Multi-Party Computation

David Aspinall and David Butler

March 17, 2025

Abstract

We use CryptHOL [1, 6] to consider Multi-Party Computation (MPC) protocols. MPC was first considered in [8] and recent advances in efficiency and an increased demand mean it is now deployed in the real world. Security is considered using the real/ideal world paradigm. We first define security in the semi-honest security setting where parties are assumed not to deviate from the protocol transcript. In this setting we prove multiple Oblivious Transfer (OT) protocols secure and then show security for the gates of the GMW protocol [3]. We then define malicious security, this is a stronger notion of security where parties are assumed to be fully corrupted by an adversary. In this setting we again consider OT.

Contents

```
theory Cyclic-Group-Ext imports
  CryptHOL.CryptHOL
  HOL−Number-Theory.Cong
begin

context cyclic-group begin

lemma generator-pow-order: g [^] order G = 1
proof(cases order G > 0)
  case True
  hence fin: finite (carrier G) by(simp add: order-gt-0-iff-finite)
  then have [symmetric]: (λx. x ⊗ g) ‘ carrier G = carrier G
    by(rule endo-inj-surj)(auto simp add: inj-on-multc)
  then have carrier G = (λ n. g [^] Suc n) ‘ {..<order G}
    using fin by(simp add: carrier-conv-generator image-image)
  then obtain n where n: 1 = g [^] Suc n n < order G by auto
   have n = order G − 1 using n inj-onD[OF inj-on-generator, of 0 Suc n] by
fastforce
  with True n show ?thesis by auto
qed simp

lemma pow-generator-mod: g [^] (k mod order G) = g [^] k
proof(cases order G > 0)
  case True
 obtain n where n: k = n ∗ order G + k mod order G by (metis div-mult-mod-eq)
  have g [^] k = (g [^] order G) [^] n ⊗ g [^] (k mod order G)
    by(subst n)(simp add: nat-pow-mult nat-pow-pow mult-ac)
  then show ?thesis by(simp add: generator-pow-order)
qed simp

lemma int-nat-pow:
  assumes a ≥ 0
  shows (g [^] (int (a ::nat))) [^] (b::int)  = g [^] (a∗b)
  using assms
proof(cases a > 0)
  case True
  show ?thesis
    using int-pow-pow by blast
next case False
  have (g [^] (int (a ::nat))) [^] (b::int) = 1 using False by simp
  also have g [^] (a∗b) = 1 using False by simp
  ultimately show ?thesis by simp
qed

lemma pow-generator-mod-int: g [^] ((k :: int) mod order G) = g [^] k
proof(cases order G > 0)
  case True
  obtain n :: int where n: k = order G ∗ n + k mod order G
    by (metis div-mult-mod-eq mult.commute)
```

then have g [^] *k* = **g** [^] (*order G* ∗ *n*) ⊗ **g** [^] (*k mod order G*)
using *int-pow-mult nat-pow-mult* **by** (*metis generator-closed*)
then have g [^] *k* = (**g** [^] *order G*) [^] *n* ⊗ **g** [^] (*k mod order G*)
using *int-nat-pow* **by** (*simp add*: *int-pow-int*)
then show *?thesis* **by**(*simp add*: *generator-pow-order*)
qed *simp*

lemma *pow-gen-mod-mult*:
shows(**g** [^] (*a*::*nat*) ⊗ **g** [^] (*b*::*nat*)) [^] ((*c*::*int*)∗ *int* (*d*::*nat*)) = (**g** [^] *a* ⊗ **g** [^] *b*) [^] ((*c*∗*int d*) *mod* (*order G*))
proof−
have (**g** [^] (*a*::*nat*) ⊗ **g** [^] (*b*::*nat*)) ∈ *carrier G* **by** *simp*
then obtain *n* :: *nat* **where** *n*: **g** [^] *n* = (**g** [^] (*a*::*nat*) ⊗ **g** [^] (*b*::*nat*))
by (*simp add*: *monoid.nat-pow-mult*)
also obtain *r* **where** *r*: *r* = *c*∗*int d* **by** *simp*
have (**g** [^] (*a*::*nat*) ⊗ **g** [^] (*b*::*nat*)) [^] ((*c*::*int*)∗*int* (*d*::*nat*)) = (**g** [^] *n*) [^] *r* **using** *n r* **by** *simp*
moreover have... = (**g** [^] *n*) [^] (*r mod* (*order G*)) **using** *pow-generator-mod-int pow-generator-mod*
by (*metis int-nat-pow int-pow-int mod-mult-right-eq zero-le*)
moreover have ... = (**g** [^] *a* ⊗ **g** [^] *b*) [^] ((*c*∗*int d*) *mod* (*order G*)) **using** *r n* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *pow-generator-eq-iff-cong*:
finite (*carrier G*) ⟹ **g** [^] *x* = **g** [^] *y* ⟷ [*x* = *y*] (*mod order G*)
by(*subst* (*1 2*) *pow-generator-mod*[*symmetric*])(*auto simp add*: *cong-def order-gt-0-iff-finite intro*: *inj-onD*[*OF inj-on-generator*])

