ABY3 Multiplication and Array Shuffling

Shuwei Hu

April 18, 2024

Abstract

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

Contents

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

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

consts $L :: nat$

specification (L) *L-gt-1*: $L > 1$
  ⟨*proof*⟩

typedef $natL = \{0..<int\ L\}$
  ⟨*proof*⟩

setup-lifting *type-definition-natL*

instantiation $natL :: comm\text{-}ring\text{-}1$
begin

lift-definition *zero-natL* :: *natL* **is** *0*

⟨*proof*⟩

lift-definition *one-natL* :: *natL* **is** *1*
⟨*proof*⟩

lift-definition *uminus-natL* :: *natL* ⇒ *natL* **is** λ*x*. (−*x*) *mod int L*
⟨*proof*⟩

lift-definition *plus-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** λ*x y*. (*x* + *y*) *mod int L*
⟨*proof*⟩

lift-definition *minus-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** λ*x y*. (*x* − *y*) *mod int L*
⟨*proof*⟩

lift-definition *times-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** λ*x y*. (*x* ∗ *y*) *mod int L*
⟨*proof*⟩

instance
⟨*proof*⟩

end
typ *int*

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

lift-definition *inf-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** *inf*
⟨*proof*⟩

lift-definition *sup-natL* :: *natL* ⇒ *natL* ⇒ *natL* **is** *sup*
⟨*proof*⟩

lift-definition *less-eq-natL* :: *natL* ⇒ *natL* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩

lift-definition *less-natL* :: *natL* ⇒ *natL* ⇒ *bool* **is** *less* ⟨*proof*⟩

lift-definition *top-natL* :: *natL* **is** *int L*−*1*
⟨*proof*⟩

lift-definition *bot-natL* :: *natL* **is** *0*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *natL* :: *semiring-modulo*
begin

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

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

instance
⟨*proof*⟩

end

instance *natL* :: *finite*
⟨*proof*⟩

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

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

datatype *Role* = *Party1* | *Party2* | *Party3*

lemma *Role-exhaust'*[*dest*!]:
r ≠ *Party1* ⟹ *r* ≠ *Party2* ⟹ *r* ≠ *Party3* ⟹ *False*
⟨*proof*⟩

lemma *Role-exhaust*:
(*r1*::*Role*)≠*r2* ⟹ *r2*≠*r3* ⟹ *r3*≠*r1* ⟹ *r*=*r1* ∨ *r*=*r2* ∨ *r*=*r3*
⟨*proof*⟩

type-synonym *'a sharing* = *Role* ⇒ *'a*

instance *Role* :: *finite*
⟨*proof*⟩

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

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

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

⟨*proof*⟩

definition *make-sharing′* :: *Role* ⇒ *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*
⟨*proof*⟩

lemma *make-sharing-sel*[*simp*]:
shows
get-party Party1 (*make-sharing x1 x2 x3*) = *x1*
get-party Party2 (*make-sharing x1 x2 x3*) = *x2*
get-party Party3 (*make-sharing x1 x2 x3*) = *x3*
⟨*proof*⟩

primrec *next-role* **where**
next-role Party1 = *Party2*
| *next-role Party2* = *Party3*
| *next-role Party3* = *Party1*

primrec *prev-role* **where**
prev-role Party1 = *Party3*
| *prev-role Party2* = *Party1*
| *prev-role Party3* = *Party2*

lemma *next-sharing-neq-self*[*simp*]:
next-role r = *r* ⟷ *False r* = *next-role r* ⟷ *False*
⟨*proof*⟩

lemma *prev-sharing-neq-self*[*simp*]:
prev-role r = *r* ⟷ *False r* = *prev-role r* ⟷ *False*
⟨*proof*⟩

lemma *next-sharing-neq-prev*[*simp*]:
next-role r = *prev-role r* ⟷ *False prev-role r* = *next-role r* ⟷ *False*
⟨*proof*⟩

lemma *role-otherE*:
obtains *r* :: *Role* **where** *r0*≠*r r*≠*r1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sharing-cases*[*cases type*]:
obtains *x1 x2 x3* **where** *s* = *make-sharing x1 x2 x3*
⟨*proof*⟩

lemma *sharing-cases′*:
assumes *p1*≠*p2 p2*≠*p3 p3*≠*p1*
obtains *x1 x2 x3* **where** *s* = *make-sharing′ p1 p2 p3 x1 x2 x3*
⟨*proof*⟩

lemma *make-sharing-collapse*[*simp*]:
make-sharing (*get-party Party1 s*) (*get-party Party2 s*) (*get-party Party3 s*) = *s*
⟨*proof*⟩

