute*) (*simp add*: *measure-pmf-zero-iff*)

definition *spmf-prob* :: ′*a spmf* ⇒ ′*a set* ⇒ *real* **where**
spmf-prob p = *Sigma-Algebra.measure* (*measure-spmf p*)

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

lemma *spmf-prob-pmf*:
spmf-prob p A = *pmf-prob p* (*Some* ‘ *A*)
unfolding *spmf-prob-def pmf-prob-def*
unfolding *measure-measure-spmf-conv-measure-pmf*
..

lemma *bayes-spmf*:
spmf-prob (*cond-spmf p A*) *B*
= *spmf-prob* (*cond-spmf p B*) *A* ∗ *spmf-prob p B* / *spmf-prob p A*
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-prob p A = *weight-spmf p* ∗ *pmf-prob* (*pmf-of-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-spmf (*cond-spmf p A*) *B* = *cond-spmf p* (*A* ∩ *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*]] *measure-le-0-iff*
by *metis*

lemma *cond-spmf-prob*:
spmf-prob p (*A* ∩ *B*) = *spmf-prob* (*cond-spmf p A*) *B* ∗ *spmf-prob p A*
unfolding *spmf-prob-def measure-cond-spmf*
using *Int-lower1*[*THEN measure-spmf.finite-measure-mono*[*simplified*]] *measure-le-0-iff*
by (*metis mult-eq-0-iff nonzero-eq-divide-eq*)

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

lemma *spmf-prob-empty*:
spmf-prob empty-spmf A = *0*
unfolding *spmf-prob-def empty-spmf-def*
by *simp*

definition *le-spmf* :: ′*a spmf* ⇒ ′*a spmf* ⇒ *bool* **where**
le-spmf p q ⟷ (∃ *k*≤*1*. *p* = *scale-spmf k q*)

definition *lt-spmf* :: ′*a spmf* ⇒ ′*a spmf* ⇒ *bool* **where**
lt-spmf p q ⟷ (∃ *k*<*1*. *p* = *scale-spmf k q*)

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

lemma *spmf-prob-cond-Int*:
spmf-prob (*cond-spmf p C*) (*A* ∩ *B*)
= *spmf-prob* (*cond-spmf p* (*B* ∩ *C*)) *A* ∗ *spmf-prob* (*cond-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-spmf (*mk-lossless p*) *A* = *cond-spmf p A*
apply (*fold cond-spmf-UNIV*)
apply (*unfold cond-spmf-Int*)
by *simp*

primrec *sequence-spmf* :: *'a spmf list* ⇒ *'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-spmf (*sequence-spmf xs*) = {*l. list-all2* (λ*x s. x* ∈ *set-spmf s*) *l xs*}
by (*induction xs*) (*auto simp*: *list-all2-Cons2*)

lemma *map-spmf-map-sequence*:
map-spmf (*map f*) (*sequence-spmf xs*) = *sequence-spmf* (*map* (*map-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-spmf (*map return-spmf xs*) = *return-spmf xs*
by (*induction xs*) *auto*

lemma *sequence-bind-cong*:
⟦*xs=ys*; ⋀*y. length y* = *length ys* ⟹ *f y* = *g y*⟧ ⟹ *bind-spmf* (*sequence-spmf xs*) *f* = *bind-spmf* (*sequence-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*:
(⋀*l. length l* = *n* ⟹ *f l* = *g l*)
⟹ *bind-spmf* (*sequence-spmf* (*replicate n x*)) *f* = *bind-spmf* (*sequence-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*:
(⋀*l. length l* = *length x* ⟹ *f l* = *g l*)
⟹ *bind-spmf* (*sequence-spmf* (*map m x*)) *f* = *bind-spmf* (*sequence-spmf* (*map*

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* $\Longrightarrow$ *lossless-spmf x*) $\Longrightarrow$ *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*:
  $p \neq q \Longrightarrow$ *map-spmf* (λs. (*get-party p s*, *get-party q s*)) (*share-nat x*) = *spmf-of-set UNIV*
proof −
  assume *pq*: $p \neq q$
  obtain *r* **where** *r*: $q \neq r$ $r \neq p$
    using *role-otherE* **.**
  show *map-spmf* (λs. (*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-uniform-sharing s ⟷ (∃ *x* :: *natL spmf*. *s* = *bind-spmf x share-nat*)

definition *is-uniform-sharing2* :: (*natL sharing* × *natL sharing*) *spmf* ⇒ *bool*
where
is-uniform-sharing2 s ⟷ (∃ *xy* :: (*natL* × *natL*) *spmf*.
s = *bind-spmf xy* (λ(*x*,*y*). *pair-spmf* (*share-nat x*) (*share-nat y*)))

lemma *share-nat-uniform*:
is-uniform-sharing (*share-nat x*)
unfolding *is-uniform-sharing-def*
apply (*rule exI*[**where** *x*=*return-spmf x*])
by *simp*

lemma *share-nat-lossless*:
lossless-spmf (*share-nat x*)
unfolding *share-nat-def*
unfolding *lossless-spmf-of-set*
apply *rule*
subgoal
by (*rule finite-subset*[**where** *B*=*UNIV*]) *auto*
subgoal
unfolding *vimage-def*
apply *simp*
apply (*rule exI*[**where** *x*=*make-sharing x 0 0*])
apply *simp*
done
done

lemma *uniform-sharing2*:
is-uniform-sharing s ⟹ *p1* ≠ *p2* ⟹ *map-spmf* (λ*x*. (*get-party p1 x*, *get-party p2 x*)) *s* = *scale-spmf* (*weight-spmf s*) (*spmf-of-set UNIV*)
unfolding *is-uniform-sharing-def*
apply (*erule exE*)
apply *simp*
unfolding *map-bind-spmf comp-def*
unfolding *share-nat-2party-uniform*
unfolding *bind-spmf-const*
unfolding *weight-bind-spmf*
apply (*subst Bochner-Integration.integral-cong*[**where** *g*=λ-. *1*])
apply (*rule refl*)
subgoal using *share-nat-lossless* **unfolding** *lossless-spmf-def* **by** *simp*
subgoal by *simp*
done

lemma *uniform-sharing*:
is-uniform-sharing s ⟹ *map-spmf* (*get-party p*) *s* = *scale-spmf* (*weight-spmf s*) (*spmf-of-set UNIV*)
supply [*simp*] = *spmf.map-comp comp-def*

apply (*unfold uniform-sharing2*[**where** *?p1.0=p* **and** *?p2.0=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-scale-spmf*)
by *simp*

lemma *uniform-sharing′*:
⟦*is-uniform-sharing s*; *lossless-spmf s*⟧ ⟹ *map-spmf* (*get-party p*) *s* = *spmf-of-set UNIV*
by (*simp add*: *uniform-sharing lossless-weight-spmfD*)