lemma *cyclic-group-commute*:
assumes *a* ∈ *carrier G b* ∈ *carrier G*
shows *a* ⊗ *b* = *b* ⊗ *a*
(**is** *?lhs* = *?rhs*)
proof−
obtain *n* :: *nat* **where** *n*: *a* = **g** [^] *n* **using** *generatorE assms* **by** *auto*
also obtain *k* :: *nat* **where** *k*: *b* = **g** [^] *k* **using** *generatorE assms* **by** *auto*
ultimately have *?lhs* = **g** [^] *n* ⊗ **g** [^] *k* **by** *simp*
then have ... = **g** [^] (*n* + *k*) **by**(*simp add*: *nat-pow-mult*)
then have ... = **g** [^] (*k* + *n*) **by**(*simp add*: *add.commute*)
then show *?thesis* **by**(*simp add*: *nat-pow-mult n k*)
qed

lemma *cyclic-group-assoc*:
assumes *a* ∈ *carrier G b* ∈ *carrier G c* ∈ *carrier G*
shows (*a* ⊗ *b*) ⊗ *c* = *a* ⊗ (*b* ⊗ *c*)
(**is** *?lhs* = *?rhs*)
proof−
obtain *n* :: *nat* **where** *n*: *a* = **g** [^] *n* **using** *generatorE assms* **by** *auto*

```
  obtain k :: nat where k: b = g [^] k using generatorE assms by auto
  obtain j :: nat where j: c = g [^] j using generatorE assms by auto
  have ?lhs = (g [^] n ⊗ g [^] k) ⊗ g [^] j using n k j by simp
  then have ... = g [^] (n + (k + j)) by(simp add: nat-pow-mult add.assoc)
  then show ?thesis by(simp add: nat-pow-mult n k j)
qed

lemma l-cancel-inv:
  assumes h ∈ carrier G
  shows (g [^] (a :: nat) ⊗ inv (g [^] a)) ⊗ h = h
(is ?lhs = ?rhs)
proof−
  have ?lhs = (g [^] int a ⊗ inv (g [^] int a)) ⊗ h by simp
  then have ... = (g [^] int a ⊗ (g [^] (− a))) ⊗ h using int-pow-neg[symmetric]
by simp
  then have ... = g [^] (int a − a)  ⊗ h by(simp add: int-pow-mult)
  then have ... = g [^] ((0:: int)) ⊗ h by simp
  then show ?thesis by (simp add: assms)
qed

lemma inverse-split:
  assumes a ∈ carrier G and b ∈ carrier G
  shows inv (a ⊗ b) = inv a ⊗ inv b
 by (simp add:  assms comm-group.inv-mult cyclic-group-commute group-comm-groupI)

lemma inverse-pow-pow:
  assumes a ∈ carrier G
  shows inv (a [^] (r::nat)) = (inv a) [^] r
proof −
  have a [^] r ∈ carrier G
    using assms by blast
  then show ?thesis
    by (simp add: assms nat-pow-inv)
qed

lemma l-neq-1-exp-neq-0:
  assumes l ∈ carrier G
    and l ≠ 1
    and l = g [^] (t::nat)
  shows t ≠ 0
proof(rule ccontr)
  assume ¬ (t ≠ 0)
  hence t = 0 by simp
  hence g [^] t = 1 by simp
  then show False using assms by simp
qed

lemma order-gt-1-gen-not-1:
  assumes order G > 1
```