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*⟧
⟹ *x* = *y*
⟨*proof*⟩

lemma *sharing-eqI*[*intro*]:
(⋀*r*. *get-party r x* = *get-party r y*) ⟹ *x* = *y*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *prod-sharing-map-sel*[*simp*]:
map-sharing fst (*prod-sharing x y*) = *x*
map-sharing snd (*prod-sharing x y*) = *y*
⟨*proof*⟩

definition *shift-sharing* :: *'a sharing* ⇒ *'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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *derep-sharing-reshare*[*simp*]:
derep-sharing (*reshare s*) = *s*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-repsharing-def-alt*:
valid-repsharing s ⟹ *map-repsharing f s* = *map-sharing* (*map-prod f f*) *s*
⟨*proof*⟩

lemma *reshare-derep-sharing*[*simp*]:
valid-repsharing s ⟹ *reshare* (*derep-sharing s*) = *s*
⟨*proof*⟩

lemma *valid-reshare*[*simp*]:
valid-repsharing (*reshare s*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*)

⟨*proof*⟩

lemma *corec-prod-inject*[*simp*]:
corec-prod f g = *corec-prod f′ g′* ⟷ *f* = *f′* ∧ *g* = *g′*
⟨*proof*⟩

lemma *inj-on-corec-prodI*:
inj-on f A ∨ *inj-on g A* ⟹ *inj-on* (λ*x*. *corec-prod* (*f x*) (*g x*)) *A*
⟨*proof*⟩

lemma *inj-on-reshare*[*simp*]:
inj-on reshare A
⟨*proof*⟩

lemma *inj-on-make-repsharing-eq-sharing*:
inj-on (λ*x*. *make-repsharing* (*f x*) (*g x*) (*h x*)) *A* ⟷ *inj-on* (λ*x*. *make-sharing* (*f x*) (*g x*) (*h x*)) *A*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *reshare-inject*[*simp*]:
reshare a = *reshare b* ⟷ *a* = *b*
⟨*proof*⟩

lemma *make-repsharing-inject*[*simp*]:
make-repsharing x1 x2 x3 = *make-repsharing y1 y2 y3* ⟷ *x1*=*y1* ∧ *x2*=*y2* ∧ *x3*=*y3*
⟨*proof*⟩

lemma *inj-on-make-sharingI*:
inj-on f A ∨ *inj-on g A* ∨ *inj-on h A* ⟹ *inj-on* (λ*x*. *make-sharing* (*f x*) (*g x*) (*h x*)) *A*
⟨*proof*⟩

lemma *inj-on-make-repsharingI*:
inj-on f A ∨ *inj-on g A* ∨ *inj-on h A* ⟹ *inj-on* (λ*x*. *make-repsharing* (*f x*) (*g x*) (*h x*)) *A*
⟨*proof*⟩

lemma *finite′-card-0*: *finite′ A* $\longleftrightarrow$ *card A* $\neq$ *0*
and *card-0-finite′*: *card A* = *0* $\longleftrightarrow$ $\neg$*finite′ A*
⟨*proof*⟩

lemma *spmf-of-set-bind*:
assumes *fin*: *finite A*
and *disj*: *disjoint-family-on f A*
and *card*: $\bigwedge$*a*. *a*∈*A* $\Longrightarrow$ *card* (*f a*) = *c*
shows *spmf-of-set* (*A* ⤜ *f*) = *spmf-of-set A* ⤜ (λ*x*. *spmf-of-set* (*f x*))