lemma *zero-masking-uniform*:
p≠q ⟹ (*map-spmf* ((λ*t*. (*get-party p t*, *get-party q t*)) ∘ *map-sharing2* (+) *s*) *zero-sharing*) = *spmf-of-set UNIV*
proof −
assume *pq*: *p≠q*
obtain *r* **where** *r*: *q≠r r≠p*
using *role-otherE* **.**
note [*simp*] = *make-sharing′-sel*[*OF pq r*]
have *inj*: *inj* (λ*x*. (*get-party p s* + *fst x*, *get-party q s* + *snd x*))
unfolding *inj-def* **by** *simp*
have *surj*: *surj* (λ*x*. (*get-party p s* + *fst x*, *get-party q s* + *snd x*))
by (*rule finite-UNIV-inj-surj*[*OF - inj*]) *simp*
show (*map-spmf* ((λ*t*. (*get-party p t*, *get-party q t*)) ∘ *map-sharing2* (+) *s*) *zero-sharing*) = *spmf-of-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≠p2 p2≠p3 p3≠p1* ⋀*p*. *p*∈{*p1*,*p2*,*p3*} ⟹ *get-party p s* = *get-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≠p2 p2≠p3 p3≠p1*
shows *map-sharing f* (*make-sharing′ p1 p2 p3 x1 x2 x3*) = *make-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≠p2 p2≠p3 p3≠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* :: *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* :: *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*::*natL sharing*)) *t* = *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* :: *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*=*spmf-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 *map-spmf* (λ*x*. (*x*, *f x*)) (*spmf-of-set s*) = *map-spmf* (λ*y*. (*Hilbert-Choice.inv-into U f y*, *y*)) (*spmf-of-set* (*f* ‘ *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*=*spmf-of-set*])
using *assms* **by** (*auto simp*: *inv-into-f-f*[*OF inj*] *intro*!: *image-eqI*)

lemma *inv-uniform*:
inj f ⟹ *map-spmf* (λ*x*. (*x*, *f x*)) (*spmf-of-set s*) = *map-spmf* (λ*y*. (*Hilbert-Choice.inv f y*, *y*)) (*spmf-of-set* (*f* ‘ *s*))
using *inv-uniform′*[**where** *U*=*UNIV*, *simplified*] **.**

lemma *reconstruct-plus*:
reconstruct (*map-sharing2* (+) *x y*) = *reconstruct x* + *reconstruct y*
by (*cases x*; *cases y*) *simp*

lemma *reconstruct-minus*:
reconstruct (*map-sharing2* (−) *x y*) = *reconstruct x* − *reconstruct y*
by (*cases x*; *cases y*) *simp*

lemma *plus-reconstruct*:
map-sharing2 (+) *x* ‘ *reconstruct* −‘ {*y*} = *reconstruct* −‘ {*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*=*map-sharing2* (−) *t x*]) *auto*
done
done

lemma *inv-zero-sharing*:

map-spmf (λζ. (ζ, *map-sharing2* (+) *x* ζ)) *zero-sharing* = *map-spmf* (λ*y*. (*map-sharing2* (−) *y x*, *y*)) (*share-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* ← *s*; *g x* (*f x*)}) = (*do* {(*x*,*y*) ← *map-spmf* (λ*x*. (*x*, *f x*)) *s*; *g x y*})
unfolding *bind-map-spmf comp-def* **by** *simp*

lemma *hoist-map-spmf'*:
(*do* {*x* ← *s*; *g x* (*f x*)}) = (*do* {(*y*,*x*) ← *map-spmf* (λ*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* ← *s*; *g x* (*HOIST-TAG* (*f x*))}) = (*do* {(*x*,*y*) ← *map-spmf* (λ*x*. (*x*, *f x*)) *s*; *g x y*})
unfolding *HOIST-TAG-def* **using** *hoist-map-spmf* **.**

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

lemma *shift-sharing-make-sharing*:
shift-sharing (*make-sharing x1 x2 x3*) = *make-sharing x3 x1 x2*
unfolding *shift-sharing-def*
by *simp*

lemma *reconstruct-share-nat*:
map-spmf (λ*xs*. (*xs*, *reconstruct xs*)) (*share-nat x*) = *map-spmf* (λ*xs*. (*xs*, *x*)) (*share-nat x*) **for** *x*
unfolding *share-nat-def* **by** (*auto cong*: *map-spmf-cong-simp*)

lemma *weight-share-nat*:
weight-spmf (*share-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*)
unbundle *no m-inv-syntax*

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

fun *shuffleF* :: *natL sharing list* ⇒ *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* ⇒ *nat*
type-synonym *phase-msg* = *zero-sharing* × *party2-data* × *party01-permutation*

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

definition *aby3-stack-sharing* :: *Role* ⇒ *natL sharing* ⇒ *natL sharing* **where**
  aby3-stack-sharing r s = *make-sharing′ r* (*next-role r*) (*prev-role r*)
      (*get-party r s*)
      (*get-party* (*next-role r*) *s* + *get-party* (*prev-role r*) *s*)
      0

definition *aby3-do-permute* :: *Role* ⇒ *natL sharing list* ⇒ (*phase-msg* × *natL sharing list*) *spmf* **where**
  aby3-do-permute r x = (*do* {
    let n = *length x*;
    ζ ← *sequence-spmf* (*replicate n zero-sharing*);
    π ← *spmf-of-set* {*π*. *π permutes* {..<*n*}};
    let x2 = *map* (*get-party* (*prev-role r*)) *x*;
    let y′ = *map* (*aby3-stack-sharing r*) *x*;
    let y = *map2* (*map-sharing2* (+)) (*permute-list π y′*) *ζ*;
    let msg = (*ζ*, *x2*, *π*);
    return-spmf (*msg*, *y*)

```
  })

definition aby3-shuffleR :: natL sharing list ⇒ (role-msg sharing × natL sharing
list) spmf where
  aby3-shuffleR x = (do {
    ((ζa,x',πa), a) ← aby3-do-permute Party1 x;      — 1st round
    ((ζb,a',πb), b) ← aby3-do-permute Party2 a;      — 2nd round
    ((ζc,b',πc), c) ← aby3-do-permute Party3 b;      — 3rd round
      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 (msg, c)
  })

definition aby3-shuffleF :: natL sharing list ⇒ natL sharing list spmf where
  aby3-shuffleF x = (do {
    π ← spmf-of-set {π. π permutes {..<length x}};
    let x1 = map reconstruct x;
    let xπ = permute-list π x1;
    y ← sequence-spmf (map share-nat xπ);
    return-spmf y
  })

definition S1 :: natL list ⇒ natL list ⇒ role-msg spmf where
    S1 x1 yc1 = (do {
      let n = length x1;

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

       ya1::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

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

— round 1
       let ζa1 = map2 (−) ya1 (permute-list πa x1);

— round 2
       let ζb1 = yb1;
```

— round 3
```
      let b' = yb2;
      let ζc1 = map2 (−) (yc1) (permute-list πc (map2 (+) yb1 yb2));  —
```
non-free message

```
    let msg1 = ((ζa1, ζb1, ζc1), b', πa, πc);
    return-spmf msg1
  })
```

definition *S2* :: *natL list* ⇒ *natL list* ⇒ *role-msg spmf* **where**

```
  S2 x2 yc2 = (do {
    let n = length x2;
    x3 ← sequence-spmf (replicate n (spmf-of-set UNIV));

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

      ya2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

      yb2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
```