shows g ≠ **1**
proof(*rule ccontr*)
assume ¬ **g** ≠ **1**
hence g = **1 by** *simp*
hence *g-pow-eq-1*: **g** [^] *n* = **1 for** *n* :: *nat* **by** *simp*
hence *range* (λ*n* :: *nat*. **g** [^] *n*) = {**1**} **by** *auto*
hence *carrier G* ⊆ {**1**} **using** *generator* **by** *auto*
hence *order G* < *1*
by (*metis One-nat-def assms g-pow-eq-1 inj-onD inj-on-generator lessThan-iff not-gr-zero zero-less-Suc*)
with *assms* **show** *False* **by** *simp*
qed

lemma *power-swap*: ((**g** [^] (α*0*::*nat*)) [^] (*r*::*nat*)) = ((**g** [^] *r*) [^] α*0*)
(**is** *?lhs* = *?rhs*)
proof−
have *?lhs* = **g** [^] (α*0* ∗ *r*)
using *nat-pow-pow mult.commute* **by** *auto*
hence ... = **g** [^] (*r* ∗ α*0*)
by(*metis mult.commute*)
thus *?thesis* **using** *nat-pow-pow* **by** *auto*
qed

end

end
theory *Number-Theory-Aux* **imports**
HOL−*Number-Theory.Cong*
HOL−*Number-Theory.Residues*
begin

lemma *bezw-inverse*:
assumes *gcd* (*e* :: *nat*) (*N* ::*nat*) = *1*
shows [*nat e* ∗ *nat* ((*fst* (*bezw e N*)) *mod N*) = *1*] (*mod nat N*)
proof−
have (*fst* (*bezw e N*) ∗ *e* + *snd* (*bezw e N*) ∗ *N*) *mod N* = *1 mod N*
by (*metis assms bezw-aux zmod-int*)
hence (*fst* (*bezw e N*) *mod N* ∗ *e mod N*) = *1 mod N*
by (*simp add*: *mod-mult-right-eq mult.commute*)
hence *cong-eq*: [(*fst* (*bezw e N*) *mod N* ∗ *e*) = *1*] (*mod N*)
by (*metis of-nat-1 zmod-int cong-def*)
hence [*nat* (*fst* (*bezw e N*) *mod N*) ∗ *e* = *1*] (*mod N*)
proof −
{ assume *int* (*nat* (*fst* (*bezw e N*) *mod int N*)) ≠ *fst* (*bezw e N*) *mod int N*
have *N* = *0* ⟶ *0* ≤ *fst* (*bezw e N*) *mod int N*
by *fastforce*
then have *int* (*nat* (*fst* (*bezw e N*) *mod int N*)) = *fst* (*bezw e N*) *mod int N*
by *fastforce* **}**
then have [*int* (*nat* (*fst* (*bezw e N*) *mod int N*) ∗ *e*) = *int 1*] (*mod int N*)

by (*metis cong-eq of-nat-1 of-nat-mult*)
then show *?thesis*
using *cong-int-iff* **by** *blast*
qed
then show *?thesis* **by**(*simp add*: *mult.commute*)
qed

lemma *inverse*:
assumes $gcd\ x\ (q{::}nat) = 1$
and $q > 0$
shows $[x * (fst\ (bezw\ x\ q)) = 1]\ (mod\ q)$
proof−
have *int-eq*: $fst\ (bezw\ \ x\ q) * x + snd\ (bezw\ x\ q) * int\ q = 1$
by (*metis assms*(*1*) *bezw-aux of-nat-1*)
hence *int-eq′*: $(fst\ (bezw\ \ x\ q) * x + snd\ (bezw\ x\ q) * int\ q)\ mod\ q = 1\ mod\ q$
by (*metis of-nat-1 zmod-int*)
hence $(fst\ (bezw\ x\ q) * x)\ mod\ q = 1\ mod\ q$
by *simp*
hence $[(fst\ (bezw\ x\ q)) * x\ = 1]\ (mod\ q)$
using *cong-def int-eq int-eq′* **by** *metis*
then show *?thesis* **by**(*simp add*: *mult.commute*)
qed