⟨*proof*⟩

lemma *ap-set-singleton*:
ap-set {*f*} *A* = *f* ‘ *A*
⟨*proof*⟩

lemma *ap-set-Union*:
ap-set F A = (⋃*f*∈*F*. *f* ‘ *A*)
⟨*proof*⟩

lemma *ap-set-curry*:
ap-set (*ap-set F A*) *B* = *ap-set* (*case-prod* ‘ *F*) (*A* × *B*)
⟨*proof*⟩

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*)
})
⟨*proof*⟩

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)
  })
  ⟨proof⟩

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

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

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

lemma valid-nat-repsharing-reshare[simp]:
  valid-nat-repsharing (reconstruct s) (reshare s)
  ⟨proof⟩

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

lemma repshare-nat-def-alt':
  repshare-nat x = spmf-of-set (Collect (valid-nat-repsharing x))
  ⟨proof⟩

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

lemma prod-sharing-map-fst-snd[simp]:
  prod-sharing (map-sharing fst s) (map-sharing snd s) = s
  ⟨proof⟩

end
theory Spmf-Common
  imports CryptHOL.CryptHOL
begin

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

lemma *mk-lossless-back-eq*:
scale-spmf (*weight-spmf s*) (*mk-lossless s*) = *s*
⟨*proof*⟩

lemma *cond-spmf-enforce*:
cond-spmf sx (*Collect A*) = *mk-lossless* (*enforce-spmf A sx*)
⟨*proof*⟩

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

lemma *rel-scale-spmf-refl*:
rel-scale-spmf s s
⟨*proof*⟩

lemma *rel-scale-spmf-sym*:
rel-scale-spmf s t ⟹ *rel-scale-spmf t s*
⟨*proof*⟩

lemma *rel-scale-spmf-trans*:
rel-scale-spmf s t ⟹ *rel-scale-spmf t u* ⟹ *rel-scale-spmf s u*
⟨*proof*⟩

lemma *rel-scale-spmf-equiv*:
equivp rel-scale-spmf
⟨*proof*⟩

lemma *spmf-eq-iff*: *p* = *q* ⟷ (∀ *i*. *spmf p i* = *spmf q i*)
⟨*proof*⟩

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

lemma *rel-scale-spmf-None*:
rel-scale-spmf s t ⟹ *s* = *return-pmf None* ⟷ *t* = *return-pmf None*
⟨*proof*⟩

lemma *rel-scale-spmf-def-alt*:
rel-scale-spmf s t ⟷ (∃ *k*>*0*. *s* = *scale-spmf k t*)
⟨*proof*⟩

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*⟩

lemma *rel-scale-spmf-scale*:
r > *0* ⟹ *rel-scale-spmf s t* ⟹ *rel-scale-spmf s* (*scale-spmf r t*)
⟨*proof*⟩

lemma *rel-scale-spmf-mk-lossless*:
rel-scale-spmf s t ⟹ *rel-scale-spmf s* (*mk-lossless t*)
⟨*proof*⟩

lemma *rel-scale-spmf-eq-iff*:
rel-scale-spmf s t ⟹ *weight-spmf s* = *weight-spmf t* ⟹ *s* = *t*
⟨*proof*⟩

lemma *rel-scale-spmf-set-restrict*:
finite A ⟹ *rel-scale-spmf* (*restrict-spmf* (*spmf-of-set A*) *B*) (*spmf-of-set* (*A* ∩ *B*))
⟨*proof*⟩

lemma *spmf-of-set-restrict-empty*:
A ∩ *B* = {} ⟹ *restrict-spmf* (*spmf-of-set A*) *B* = *return-pmf None*
⟨*proof*⟩

lemma *spmf-of-set-restrict-scale*:
finite A ⟹ *restrict-spmf* (*spmf-of-set A*) *B* = *scale-spmf* (*card* (*A*∩*B*) / *card A*) (*spmf-of-set* (*A*∩*B*))
⟨*proof*⟩

lemma *min-em2*:
min a b = *c* ⟹ *a*≠*c* ⟹ *b*=*c*
⟨*proof*⟩

lemma *weight-0-spmf*:
weight-spmf s = *0* ⟹ *spmf s i* = *0*
⟨*proof*⟩

lemma *mk-lossless-scale-absorb*:
r > *0* ⟹ *mk-lossless* (*scale-spmf r s*) = *mk-lossless s*
⟨*proof*⟩