— round 1
```
      let x' = x3;
      let ζa2 = map2 (−) ya2 (permute-list πa (map2 (+) x2 x3));
```

— round 2
```
      let ζb2 = map2 (−) yb2 (permute-list πb ya2);
```

— round 3
```
      let ζc2 = yc2;  — non-free message

    let msg2 = ((ζa2, ζb2, ζc2), x', πa, πb);
    return-spmf msg2
  })
```

definition *S3* :: *natL list* ⇒ *natL list* ⇒ *role-msg spmf* **where**

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

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

      ya3::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
      ya1::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

      yb3::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
```

— round 1

let ζa3 = ya3;

— round 2

let a′ = ya1;
let ζb3 = map2 (−) yb3 (permute-list πb (map2 (+) ya3 ya1));

— round 3

let ζc3 = map2 (−) yc3 (permute-list πc yb3); — non-free message

let msg3 = ((ζa3, ζb3, ζc3), a′, πb, πc);
return-spmf msg3

})

definition *S* :: *Role* ⇒ *natL list* ⇒ *natL list* ⇒ *role-msg spmf* **where**
S r = *get-party r* (*make-sharing S1 S2 S3*)

definition *is-uniform-sharing-list* :: *natL sharing list spmf* ⇒ *bool* **where**
is-uniform-sharing-list xss ⟷ (∃ *xs*. *xss* = *bind-spmf xs* (*sequence-spmf* ∘ *map share-nat*))

lemma *case-prod-nesting-same*:
case-prod (λ*a b*. *f* (*case-prod g x*) *a b*) *x* = *case-prod* (λ*a b*. *f* (*g a b*) *a b*) *x*
by (*cases x*) *simp*

lemma *zip-map-map-same*:
map (λ*x*. (*f x*, *g x*)) *xs* = *zip* (*map f xs*) (*map g xs*)
unfolding *zip-map-map*
unfolding *zip-same-conv-map*
by *simp*

lemma *dup-map-eq*:
length xs = *length ys* ⟹ (*xs*, *map2 f ys xs*) = (λ*xys*. (*map fst xys*, *map snd xys*)) (*map2* (λ*x y*. (*y*, *f x y*)) *ys xs*)
by (*auto simp*: *map-snd-zip*[*unfolded snd-def*])

abbreviation *map2-spmf f xs ys* ≡ *map-spmf* (*case-prod f*) (*pair-spmf xs ys*)
abbreviation *map3-spmf f xs ys zs* ≡ *map2-spmf* (λ*a*. *case-prod* (*f a*)) *xs* (*pair-spmf ys zs*)

lemma *map-spmf-cong2*:
assumes *p* = *map-spmf m q* ⋀*x*. *x*∈*set-spmf q* ⟹ *f* (*m x*) = *g x*
shows *map-spmf f p* = *map-spmf g q*
using *assms* **by** (*simp add*: *spmf.map-comp cong*: *map-spmf-cong*)

lemma *bind-spmf-cong2*:
assumes *p* = *map-spmf m q* ⋀*x*. *x*∈*set-spmf q* ⟹ *f* (*m x*) = *g x*

shows *bind-spmf p f* = *bind-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* ⟹ *map2-spmf* (*map2 f*) (*sequence-spmf xss*) (*sequence-spmf yss*) = *sequence-spmf* (*map2* (*map2-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*=λ((*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* ⇒ ′*b* ⇒ ′*c* ⇒ ′*d*) ⇒ ′*a list* ⇒ ′*b list* ⇒ ′*c list* ⇒ ′*d list*
where
map3 f a b c ≡ *map2* (λ*a* (*b*,*c*). *f a b c*) *a* (*zip b c*)

lemma *map3-spmf-map3-sequence*:
length xss = *length yss* ⟹ *length yss* = *length zss* ⟹ *map3-spmf* (*map3 f*) (*sequence-spmf xss*) (*sequence-spmf yss*) (*sequence-spmf zss*) = *sequence-spmf* (*map3* (*map3-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*=λ((*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*:
 $x \in A \times B \Longrightarrow snd\ x \in B$
 by *auto*

lemma *list-map-cong2*:
 $x = map\ m\ y \Longrightarrow (\bigwedge z.\ z{\in}set\ y\ {=}simp{=}{>}\ f\ (m\ z) = g\ z) \Longrightarrow map\ f\ x = map\ g\ y$
 unfolding *simp-implies-def*
 by *simp*

lemma *map-swap-zip*:
 map prod.swap (zip xs ys) = 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 = length x ⟹
  map-spmf (*λζs.* (*ζs*, *map2* (*map-sharing2* (+)) *x ζs*)) (*sequence-spmf* (*replicate n zero-sharing*))
  =
  map-spmf (*λys.* (*map2* (*map-sharing2* (−)) *ys x*, *ys*)) (*sequence-spmf* (*map* (*share-nat* ∘*reconstruct*) *x*))
proof −
 assume *n*: *n = length x*
 have *map-spmf* (*λζs.* (*ζs*, *map2* (*map-sharing2* (+)) *x ζs*)) (*sequence-spmf* (*replicate n zero-sharing*))
=
map2-spmf (*λζs x.* (*ζs*, *map2* (*map-sharing2* (+)) *x ζs*)) (*sequence-spmf* (*replicate n zero-sharing*)) (*sequence-spmf* (*map return-spmf x*))
  unfolding *sequence-map-return-spmf*
  apply (*rule map-spmf-cong2*[**where** *m=fst*])
  subgoal by *simp*
  subgoal by (*auto simp*: *case-prod-unfold dest*: *in-pairD2*)
  done

 also have ... = *map-spmf* (*λζxs.* (*map fst ζxs*, *map snd ζxs*)) (*map2-spmf* (*map2* (*λζ x.* (*ζ*, *map-sharing2* (+) *x ζ*))) (*sequence-spmf* (*replicate n zero-sharing*)) (*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* (*λζxs.* (*map fst ζxs*, *map snd ζxs*)) (*map2-spmf* (*map2*

(λy x. (map-sharing2 (−) y x, y))) (sequence-spmf (map (share-nat ∘ 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 (λys x. (map2 (map-sharing2 (−)) ys x, ys)) (sequence-spmf (map (share-nat ∘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 (λys. (map2 (map-sharing2 (−)) ys x, ys)) (sequence-spmf (map (share-nat ∘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 ∘ (case-prod (map-sharing2 f)) = case-prod f ∘ 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 ∘ 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 $\bigwedge x.\ f\ (f'\ x) = x$ $f' = inv\ f$ $map\text{-}spmf\ f'\ x = y$
shows $x = map\text{-}spmf\ f\ y$
proof −
have *surj*: *surj f*
apply (*rule surjI*) **using** *assms*(*1*) **.**
have $map\text{-}spmf\ f\ (map\text{-}spmf\ f'\ x) = map\text{-}spmf\ f\ y$
unfolding *assms*(*3*) **..**
thus *?thesis*
using *assms*(*1*) **by** (*simp add*: *spmf.map-comp surj-iff comp-def*)
qed

lemma *lift-map-spmf-pairs*:
map2-spmf f = *F* ⟹ *map-spmf* (*case-prod f*) (*pair-spmf A B*) = *F A B*
by *auto*

lemma *measure-pair-spmf-times'*:
C = *A* × *B* ⟹ *measure* (*measure-spmf* (*pair-spmf p q*)) *C* = *measure* (*measure-spmf p*) *A* ∗ *measure* (*measure-spmf q*) *B*
by (*simp add*: *measure-pair-spmf-times*)

lemma *map-spmf-pairs-tmp*:
map-spmf (λ(*a*,*b*,*c*,*d*,*e*,*f*,*g*). (*a*,*e*,*b*,*f*,*c*,*g*,*d*)) (*pair-spmf A* (*pair-spmf B* (*pair-spmf C* (*pair-spmf D* (*pair-spmf E* (*pair-spmf F G*))))))
= (*pair-spmf A* (*pair-spmf E* (*pair-spmf B* (*pair-spmf F* (*pair-spmf C* (*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*:
(*case case x of* (*a*, *b*, *c*, *d*, *e*, *f*, *g*) ⇒ (*a*, *e*, *b*, *f*, *c*, *g*, *d*) *of*
(*ya*, *yb*, *yc*, *yd*, *ye*, *yf*, *yg*) ⇒ *F ya yb yc yd ye yf yg*)
= (*case x of* (*a*,*b*,*c*,*d*,*e*,*f*,*g*) ⇒ *F a e b f c g d*)
by (*cases x*) *simp*

lemma *bind-spmf-permutes-cong*:
(⋀π. π *permutes* {..<(*x*::*nat*)} ⟹ *f* π = *g* π)
⟹ *bind-spmf* (*spmf-of-set* {π. π *permutes* {..<*x*}}) *f* = *bind-spmf* (*spmf-of-set* {π. π *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* ⟷ *list-all2* (λ*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*:
(⋀*l*. *map reconstruct l* = *x* ⟹ *f l* = *g l*)
⟹ *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*:
(⋀*x*. *x*∈*set xs* ⟹ *reconstruct* (*f x*) = *reconstruct x*) ⟹ *map* (*reconstruct* ∘ *f*) *xs* = *map reconstruct xs*
by (*induction xs*) *auto*

lemma *permute-list-replicate*:
p permutes {..<*n*} ⟹ *permute-list p* (*replicate n x*) = *replicate n x*
apply (*fold map-replicate-const*[**where** *lst*=[*0*..<*n*], *simplified*])
apply (*simp add*: *permute-list-map*)
unfolding *map-replicate-const*
by *simp*

lemma *map2-minus-zero*:
length xs = *length ys* ⟹ (⋀*y*::*natL*. *y*∈*set ys* ⟹ *y* = *0*) ⟹ *map2* (−) *xs ys* = *xs*
by (*induction xs ys rule*: *list-induct2*) *auto*

lemma *permute-comp-left-inj*:
p permutes {..<*n*} ⟹ *inj* (λ*p*′. *p* ∘ *p*′)
by (*rule fun.inj-map*) (*rule permutes-inj-on*)

lemma *permute-comp-left-inj-on*:
p permutes {..<*n*} ⟹ *inj-on* (λ*p*′. *p* ∘ *p*′) *A*
using *permute-comp-left-inj inj-on-subset* **by** *blast*

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

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

lemma *permutes-inv-comp-left-right*:
πa *permutes* $\{..<n\}$ $\Longrightarrow$ πb *permutes* $\{..<n\}$ $\Longrightarrow$ *inv* $(\lambda p'.\ \pi a \circ p' \circ \pi b) = (\lambda p'.\ inv\ \pi a \circ p' \circ inv\ \pi b)$
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* $\{..<n\}$ $\Longrightarrow$ πb *permutes* $\{..<n\}$ $\Longrightarrow$ *inv* $(\lambda p'.\ \pi a \circ \pi b \circ p') = (\lambda p'.\ inv\ \pi b \circ inv\ \pi a \circ p')$
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* $\{..<n\}$ $\Longrightarrow$ πb *permutes* $\{..<n\}$ $\Longrightarrow$ *inv* $(\lambda p'.\ p' \circ \pi a \circ \pi b) = (\lambda p'.\ p' \circ inv\ \pi b \circ inv\ \pi a)$
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 $\{\pi a \circ \pi \circ \pi b\ |\pi.\ \pi\ permutes\ S\} = \{\pi.\ \pi\ permutes\ S\}$
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 $\{\pi a \circ \pi b \circ \pi\ |\pi.\ \pi$ *permutes S*$\} = \{\pi.\ \pi$ *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 $\{\pi \circ \pi a \circ \pi b\ |\pi.\ \pi$ *permutes S*$\} = \{\pi.\ \pi$ *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* ≡ *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* $(\lambda(\pi,\pi a,\pi c).\ ((\pi a, inv\ \pi a \circ \pi \circ inv\ \pi c,\pi c),\pi))$ (*pair-spmf* (*random-perm*

n) (*pair-spmf* (*random-perm n*) (*random-perm n*)))
  (**is** *?lhs* = *?rhs*)
proof −
  have *?lhs* = (*do* {πa ← *random-perm n*; πc ← *random-perm n*; (πb,*p*) ← *map-spmf* ($\lambda \pi b$. (πb,$\pi a \circ \pi b \circ \pi c$)) (*random-perm n*); *return-spmf* ((πa, πb, πc), *p*)})
    unfolding *map-spmf-conv-bind-spmf pair-spmf-alt-def*
    apply *simp*
    apply (*subst* (*4*) *bind-commute-spmf*)
    ..
  also
  have ... = (*do* {πa ← *random-perm n*; πc ← *random-perm n*; *map-spmf* (λp. ((πa, *inv* $\pi a \circ p \circ$ *inv* πc,πc),*p*)) (*random-perm n*)})
    unfolding *random-perm-def*
    supply [*intro*!] =
      bind-spmf-permutes-cong
    apply *rule+*
    apply (*subst inv-uniform*)
    subgoal for πa πc
      apply (*rule inj-compose*[*unfolded comp-def*, **where** *f*=λp. $p \circ \pi c$])
      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 image-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* ≡ *spmf-of-set* {π. π *permutes* {..<*n*::*nat*}}
  shows
    map-spmf (λ(πa,πb,πc). ((πa,πb,πc),$\pi a \circ \pi b \circ \pi c$)) (*pair-spmf* (*random-perm n*) (*pair-spmf* (*random-perm n*) (*random-perm n*)))
    = *map-spmf* (λ(π,πa,πb). ((πa,πb,*inv* $\pi b \circ$ *inv* $\pi a \circ \pi$),π)) (*pair-spmf* (*random-perm n*) (*pair-spmf* (*random-perm n*) (*random-perm n*)))
    (**is** *?lhs* = *?rhs*)
proof −
  have *?lhs* = (*do* {πa ← *random-perm n*; πb ← *random-perm n*; (πc,π) ← *map-spmf* ($\lambda \pi c$. (πc,$\pi a \circ \pi b \circ \pi c$)) (*random-perm n*); *return-spmf* ((πa, πb, πc), π)})
    unfolding *map-spmf-conv-bind-spmf pair-spmf-alt-def*

by *simp*
also
have ... = (*do* {πa ← *random-perm n*; πb ← *random-perm n*; *map-spmf* ($\lambda\pi$. ((πa, πb, *inv* πb ∘ *inv* πa ∘ π),π)) (*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* ≡ *spmf-of-set* {π. π *permutes* {..<*n*::*nat*}}
shows
map-spmf (λ(πa,πb,πc). ((πa,πb,πc),πa ∘ πb ∘ πc)) (*pair-spmf* (*random-perm n*) (*pair-spmf* (*random-perm n*) (*random-perm n*)))
= *map-spmf* (λ(π,πb,πc). ((π ∘ *inv* πc ∘ *inv* πb,πb,πc),π)) (*pair-spmf* (*random-perm n*) (*pair-spmf* (*random-perm n*) (*random-perm n*)))
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = (*do* {πb ← *random-perm n*; πc ← *random-perm n*; (πa,π) ← *map-spmf* ($\lambda\pi a$. (πa,πa ∘ πb ∘ πc)) (*random-perm n*); *return-spmf* ((πa, πb, πc), π)})
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* {πb ← *random-perm n*; πc ← *random-perm n*; *map-spmf* ($\lambda\pi$. ((π ∘ *inv* πc ∘ *inv* πb, πb, πc),π)) (*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 image-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 (λ*a b*. *return-spmf* (*f a b*)) = (λ*x*. *return-spmf* (*case-prod f x*))
by *auto*

lemma *sequence-share-nat-calc′*:
assumes *r1*≠*r2 r2*≠*r3 r3*≠*r1*
shows
sequence-spmf (*map share-nat xs*) = (*do* {
let n = *length xs*;
let random-seq = *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
(*dp*, *dpn*) ← (*pair-spmf random-seq random-seq*);
return-spmf (*map3* (λ*x a b*. *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 = *length xs*;
let random-seq = *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
(*xs*, *dp*, *dpn*) ← *pair-spmf* (*sequence-spmf* (*map return-spmf xs*)) (*pair-spmf random-seq random-seq*);
return-spmf (*map3* (λ*x a b*. *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*=*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 ∘ 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 ⟹ map2 (λ-. 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 ⟹ map2 (λa 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 (λx. (x, map reconstruct x)) (sequence-spmf (map share-nat y)) =
map-spmf (λx. (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 ← x-dist;
       (msg, y) ← 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-dist = xs ≫= (sequence-spmf ∘ 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*) ζ*a*, *map* (*get-party Party1*) ζ*b*, *map* (*get-party Party1*) ζ*c*), *b*′, π*a*, π*c*);