lemma *prod-not-prime*:
assumes $prime\ (x{::}nat)$
and $prime\ y$
and $x > 2$
and $y > 2$
shows $\neg\ prime\ ((x-1)*(y-1))$
by (*metis assms One-nat-def Suc-diff-1 nat-neq-iff numeral-2-eq-2 prime-gt-0-nat prime-product*)

lemma *ex-inverse*:
assumes *coprime*: $coprime\ (e :: nat)\ ((P-1)*(Q-1))$
and $prime\ P$
and $prime\ Q$
and $P \neq Q$
shows $\exists\ d.\ [e*d = 1]\ (mod\ (P-1)) \wedge d \neq 0$
proof−
have $coprime\ e\ (P-1)$
using *assms*(*1*) **by** *simp*
then obtain *d* **where** *d*: $[e*d = 1]\ (mod\ (P-1))$
using *cong-solve-coprime-nat* **by** *auto*
then show *?thesis* **by** (*metis cong-0-1-nat cong-1 mult-0-right zero-neq-one*)
qed

lemma *ex-k1-k2*:
assumes *coprime*: $coprime\ (e :: nat)\ ((P-1)*(Q-1))$
and $[e*d = 1]\ (mod\ (P-1))$

```
  shows ∃ k1 k2. e*d + k1*(P−1) = 1 + k2*(P−1)
  by (metis assms(2) cong-iff-lin-nat)

lemma ex-k-mod:
  assumes coprime: coprime (e :: nat) ((P−1)*(Q−1))
    and P ≠ Q
    and prime P
    and prime Q
    and d ≠ 0
    and  [e*d = 1] (mod (P−1))
  shows ∃ k. e*d = 1 + k*(P−1)
proof−
  have e > 0
    using assms(1) assms(2) prime-gt-0-nat by fastforce
  then have e*d ≥ 1 using assms by simp
  then obtain k where k: e*d = 1 + k*(P−1)
    using assms(6) cong-to-1'-nat by auto
  then show ?thesis
    by simp
qed

lemma fermat-little:
  assumes prime (P :: nat)
  shows [x^P = x] (mod P)
proof(cases P dvd x)
  case True
  hence x mod P = 0 by simp
  moreover have x ^ P mod P = 0
    by (simp add: True assms prime-dvd-power-nat-iff prime-gt-0-nat)
  ultimately show ?thesis
    by (simp add: cong-def)
next
  case False
  hence [x ^(P − 1) = 1] (mod P)
    using fermat-theorem assms by blast
  then show ?thesis
   by (metis assms cong-def diff-diff-cancel diff-is-0-eq' diff-zero mod-mult-right-eq
power-eq-if power-one-right prime-ge-1-nat zero-le-one)
qed

end
```

1 Uniform Sampling

Here we prove different one time pad lemmas based on uniform sampling we require throughout our proofs.

```
theory Uniform-Sampling
  imports
```

CryptHOL.Cyclic-Group-SPMF
HOL−Number-Theory.Cong
CryptHOL.List-Bits
begin

If q is a prime we can sample from the units.

definition *sample-uniform-units* :: *nat* ⇒ *nat spmf*
where *sample-uniform-units q* = *spmf-of-set* ({..< *q*} − {*0*})

lemma *set-spmf-sampl-uni-units* [*simp*]: *set-spmf* (*sample-uniform-units q*) = {..< *q*} − {*0*}
by(*simp add*: *sample-uniform-units-def*)

lemma *lossless-sample-uniform-units*:
assumes *q* > *1*
shows *lossless-spmf* (*sample-uniform-units q*)
apply(*simp add*: *sample-uniform-units-def*)
using *assms* **by** *auto*

General lemma for mapping using uniform sampling from units.

lemma *one-time-pad-units*:
assumes *inj-on*: *inj-on f* ({..<*q*} − {*0*})
and *sur*: *f* ‘ ({..<*q*} − {*0*}) = ({..<*q*} − {*0*})
shows *map-spmf f* (*sample-uniform-units q*) = (*sample-uniform-units q*)
(**is** *?lhs* = *?rhs*)
proof−
have *rhs*: *?rhs* = *spmf-of-set* (({..<*q*} − {*0*}))
by(*auto simp add*: *sample-uniform-units-def*)
also have *map-spmf*(λ*s*. *f s*) (*spmf-of-set* ({..<*q*} − {*0*})) = *spmf-of-set* ((λ*s*. *f s*) ‘ ({..<*q*} − {*0*}))
by(*simp add*: *inj-on*)
also have *f* ‘ ({..<*q*} − {*0*}) = ({..<*q*} − {*0*})
apply(*rule endo-inj-surj*) **by**(*simp*, *simp add*: *sur*, *simp add*: *inj-on*)
ultimately show *?thesis* **using** *rhs* **by** *simp*
qed