lemma *scale-spmf-None-iff*:
scale-spmf k s = *return-pmf None* ⟷ *k*≤*0* ∨ *s*=*return-pmf None*
⟨*proof*⟩

lemma *spmf-of-pmf-the*:
lossless-spmf s ⟹ *spmf-of-pmf* (*map-pmf the s*) = *s*
⟨*proof*⟩

lemma *lossless-mk-lossless*:
$s \neq$ *return-pmf None* $\Longrightarrow$ *lossless-spmf* (*mk-lossless s*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *pmf-spmf*:
pmf-of-spmf (*spmf-of-pmf p*) = *p*
⟨*proof*⟩

lemma *spmf-pmf*:
lossless-spmf p $\Longrightarrow$ *spmf-of-pmf* (*pmf-of-spmf p*) = *p*
⟨*proof*⟩

lemma *pmf-of-spmf-scale-spmf*:
r > *0* $\Longrightarrow$ *pmf-of-spmf* (*scale-spmf r p*) = *pmf-of-spmf p*
⟨*proof*⟩

lemma *nonempty-spmf-weight*:
p $\neq$ *return-pmf None* $\longleftrightarrow$ *weight-spmf p* > *0*
⟨*proof*⟩

lemma *pmf-of-spmf-mk-lossless*:
pmf-of-spmf (*mk-lossless p*) = *pmf-of-spmf p*
⟨*proof*⟩

lemma *spmf-pmf'*:
p $\neq$ *return-pmf None* $\Longrightarrow$ *spmf-of-pmf* (*pmf-of-spmf p*) = *mk-lossless p*
⟨*proof*⟩

lemma *rel-scale-spmf-cond-UNIV*:
rel-scale-spmf s (*cond-spmf s UNIV*)
⟨*proof*⟩

lemma *set-pmf p* ∩ *g* $\neq$ {} $\Longrightarrow$ *pmf-prob p* (*f* ∩ *g*) = *pmf-prob* (*cond-pmf p g*) *f* ∗ *pmf-prob p g*
⟨*proof*⟩

lemma *bayes*:
set-pmf p ∩ *A* $\neq$ {} $\Longrightarrow$ *set-pmf p* ∩ *B* $\neq$ {} $\Longrightarrow$
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*

⟨*proof*⟩

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

lemma *spmf-prob* = *measure-measure-spmf*
⟨*proof*⟩

lemma *spmf-prob-pmf*:
spmf-prob p A = *pmf-prob p* (*Some* ‘ *A*)
⟨*proof*⟩

lemma *bayes-spmf*:
spmf-prob (*cond-spmf p A*) *B*
= *spmf-prob* (*cond-spmf p B*) *A* ∗ *spmf-prob p B* / *spmf-prob p A*
⟨*proof*⟩

lemma *spmf-prob-pmf-of-spmf*:
spmf-prob p A = *weight-spmf p* ∗ *pmf-prob* (*pmf-of-spmf p*) *A*
⟨*proof*⟩

lemma *cond-spmf-Int*:
cond-spmf (*cond-spmf p A*) *B* = *cond-spmf p* (*A* ∩ *B*)
⟨*proof*⟩

lemma *cond-spmf-prob*:
spmf-prob p (*A* ∩ *B*) = *spmf-prob* (*cond-spmf p A*) *B* ∗ *spmf-prob p A*
⟨*proof*⟩