let msg2 = ((*map* (*get-party Party2*) ζ*a*, *map* (*get-party Party2*) ζ*b*, *map* (*get-party Party2*) ζ*c*), *x*′, π*a*, π*b*);
let msg3 = ((*map* (*get-party Party3*) ζ*a*, *map* (*get-party Party3*) ζ*b*, *map* (*get-party Party3*) ζ*c*), *a*′, π*b*, π*c*);
let msg = *make-sharing msg1 msg2 msg3*;
return-spmf (*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
ζ*a* ← *sequence-spmf* (*replicate n zero-sharing*);
π*a* ← *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* π*a y*′) ζ*a*;

— round 2
ζ*b* ← *sequence-spmf* (*replicate n zero-sharing*);
π*b* ← *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* π*b y*′) ζ*b*;

— round 3
ζ*c* ← *sequence-spmf* (*replicate n zero-sharing*);
π*c* ← *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* π*c y*′) ζ*c*;

let msg1 = ((*map* (*get-party Party1*) ζ*a*, *map* (*get-party Party1*) ζ*b*, *map* (*get-party Party1*) ζ*c*), *b*′, π*a*, π*c*);
let msg2 = ((*map* (*get-party Party2*) ζ*a*, *map* (*get-party Party2*) ζ*b*, *map* (*get-party Party2*) ζ*c*), *x*′, π*a*, π*b*);
let msg3 = ((*map* (*get-party Party3*) ζ*a*, *map* (*get-party Party3*) ζ*b*, *map* (*get-party Party3*) ζ*c*), *a*′, π*b*, π*c*);
let msg = *make-sharing msg1 msg2 msg3*;
return-spmf (*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*);

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 (permute-list πa xs));
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 (permute-list (πa ∘ πb) xs));
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 (permute-list (πa ∘ πb ∘ πc) xs));
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
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 πa πb πc 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=reconstruct and g=id and xs=xd
and ys=permute-list πa 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 πa πb πc 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=reconstruct and g=id and xs=xh
and ys=permute-list (πa ∘ πb) 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 ← 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 (prev-role Party1)) x;
```

```
        let y' = map (aby3-stack-sharing Party1) x;
        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;
        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;
        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)
  })
    unfolding Let-def ..
  finally have hoisted-save: ?lhs = ... .
  let ?hoisted = ...

  { assume r: r = Party1
    have project-to-Party1:
    ?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 ζa1 = map (case-prod (−)) (zip (map (get-party Party1) a) (map
(get-party Party1) (permute-list πa y')));

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

let y′ = map (aby3-stack-sharing Party2) a;
let ζb1 = map (case-prod (−)) (zip (map (get-party Party1) b) (map (get-party Party1) (permute-list πb y′)));

— round 3
let b′ = map (get-party Party2) b;
let y′ = map (aby3-stack-sharing Party3) b;
let ζc1 = map (case-prod (−)) (zip (map (get-party Party1) c) (map (get-party Party1) (permute-list πc y′)));

let msg1 = ((ζa1, ζb1, ζc1), b′, πa, πc);
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*:
. . . = (*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 ζa1 = map2 (−) (map (get-party Party1) a) (permute-list πa (map (get-party Party1) y′));

— round 2
let a′ = map (get-party Party1) a;
let y′ = map (aby3-stack-sharing Party2) a;
let ζb1 = map2 (−) (map (get-party Party1) b) (permute-list πb (map (get-party Party1) y′));

— round 3
let b′ = map (get-party Party2) b;
let y′ = map (aby3-stack-sharing Party3) b;
let ζc1 = map2 (−) (map (get-party Party1) c) (permute-list πc (map (get-party Party1) y′));

let msg1 = ((ζa1, ζb1, ζc1), b′, πa, πc);
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*:
$\ldots$ = (*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 ζa1 = *map2* (−) (*map* (*get-party Party1*) *a*) (*permute-list πa* (*map* (*get-party Party1*) *x*));

— round 2
let ζb1 = *map2* (−) (*map* (*get-party Party1*) *b*) (*permute-list πb* (*replicate* (*length a*) *0*));

— 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 *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* {..<*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 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*)
})
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*:
$\ldots = (do\ \{$
xs ← *xs*;
let n = *length xs*;
x ← *sequence-spmf* (*map share-nat xs*);

(πa,πb,πc) ← *pair-spmf* (*spmf-of-set* {π. π *permutes* {..<n}}) (*pair-spmf* (*spmf-of-set* {π. π *permutes* {..<n}}) (*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* ($\pi a \circ \pi b$) *xs*));
c ← *sequence-spmf* (*map share-nat* (*permute-list* ($\pi a \circ \pi b \circ \pi c$) *xs*));

— round 1
let $\zeta a1$ = *map2* (−) (*map* (*get-party Party1*) a) (*permute-list* πa (*map* (*get-party Party1*) x));

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

— round 3
let b' = *map* (*get-party Party2*) b;
let $\zeta c1$ = *map2* (−) (*map* (*get-party Party1*) c) (*permute-list* πc (*map2* (+) (*map* (*get-party Party1*) b) (*map* (*get-party Party2*) b)));

let msg1 = (($\zeta a1$, $\zeta b1$, $\zeta c1$), b', πa, πc);
return-spmf (*map* (*get-party Party1*) x, *msg1*, c)
})
unfolding *pair-spmf-alt-def* **by** *simp*

also have *break-perms-2*:
$\ldots = (do\ \{$
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), $\pi a \circ \pi b \circ \pi c$)) (*pair-spmf* (*spmf-of-set* {π. π *permutes* {..<n}}) (*pair-spmf* (*spmf-of-set* {π. π *permutes* {..<n}}) (*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* ($\pi a \circ \pi b$) *xs*));
c ← *sequence-spmf* (*map share-nat* (*permute-list* π *xs*));

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

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

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

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

also have *break-perms-3*:
$\ldots = (do\ \{$
$xs \leftarrow xs;$
let $n = length\ xs;$
$x \leftarrow sequence\text{-}spmf\ (map\ share\text{-}nat\ xs);$

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

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

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

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

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

let $msg1 = ((\zeta a1, \zeta b1, \zeta c1), b', \pi a, \pi c);$
return-spmf $(map\ (get\text{-}party\ Party1)\ x, msg1, c)$
})
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* $\pi a \circ \pi \circ$ *inv* πc;