General lemma for mapping using uniform sampling.

lemma *one-time-pad*:
assumes *inj-on*: *inj-on f* {..<*q*}
and *sur*: *f* ‘ {..<*q*} = {..<*q*}
shows *map-spmf f* (*sample-uniform q*) = (*sample-uniform q*)
(**is** *?lhs* = *?rhs*)
proof−
have *rhs*: *?rhs* = *spmf-of-set* ({..< *q*})
by(*auto simp add*: *sample-uniform-def*)
also have *map-spmf*(λ*s*. *f s*) (*spmf-of-set* {..<*q*}) = *spmf-of-set* ((λ*s*. *f s*) ‘ {..<*q*})
by(*simp add*: *inj-on*)
also have *f* ‘ {..<*q*} = {..<*q*}

```
    apply(rule endo-inj-surj) by(simp, simp add: sur, simp add: inj-on)
  ultimately show ?thesis using rhs by simp
qed
```

The addition map case.

```
lemma inj-add:
  assumes x:  x < q
    and x': x' < q
    and map: ((y :: nat) + x) mod q = (y + x') mod q
  shows x = x'
proof−
  have aa: ((y :: nat) + x) mod q = (y + x') mod q ⟹ x mod q = x' mod q
  proof−
    have 4: ((y:: nat) + x) mod q = (y + x') mod q ⟹ [((y:: nat) + x) = (y +
x')] (mod q)
      by(simp add: cong-def)
    have 5: [((y:: nat) + x) = (y + x')] (mod q) ⟹ [x = x'] (mod q)
      by (simp add: cong-add-lcancel-nat)
    have 6: [x = x'] (mod q) ⟹ x mod q = x' mod q
      by(simp add: cong-def)
    then show ?thesis by(simp add: map 4 5 6)
  qed
  also have bb: x mod q = x' mod q ⟹ x = x'
    by(simp add: x x')
  ultimately show ?thesis by(simp add: map)
qed

lemma inj-uni-samp-add: inj-on (λ(b :: nat). (y + b) mod q ) {..<q}
  by(simp add: inj-on-def)(auto simp only: inj-add)

lemma surj-uni-samp:
  assumes inj: inj-on  (λ(b :: nat). (y + b) mod q ) {..<q}
  shows (λ(b :: nat). (y + b) mod q) ‘ {..< q} =  {..< q}
  apply(rule endo-inj-surj) using inj by auto

lemma samp-uni-plus-one-time-pad:
  shows map-spmf (λb. (y + b) mod q) (sample-uniform q) = (sample-uniform q)
  using inj-uni-samp-add surj-uni-samp one-time-pad by simp
```

The multiplicaton map case.

```
lemma inj-mult:
  assumes coprime: coprime x (q::nat)
    and y: y < q
    and y': y' < q
  and map: x ∗ y mod q = x ∗ y' mod q
shows y = y'
proof−
  have x∗y mod q = x∗y' mod q ⟹ y mod q = y' mod q
  proof−
```

```
    have x∗y mod q = x∗y′ mod q ⟹ [x∗y = x∗y′] (mod q)
      by(simp add: cong-def)
    also have [x∗y = x∗y′] (mod q) = [y = y′] (mod q)
      by(simp add: cong-mult-lcancel-nat coprime)
    also have [y = y′] (mod q) ⟹ y mod q = y′ mod q
      by(simp add: cong-def)
    ultimately show ?thesis by(simp add: map)
  qed
  also have y mod q = y′ mod q ⟹ y = y′
    by(simp add: y y′)
  ultimately show ?thesis by(simp add: map)
qed

lemma inj-on-mult:
  assumes coprime: coprime x (q::nat)
  shows inj-on (λ b. x∗b mod q) {..<q}
  apply(auto simp add: inj-on-def)
  using coprime by(simp only: inj-mult)

lemma surj-on-mult:
  assumes coprime: coprime x (q::nat)
    and inj: inj-on (λ b. x∗b mod q) {..<q}
  shows (λ b. x∗b mod q) ‘ {..< q} = {..< q}
  apply(rule endo-inj-surj) using coprime inj by auto

lemma mult-one-time-pad:
  assumes coprime: coprime x q
  shows map-spmf (λ b. x∗b mod q) (sample-uniform q) = (sample-uniform q)
  using inj-on-mult surj-on-mult one-time-pad coprime by simp
```