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

lemma *spmf-prob-empty*:
spmf-prob empty-spmf A = *0*
⟨*proof*⟩

definition *le-spmf* :: *'a spmf* ⇒ *'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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cond-spmf-mk-lossless*:
cond-spmf (*mk-lossless p*) *A* = *cond-spmf p A*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *map-spmf-map-sequence*:
map-spmf (*map f*) (*sequence-spmf xs*) = *sequence-spmf* (*map* (*map-spmf f*) *xs*)
⟨*proof*⟩

lemma *sequence-map-return-spmf*:
sequence-spmf (*map return-spmf xs*) = *return-spmf xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lossless-pair-spmf-iff*:
lossless-spmf (*pair-spmf a b*) ⟷ *lossless-spmf a* ∧ *lossless-spmf b*
⟨*proof*⟩

lemma *lossless-sequence-spmf*:
(⋀*x*. *x*∈*set xs* ⟹ *lossless-spmf x*) ⟹ *lossless-spmf* (*sequence-spmf xs*)
⟨*proof*⟩

end
theory *Sharing-Lemmas*
imports

Additive-Sharing
begin

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

lemma *share-nat-party-uniform*:
map-spmf (*get-party p*) (*share-nat x*) = *spmf-of-set UNIV*
⟨*proof*⟩

definition *is-uniform-sharing* :: *natL sharing spmf* ⇒ *bool* **where**
is-uniform-sharing s ⟷ ($\exists$ *x* :: *natL spmf*. *s* = *bind-spmf x share-nat*)

definition *is-uniform-sharing2* :: (*natL sharing* × *natL sharing*) *spmf* ⇒ *bool* **where**
is-uniform-sharing2 s ⟷ ($\exists$ *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*)
⟨*proof*⟩

lemma *share-nat-lossless*:
lossless-spmf (*share-nat x*)
⟨*proof*⟩

lemma *uniform-sharing2*:
is-uniform-sharing s $\Longrightarrow$ *p1* ≠ *p2* $\Longrightarrow$ *map-spmf* ($\lambda x.$ (*get-party p1 x*, *get-party p2 x*)) *s* = *scale-spmf* (*weight-spmf s*) (*spmf-of-set UNIV*)
⟨*proof*⟩

lemma *uniform-sharing*:
is-uniform-sharing s $\Longrightarrow$ *map-spmf* (*get-party p*) *s* = *scale-spmf* (*weight-spmf s*) (*spmf-of-set UNIV*)
⟨*proof*⟩

lemma *uniform-sharing′*:
⟦*is-uniform-sharing s*; *lossless-spmf s*⟧ $\Longrightarrow$ *map-spmf* (*get-party p*) *s* = *spmf-of-set UNIV*
⟨*proof*⟩

lemma *zero-masking-uniform*:
p≠*q* $\Longrightarrow$ (*map-spmf* (($\lambda t.$ (*get-party p t*, *get-party q t*)) ∘ *map-sharing2* (+) *s*) *zero-sharing*) = *spmf-of-set UNIV*

⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *add-sharing-inj*:
inj (*map-sharing2* (+) (*s* :: *natL sharing*))
⟨*proof*⟩

lemma *add-sharing-surj*:
surj (*map-sharing2* (+) (*s* :: *natL sharing*))
⟨*proof*⟩

lemma *sharing-add-inv-eq-minus*:
Hilbert-Choice.inv (*map-sharing2* (+) (*s*::*natL sharing*)) *t* = *map-sharing2* (−) *t s*
⟨*proof*⟩

lemma *zero-masking-eq-share-nat*:
map-spmf (*map-sharing2* (+) (*s* :: *natL sharing*)) *zero-sharing* = *share-nat* (*reconstruct s*)
⟨*proof*⟩

lemma *inv-uniform′*:
assumes *ss*: *s* ⊆ *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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *reconstruct-plus*:
reconstruct (*map-sharing2* (+) *x y*) = *reconstruct x* + *reconstruct y*
⟨*proof*⟩

lemma *reconstruct-minus*:
reconstruct (*map-sharing2* (−) *x y*) = *reconstruct x* − *reconstruct y*
⟨*proof*⟩

lemma *plus-reconstruct*:
map-sharing2 (+) *x* ‘ *reconstruct* −‘ {*y*} = *reconstruct* −‘ {*reconstruct x* + *y*}
⟨*proof*⟩

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*))
⟨*proof*⟩

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*})
⟨*proof*⟩

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*})
⟨*proof*⟩

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*})
⟨*proof*⟩

lemma *get-prev-sharing*[*simp*]:
get-party p (*shift-sharing s*) = *get-party* (*prev-role p*) *s*
⟨*proof*⟩

lemma *shift-sharing-make-sharing*:
shift-sharing (*make-sharing x1 x2 x3*) = *make-sharing x3 x1 x2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *weight-share-nat*:
weight-spmf (*share-nat x*) = *1*