a1 ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
a2 ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
let a = *map3* (λ*a 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* (λ*a b c*. *make-sharing b c* (*a* − (*b* + *c*))) (*permute-list* ($\pi a \circ \pi b$) *xs*) *b1 b2*;
c ← *sequence-spmf* (*map share-nat* (*permute-list* π *xs*));

— round 1
let $\zeta a1$ = *map2* (−) (*map* (*get-party Party1*) *a*) (*permute-list* πa (*map* (*get-party Party1*) *x*));

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

— round 3
let b' = *map* (*get-party Party2*) *b*;
let $\zeta c1$ = *map2* (−) (*map* (*get-party Party1*) *c*) (*permute-list* πc (*map2* (+) (*map* (*get-party Party1*) *b*) (*map* (*get-party Party2*) *b*)));

let msg1 = (($\zeta a1$, $\zeta b1$, $\zeta c1$), 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*);

π ← *spmf-of-set* {*π*. *π* *permutes* {..<*n*}};
πa ← *spmf-of-set* {*π*. *π* *permutes* {..<*n*}};
πc ← *spmf-of-set* {*π*. *π* *permutes* {..<*n*}};

a1::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
a2::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));

b1::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
b2::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
c ← *sequence-spmf* (*map share-nat* (*permute-list* *π* *xs*));

— round 1
let ζa1 = *map2* (−) *a1* (*permute-list* *πa* (*map* (*get-party Party1*) *x*));

— round 2
let ζb1 = *b1*;

— round 3
let b′ = *b2*;
let ζc1 = *map2* (−) (*map* (*get-party Party1*) *c*) (*permute-list* *πc* (*map2* (+) *b1 b2*));

let msg1 = ((*ζa1*, *ζb1*, *ζc1*), *b*′, *πa*, *πc*);
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
unfolding *Let-def*
apply *rule*+
apply (*auto simp*: *map2-ignore1 map2-ignore2 comp-def prod.case-distrib bind-spmf-const*)
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*)})
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* (*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 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 ← *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*}};

$a \leftarrow$ *sequence-spmf* (*map share-nat* (*permute-list* πa *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* ($\pi a \circ \pi b \circ \pi c$) *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* πa 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* πb 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* πc y')));

let msg2 = (($\zeta a2$, $\zeta b2$, $\zeta c2$), x', πa, πb);
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 $\leftarrow$ *xs*;
let n = *length xs*;
x $\leftarrow$ *sequence-spmf* (*map share-nat xs*);

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

$a \leftarrow$ *sequence-spmf* (*map share-nat* (*permute-list* πa *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* ($\pi a \circ \pi b \circ \pi c$) *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* πa (*map* (*get-party Party2*) y'));

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

let y′ = map (aby3-stack-sharing Party2) a;
let ζb2 = map2 (−) (map (get-party Party2) b) (permute-list πb (map (get-party Party2) y′));

— round 3
let b′ = map (get-party Party2) b;
let y′ = map (aby3-stack-sharing Party3) b;
let ζc2 = map2 (−) (map (get-party Party2) c) (permute-list πc (map (get-party Party2) y′));

let msg2 = ((ζa2, ζb2, ζc2), x′, πa, πb);
return-spmf (map (get-party Party2) x, 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

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 x' = *map* (*get-party Party3*) *x*;
let $\zeta a2$ = *map2* (−) (*map* (*get-party Party2*) *a*) (*permute-list* πa (*map2* (+) (*map* (*get-party Party2*) *x*) (*map* (*get-party Party3*) *x*)));

— round 2
let $\zeta b2$ = *map2* (−) (*map* (*get-party Party2*) *b*) (*permute-list* πb (*map* (*get-party Party2*) *a*));

— round 3
let $\zeta c2$ = *map2* (−) (*map* (*get-party Party2*) *c*) (*permute-list* πc (*replicate* (*length b*) *0*));

let msg2 = (($\zeta a2$, $\zeta b2$, $\zeta c2$), 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* {..<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* ($\pi a \circ \pi b$) *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 $\zeta a2$ = *map2* (−) (*map* (*get-party Party2*) *a*) (*permute-list* πa (*map2* (+) (*map* (*get-party Party2*) *x*) (*map* (*get-party Party3*) *x*)));

— round 2
let $\zeta b2$ = *map2* (−) (*map* (*get-party Party2*) *b*) (*permute-list* πb (*map* (*get-party Party2*) *a*));

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

let msg2 = (($\zeta a2$, $\zeta b2$, $\zeta c2$), x', πa, πb);
return-spmf (*map* (*get-party Party2*) *x*, *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

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* (*spmf-of-set* {π. π *permutes* {..<*n*}}) (*pair-spmf* (*spmf-of-set* {π. π *permutes* {..<*n*}}) (*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* ($\pi a \circ \pi b$) *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 $\zeta a2$ = *map2* (−) (*map* (*get-party Party2*) *a*) (*permute-list* πa (*map2* (+) (*map* (*get-party Party2*) *x*) (*map* (*get-party Party3*) *x*)));

— round 2
let $\zeta b2$ = *map2* (−) (*map* (*get-party Party2*) *b*) (*permute-list* πb (*map* (*get-party Party2*) *a*));

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

let msg2 = (($\zeta a2$, $\zeta b2$, $\zeta c2$), x', πa, πb);
return-spmf (*map* (*get-party Party2*) *x*, *msg2*, *c*)
})
unfolding *pair-spmf-alt-def* **by** *simp*

also have *break-perms-2*:

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

$((\pi a,\pi b,\pi c),\pi) \leftarrow map\text{-}spmf\ (\lambda(\pi a,\pi b,\pi c).\ ((\pi a,\pi b,\pi c),\ \pi a \circ \pi b \circ \pi c))\ (pair\text{-}spmf\ (spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\})\ (pair\text{-}spmf\ (spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\})\ (spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\})));$

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

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

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

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

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

also have *break-perms-3*:
$\ldots = (do\ \{$
$xs \leftarrow xs;$
$let\ n = length\ xs;$
$x \leftarrow sequence\text{-}spmf\ (map\ share\text{-}nat\ xs);$

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

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

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