⟨*proof*⟩

end
theory *Shuffle*
imports
CryptHOL.CryptHOL
Additive-Sharing
Spmf-Common
Sharing-Lemmas
begin

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

no-notation (*ASCII*) *comp* (**infixl** *o* 55)
no-notation *m-inv* (*inv*ı - [81] 80)

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 $\zeta a3 = ya3$;

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

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

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

})

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

definition *is-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*
⟨*proof*⟩

lemma *zip-map-map-same*:
map (λ*x*. (*f x*, *g x*)) *xs* = *zip* (*map f xs*) (*map g xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *in-pairD2*:
x ∈ *A* × *B* ⟹ *snd x* ∈ *B*
⟨*proof*⟩

lemma *list-map-cong2*:
x = *map m y* ⟹ (⋀*z*. *z*∈*set y* =*simp*=> *f* (*m z*) = *g z*) ⟹ *map f x* = *map g y*
⟨*proof*⟩

lemma *map-swap-zip*:
map prod.swap (*zip xs ys*) = *zip ys xs*
⟨*proof*⟩

lemma *inv-zero-sharing-sequence*:
n = *length x* ⟹
map-spmf (λζ*s*. (ζ*s*, *map2* (*map-sharing2* (+)) *x* ζ*s*)) (*sequence-spmf* (*replicate n zero-sharing*))
=
map-spmf (λ*ys*. (*map2* (*map-sharing2* (−)) *ys x*, *ys*)) (*sequence-spmf* (*map* (*share-nat* ∘*reconstruct*) *x*))
⟨*proof*⟩

lemma *get-party-map-sharing2*:
get-party p ∘ (*case-prod* (*map-sharing2 f*)) = *case-prod f* ∘ *map-prod* (*get-party p*) (*get-party p*)
⟨*proof*⟩

lemma *map-map-prod-zip*:
map (*map-prod f g*) (*zip xs ys*) = *zip* (*map f xs*) (*map g ys*)
⟨*proof*⟩

lemma *map-map-prod-zip′*:
map (*h* ∘ *map-prod f g*) (*zip xs ys*) = *map h* (*zip* (*map f xs*) (*map g ys*))
⟨*proof*⟩

lemma *eq-map-spmf-conv*:
assumes ⋀*x*. *f* (*f′ x*) = *x f′* = *inv f map-spmf f′ x* = *y*
shows *x* = *map-spmf f y*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*))))))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-eq-iff-list-all2*:
map f xs = *map g ys* ⟷ *list-all2* (λ*x y*. *f x* = *g y*) *xs ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

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

lemma *map2-minus-zero*:
$length\ xs = length\ ys \Longrightarrow (\bigwedge y{::}natL.\ y{\in}set\ ys \Longrightarrow y = 0) \Longrightarrow map2\ (-)\ xs\ ys = xs$
⟨*proof*⟩

lemma *permute-comp-left-inj*:
$p\ permutes\ \{..{<}n\} \Longrightarrow inj\ (\lambda p'.\ p \circ p')$
⟨*proof*⟩

lemma *permute-comp-left-inj-on*:
$p\ permutes\ \{..{<}n\} \Longrightarrow inj\text{-}on\ (\lambda p'.\ p \circ p')\ A$
⟨*proof*⟩

lemma *permute-comp-right-inj*:
$p\ permutes\ \{..{<}n\} \Longrightarrow inj\ (\lambda p'.\ p' \circ p)$
⟨*proof*⟩

lemma *permute-comp-right-inj-on*:
$p\ permutes\ \{..{<}n\} \Longrightarrow inj\text{-}on\ (\lambda p'.\ p' \circ p)\ A$
⟨*proof*⟩

lemma *permutes-inv-comp-left*:
$p\ permutes\ \{..{<}n\} \Longrightarrow inv\ (\lambda p'.\ p \circ p') = (\lambda p'.\ inv\ p \circ p')$
⟨*proof*⟩