```
— 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)
   })
   apply (unfold random-perm-right)
   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;

     x2 ← sequence-spmf (replicate n (spmf-of-set UNIV));
     x3 ← sequence-spmf (replicate n (spmf-of-set UNIV));
     let x = map3 (λa b c. make-sharing′ Party2 Party3 Party1 b c (a − (b +
c))) xs x2 x3;


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

      a2 ← sequence-spmf (replicate n (spmf-of-set UNIV));
      a3 ← sequence-spmf (replicate n (spmf-of-set UNIV));
      let a = map3 (λa b c. make-sharing′ Party2 Party3 Party1 b c (a − (b +
c))) (permute-list πa xs) a2 a3;
      b2 ← sequence-spmf (replicate n (spmf-of-set UNIV));
      b3 ← sequence-spmf (replicate n (spmf-of-set UNIV));
      let b = map3 (λa b c. make-sharing′ Party2 Party3 Party1 b c (a − (b +
c))) (permute-list (πa ◦ πb) xs) b2 b3;
      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)
})
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 ← xs;
let n = length xs;

x2 ← sequence-spmf (replicate n (spmf-of-set UNIV));
x3 ← sequence-spmf (replicate n (spmf-of-set UNIV));

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

a2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
a3::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

b2::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));
b3::natL list ← sequence-spmf (replicate n (spmf-of-set UNIV));

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

— round 1

let x′ = x3;
let ζa2 = map2 (−) a2 (permute-list πa (map2 (+) x2 x3));

— round 2

let ζb2 = map2 (−) b2 (permute-list πb a2);

— round 3

let ζc2 = (map (get-party Party2) c);

let msg2 = ((ζa2, ζb2, ζc2), x′, πa, πb);
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* ($\pi a \circ \pi b$) *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 $\zeta a3$ = *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 $\zeta b3$ = *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*:
$\ldots$ = (*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*);

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 = (*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*)
})
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* (*spmf-of-set* {*π*. *π permutes* {..<*n*}}) (*pair-spmf* (*spmf-of-set* {*π*. *π permutes* {..<*n*}}) (*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 = (*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* (*spmf-of-set* {*π*. *π permutes* {..<*n*}}) (*pair-spmf* (*spmf-of-set* {*π*. *π permutes* {..<*n*}}) (*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 π xs*));

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

— round 2

let $a' = map$ (*get-party Party1*) *a*;

let $\zeta b3$ = *map2* (−) (*map* (*get-party Party3*) *b*) (*permute-list* πb (*map2* (+) (*map* (*get-party Party3*) *a*) (*map* (*get-party Party1*) *a*)));

— round 3

let $\zeta c3$ = *map2* (−) (*map* (*get-party Party3*) *c*) (*permute-list* πc (*map* (*get-party Party3*) *b*));

let msg3 = (($\zeta a3$, $\zeta b3$, $\zeta c3$), a', πb, πc);
return-spmf (*map* (*get-party Party3*) *x*, *msg3*, *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 ← *spmf-of-set* {π. π *permutes* {..<*n*}};
πb ← *spmf-of-set* {π. π *permutes* {..<*n*}};
πc ← *spmf-of-set* {π. π *permutes* {..<*n*}};
let $\pi a = \pi \circ inv\ \pi c \circ inv\ \pi b$;

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

— round 1

let $\zeta a3$ = (*map* (*get-party Party3*) *a*);

— round 2

let $a' = map$ (*get-party Party1*) *a*;

let $\zeta b3$ = *map2* (−) (*map* (*get-party Party3*) *b*) (*permute-list* πb (*map2* (+) (*map* (*get-party Party3*) *a*) (*map* (*get-party Party1*) *a*)));

— round 3

let $\zeta c3$ = *map2* (−) (*map* (*get-party Party3*) *c*) (*permute-list* πc (*map* (*get-party Party3*) *b*));

let msg3 = (($\zeta a3$, $\zeta b3$, $\zeta c3$), a', πb, πc);
return-spmf (*map* (*get-party Party3*) *x*, *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*:

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

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

$a3 \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$a1 \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$let\ a = map3\ (\lambda a\ b\ c.\ make\text{-}sharing'\ Party3\ Party1\ Party2\ b\ c\ (a - (b + c)))\ (permute\text{-}list\ \pi a\ xs)\ a3\ a1;$
$b3 \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$b1 \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$let\ b = map3\ (\lambda a\ b\ c.\ make\text{-}sharing'\ Party3\ Party1\ Party2\ b\ c\ (a - (b + c)))\ (permute\text{-}list\ (\pi a \circ \pi b)\ xs)\ b3\ b1;$
$c \leftarrow sequence\text{-}spmf\ (map\ share\text{-}nat\ (permute\text{-}list\ \pi\ xs));$

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

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

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

$let\ msg3 = ((\zeta a3,\ \zeta b3,\ \zeta c3),\ a',\ \pi b,\ \pi c);$
$return\text{-}spmf\ (map\ (get\text{-}party\ Party3)\ x,\ msg3,\ c)$
$\})$
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*:
$\ldots = (do\ \{$
$xs \leftarrow xs;$
$let\ n = length\ xs;$

$x \leftarrow sequence\text{-}spmf\ (map\ share\text{-}nat\ xs);$

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

let $\pi a = \pi \circ$ *inv* $\pi c \circ$ *inv* πb;

a3::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
a1::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));

b3::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
b1::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));

c ← *sequence-spmf* (*map share-nat* (*permute-list* π *xs*));

— round 1
let $\zeta a3 = a3$;

— round 2
let $a' = a1$;
let $\zeta b3$ = *map2* (−) *b3* (*permute-list* πb (*map2* (+) *a3 a1*));