lemma *permutes-inv-comp-right*:
$p\ permutes\ \{..{<}n\} \Longrightarrow inv\ (\lambda p'.\ p' \circ p) = (\lambda p'.\ p' \circ inv\ p)$
⟨*proof*⟩

lemma *permutes-inv-comp-left-right*:
$\pi a\ permutes\ \{..{<}n\} \Longrightarrow \pi b\ permutes\ \{..{<}n\} \Longrightarrow inv\ (\lambda p'.\ \pi a \circ p' \circ \pi b) = (\lambda p'.\ inv\ \pi a \circ p' \circ inv\ \pi b)$
⟨*proof*⟩

lemma *permutes-inv-comp-left-left*:
$\pi a\ permutes\ \{..{<}n\} \Longrightarrow \pi b\ permutes\ \{..{<}n\} \Longrightarrow inv\ (\lambda p'.\ \pi a \circ \pi b \circ p') = (\lambda p'.\ inv\ \pi b \circ inv\ \pi a \circ p')$
⟨*proof*⟩

lemma *permutes-inv-comp-right-right*:
$\pi a\ permutes\ \{..{<}n\} \Longrightarrow \pi b\ permutes\ \{..{<}n\} \Longrightarrow inv\ (\lambda p'.\ p' \circ \pi a \circ\ \pi b) = (\lambda p'.$

$p' \circ inv\ \pi b \circ inv\ \pi a$)
⟨*proof*⟩

lemma *image-compose-permutations-left-right*:
fixes S
assumes πa *permutes* S πb *permutes* S
shows $\{\pi a \circ \pi \circ \pi b \mid \pi.\ \pi \text{ permutes } S\} = \{\pi.\ \pi \text{ permutes } S\}$
⟨*proof*⟩

lemma *image-compose-permutations-left-left*:
fixes S
assumes πa *permutes* S πb *permutes* S
shows $\{\pi a \circ \pi b \circ \pi \mid \pi.\ \pi \text{ permutes } S\} = \{\pi.\ \pi \text{ permutes } S\}$
⟨*proof*⟩

lemma *image-compose-permutations-right-right*:
fixes S
assumes πa *permutes* S πb *permutes* S
shows $\{\pi \circ \pi a \circ \pi b \mid \pi.\ \pi \text{ permutes } S\} = \{\pi.\ \pi \text{ permutes } S\}$
⟨*proof*⟩

lemma *random-perm-middle*:
defines *random-perm* $n \equiv$ *spmf-of-set* $\{\pi.\ \pi \text{ permutes } \{..<n::nat\}\}$
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*⟩

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

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

⟨*proof*⟩

lemma *case-prod-return-spmf*:
case-prod (λ*a b*. *return-spmf* (*f a b*)) = (λ*x*. *return-spmf* (*case-prod f x*))
⟨*proof*⟩

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*⟩

lemma *reconstruct-stack-sharing-eq-reconstruct*:
reconstruct ∘ *aby3-stack-sharing r* = *reconstruct*
⟨*proof*⟩

lemma *map2-ignore1*:
length xs = *length ys* ⟹ *map2* (λ-. *f*) *xs ys* = *map f ys*
⟨*proof*⟩

lemma *map2-ignore2*:
length xs = *length ys* ⟹ *map2* (λ*a b*. *f a*) *xs ys* = *map f xs*
⟨*proof*⟩

lemma *map-sequence-share-nat-reconstruct*:
map-spmf (λ*x*. (*x*, *map reconstruct x*)) (*sequence-spmf* (*map share-nat y*)) = *map-spmf* (λ*x*. (*x*, *y*)) (*sequence-spmf* (*map share-nat y*))
⟨*proof*⟩

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*⟩

lemma *Collect-case-prod*:
{*f x y* | *x y*. *P x y*} = (*case-prod f*) ' (*Collect* (*case-prod P*))
⟨*proof*⟩

lemma *inj-split-Cons'*: *inj-on* (λ(*n*, *xs*). *n*#*xs*) *X*
⟨*proof*⟩

lemma *finite-indicator-eq-sum*:
finite A ⟹ *indicat-real A x* = *sum* (*indicat-real* {*x*}) *A*
⟨*proof*⟩

lemma *spmf-of-set-Cons*:
spmf-of-set (*set-Cons A B*) = *map2-spmf* (#) (*spmf-of-set A*) (*spmf-of-set B*)
⟨*proof*⟩

lemma *sequence-spmf-replicate*:
sequence-spmf (*replicate n* (*spmf-of-set A*)) = *spmf-of-set* (*listset* (*replicate n A*))
⟨*proof*⟩

lemma *listset-replicate*:
listset (*replicate n A*) = {*l*. *length l* = *n* ∧ *set l* ⊆ *A*}
⟨*proof*⟩

lemma *map2-map2-map3*:
map2 f (*map2 g x y*) *z* = *map3* (λ*x y*. *f* (*g x y*)) *x y z*
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *S1-def-simplified*:
$S1\ x1\ yc1 = (do\ \{$
$let\ n = length\ x1;$

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

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

⟨*proof*⟩

lemma *S2-def-simplified*:
$S2\ x2\ yc2 = (do\ \{$
$let\ n = length\ x2;$

$x3 \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$\pi a \leftarrow spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\};$
$\pi b \leftarrow spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\};$
$\zeta a2{::}natL\ list \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$\zeta b2{::}natL\ list \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$

$let\ msg2 = ((\zeta a2,\ \zeta b2,\ yc2),\ x3,\ \pi a,\ \pi b);$
$return\text{-}spmf\ msg2$
$\})$
⟨*proof*⟩

lemma *S3-def-simplified*:
$S3\ x3\ yc3 = (do\ \{$
$let\ n = length\ x3;$

$\pi b \leftarrow spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\};$
$\pi c \leftarrow spmf\text{-}of\text{-}set\ \{\pi.\ \pi\ permutes\ \{..<n\}\};$
$ya3{::}natL\ list \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$ya1{::}natL\ list \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$
$\zeta b3{::}natL\ list \leftarrow sequence\text{-}spmf\ (replicate\ n\ (spmf\text{-}of\text{-}set\ UNIV));$

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

⟨*proof*⟩

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

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

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

type-synonym *mul-in* = $natL \times natL$
type-synonym *mul-msg* = $(natL \times natL) \times natL$
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*;
$\zeta \leftarrow$ *zero-sharing*;
let z = *map-sharing2* (+) *z-raw* ζ;
let msg = *prod-sharing xy-shift* ζ;
return-spmf (*msg*, *z*)
})

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

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

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

⟨*proof*⟩

theorem *mul-spec*:
fixes *x-dist y-dist* :: *natL sharing spmf*
assumes *is-uniform-sharing x-dist is-uniform-sharing y-dist*
shows
secure:
(*do* {
xs ← *x-dist*;
ys ← *y-dist*;
let inps = *prod-sharing xs ys*;
(*msgs*, *outps*) ← (*aby3-mulR inps*);
let view = (*get-party p inps*, *get-party p msgs*);
return-spmf (*view*, *outps*)
})
=
(*do* {
xs ← *x-dist*;
ys ← *y-dist*;
let inps = *prod-sharing xs ys*;
outps ← *aby3-mulF inps*;
msg ← *S* (*get-party p inps*) (*get-party p outps*);
let view = (*get-party p inps*, *msg*);
return-spmf (*view*, *outps*)
}) (**is** *?secure*)
and
correct:
is-uniform-sharing (*do* {
xs ← *x-dist*;
ys ← *y-dist*;
aby3-mulF (*prod-sharing xs ys*)
}) (**is** *?uniform*)
⟨*proof*⟩

end
theory *Multiplication-Synthesization*
imports
Multiplication
begin

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

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

type-synonym (*'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*)
}
⟨*proof*⟩

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*}
⟨*proof*⟩

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

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

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

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*)))
⟨*proof*⟩

end

References

[1] P. Laud and M. Pettai. Secure multiparty sorting protocols with covert privacy. *Nordic Conference on Secure IT Systems*, pages 216–231, 2016.

[2] P. Mohassel and P. Rindal. Aby3: A mixed protocol framework for machine learning. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.