— round 3
let $\zeta c3$ = *map2* (−) (*map* (*get-party Party3*) *c*) (*permute-list* πc *b3*);

let msg3 = (($\zeta a3$, $\zeta b3$, $\zeta c3$), 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
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*)})
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*=*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* (*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 (λ(n, xs). n#xs) X
  by (auto intro!: inj-onI)

lemma finite-indicator-eq-sum:
  finite A ⟹ 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 spmf-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)) = spmf-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 ∧ set l ⊆ 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 (λx y. f (g x y)) x y z
  by (auto simp: zip-assoc map-zip-map)

lemma inv-add-sequence:
```

assumes *n* = *length x*
shows
map-spmf (λζ::*natL list*. (ζ, *map2* (+) ζ *x*)) (*sequence-spmf* (*replicate n* (*spmf-of-set* *UNIV*)))
=
map-spmf (λ*y*. (*map2* (−) *y x*, *y*)) (*sequence-spmf* (*replicate n* (*spmf-of-set* *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*=*spmf-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 = (*do* {
let n = *length x1*;

π*a* ← *spmf-of-set* {π. π *permutes* {..<*n*}};
π*c* ← *spmf-of-set* {π. π *permutes* {..<*n*}};
ζ*a1*::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
yb1::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
yb2::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));

let ζ*c1* = *map2* (−) (*yc1*) (*permute-list* π*c* (*map2* (+) *yb1 yb2*));
return-spmf ((ζ*a1*, *yb1*, ζ*c1*), *yb2*, π*a*, π*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*=λ*x*. *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* (*spmf-of-set UNIV*));
πa ← *spmf-of-set* {*π*. *π permutes* {..<*n*}};
πb ← *spmf-of-set* {*π*. *π permutes* {..<*n*}};
ζa2::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
ζb2::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));

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

πb ← *spmf-of-set* {π. π *permutes* {..<*n*}};
πc ← *spmf-of-set* {π. π *permutes* {..<*n*}};
ya3::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
ya1::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));
$\zeta b3$::*natL list* ← *sequence-spmf* (*replicate n* (*spmf-of-set UNIV*));

let $\zeta c3$ = *map2* (−) *yc3* (*permute-list* πc (*map2* (+) $\zeta b3$ (*permute-list* πb (*map2* (+) *ya3 ya1*))));
return-spmf ((*ya3*, $\zeta b3$, $\zeta c3$), *ya1*, πb, πc)
})

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

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

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

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

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

lemma *reconstruct-do-mul*:
  reconstruct (*map-sharing2 do-mul xys* (*shift-sharing xys*)) = *reconstruct* (*map-sharing fst xys*) ∗ *reconstruct* (*map-sharing snd xys*)
  supply [*simp*] = *shift-sharing-make-sharing do-mul-def*
  by (*cases xys*) (*auto simp*: *algebra-simps*)

theorem *mul-spec*:
  fixes *x-dist y-dist* :: *natL sharing spmf*
  assumes *is-uniform-sharing x-dist is-uniform-sharing y-dist*
  shows

secure:
(*do* {
xs ← *x-dist*;
ys ← *y-dist*;
let inps = *prod-sharing xs ys*;
(*msgs*, *outps*) ← (*aby3-mulR inps*);
let view = (*get-party p inps*, *get-party p msgs*);
return-spmf (*view*, *outps*)
})
=
(*do* {
xs ← *x-dist*;
ys ← *y-dist*;
let inps = *prod-sharing xs ys*;
outps ← *aby3-mulF inps*;
msg ← *S* (*get-party p inps*) (*get-party p outps*);
let view = (*get-party p inps*, *msg*);
return-spmf (*view*, *outps*)
}) (**is** *?secure*)
and
correct:
is-uniform-sharing (*do* {
xs ← *x-dist*;
ys ← *y-dist*;
aby3-mulF (*prod-sharing xs ys*)
}) (**is** *?uniform*)
proof −
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*=λx. *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*=λx. *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 (*'val*, *'msg*) *writer-spmf* = (*'val* × *'msg*) *spmf*

definition *bind-writer-spmf* :: (*'val1*, *'msg1*) *writer-spmf* ⇒ (*'val1* ⇒ (*'val2*, *'msg2*) *writer-spmf*) ⇒ (*'val2*, (*'msg1* × *'msg2*)) *writer-spmf* **where**
bind-writer-spmf x f = *bind-spmf x* (λ(*val1*, *msg1*). *map-spmf* (λ(*val2*, *msg2*). (*val2*, (*msg1*, *msg2*))) (*f val1*))

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind-writer-spmf*

definition *flatten-sharingF* :: *natL sharing* ⇒ *natL sharing spmf* **where**
flatten-sharingF s = *share-nat* (*reconstruct s*)

definition *flatten-sharingR* :: *Role* ⇒ *natL sharing* ⇒ (*natL sharing*, *natL*) *writer-spmf* **where**
flatten-sharingR p s = *do* {
ζ ← *zero-sharing*;
let r = *map-sharing2* (+) *s* ζ;
return-spmf (*r*, (*get-party p* ζ))
}

definition *flatten-sharingS* :: *natL* ⇒ *natL* ⇒ *natL spmf* **where**
flatten-sharingS inp outp = *return-spmf* (*outp* − *inp*)

lemma *flatten-sharing-spec*:
flatten-sharingR p x = *do* {
y ← *flatten-sharingF x*;
msg ← *flatten-sharingS* (*get-party p x*) (*get-party p y*);
return-spmf (*y*, *msg*)
}
unfolding *flatten-sharingR-def*
unfolding *flatten-sharingF-def*
unfolding *flatten-sharingS-def*
apply *simp*
apply (*fold zero-masking-eq-share-nat*)
unfolding *map-spmf-conv-bind-spmf*
apply *simp*
done

definition *aby3-mulR'* :: *Role* ⇒ *natL* ⇒ *natL* ⇒ (*natL sharing*, ((*natL* × *natL*) × (*natL* × *natL*)) × *natL*) *writer-spmf* **where**
aby3-mulR' p x y = *do* {
xs ← *share-nat x*;
ys ← *share-nat y*;
let xy = *prod-sharing xs ys*;
let xy-shift = *shift-sharing xy*;
let z-raw = *map-sharing2 do-mul xy xy-shift*;
(*z*, *ζ-msg*) ← *flatten-sharingR p z-raw*;
return-spmf (*z*, (((*get-party p xs*, *get-party p ys*), *get-party p xy-shift*), *ζ-msg*))
}

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

definition *aby3-mulS'* :: *natL* ⇒ (((*natL* × *natL*) × (*natL* × *natL*)) × *natL*) *spmf* **where**
aby3-mulS' z = *do* {
x1 ← *spmf-of-set UNIV*;
x2 ← *spmf-of-set UNIV*;
y1 ← *spmf-of-set UNIV*;
y2 ← *spmf-of-set UNIV*;
let ζ = *z* − (*x1* ∗ *y1* + *x1* ∗ *y2* + *x2* ∗ *y1*);
return-spmf (((*x1*, *y1*), (*x2*, *y2*)), ζ)
}

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

lemma *reconstruct-share-nat'*:
pred-spmf (λ*s*. *reconstruct s* = *x*) (*share-nat x*)
unfolding *pred-spmf-def*
apply *auto*
unfolding *set-spmf-share-nat*
apply *simp*
done

lemma *share-nat-cong*:
x = *y* ⟹ (⋀*s*. *reconstruct s* = *x* ⟹ *f s* = *g s*) ⟹ *bind-spmf* (*share-nat x*) *f* = *bind-spmf* (*share-nat y*) *g*

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*) (λ*msg*. *return-spmf* (*r*, *msg*))
by *simp*

schematic-goal *aby3-mul-spec*:
aby3-mulR′ p x y =
bind-spmf (*aby3-mulF′ x y*) (λ*z*.
bind-spmf (*?aby3-mulS′* (*get-party p z*)) (λ*msg*.
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.