The multiplication map for sampling from units.

```
lemma inj-on-mult-units:
  assumes 1: coprime x (q::nat) shows inj-on (λ b. x∗b mod q) ({..<q} − {0})
  apply(auto simp add: inj-on-def)
  using 1 by(simp only: inj-mult)

lemma surj-on-mult-units:
  assumes coprime: coprime x (q::nat)
    and inj: inj-on (λ b. x∗b mod q) ({..<q} − {0})
  shows (λ b. x∗b mod q) ‘ ({..<q} − {0}) = ({..<q} − {0})
proof(rule endo-inj-surj)
  show finite ({..<q} − {0}) using coprime inj by(simp)
  show (λb. x ∗ b mod q) ‘ ({..<q} − {0}) ⊆ {..<q} − {0}
  proof −
    obtain n :: nat set ⇒ (nat ⇒ nat) ⇒ nat set ⇒ nat where
      ∀ x0 x1 x2. (∃ v3. v3 ∈ x2 ∧ x1 v3 ∉ x0) = (n x0 x1 x2 ∈ x2 ∧ x1 (n x0 x1
x2) ∉ x0)
      by moura
    then have subset: ∀ N f Na. n Na f N ∈ N ∧ f (n Na f N) ∉ Na ∨ f ‘ N ⊆ Na
```

by (*meson image-subsetI*)
have *mem-insert*: *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∉ {..<*q*} ∨ *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∈ *insert 0* {..<*q*}
by *force*
have *map-eq*: (*x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∈ *insert 0* {..<*q*} − {*0*}) = (*x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∈ {..<*q*} − {*0*})
by *simp*
{ assume *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* = *x* ∗ *0 mod q*
then have (*0* ≤ *q*) = (*0* = *q*) ∨ (*n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) ∉ {..<*q*} ∨ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) ∈ {*0*}) ∨ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) ∉ {..<*q*} − {*0*} ∨ *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∈ {..<*q*} − {*0*}
by (*metis antisym-conv1 insertCI lessThan-iff local.coprime inj-mult*) **}**
moreover
{ assume *0* ≠ *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q*
moreover
{ assume *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∈ *insert 0* {..<*q*} ∧ *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∉ {*0*}
then have (λ*n*. *x* ∗ *n mod q*) ‘ ({..<*q*} − {*0*}) ⊆ {..<*q*} − {*0*}
using *map-eq subset* **by** (*meson Diff-iff*) **}**
ultimately have (λ*n*. *x* ∗ *n mod q*) ‘ ({..<*q*} − {*0*}) ⊆ {..<*q*} − {*0*} ∨ (*0* ≤ *q*) = (*0* = *q*)
using *mem-insert* **by** (*metis antisym-conv1 lessThan-iff mod-less-divisor singletonD*) **}**
ultimately have (λ*n*. *x* ∗ *n mod q*) ‘ ({..<*q*} − {*0*}) ⊆ {..<*q*} − {*0*} ∨ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) ∉ {..<*q*} − {*0*} ∨ *x* ∗ *n* ({..<*q*} − {*0*}) (λ*n*. *x* ∗ *n mod q*) ({..<*q*} − {*0*}) *mod q* ∈ {..<*q*} − {*0*}
by *force*
then show (λ*n*. *x* ∗ *n mod q*) ‘ ({..<*q*} − {*0*}) ⊆ {..<*q*} − {*0*}
using *subset* **by** *meson*
qed
show *inj-on* (λ*b*. *x* ∗ *b mod q*) ({..<*q*} − {*0*}) **using** *assms* **by**(*simp*)
qed

lemma *mult-one-time-pad-units*:
assumes *coprime*: *coprime x q*
shows *map-spmf* (λ *b*. *x*∗*b mod q*) (*sample-uniform-units q*) = *sample-uniform-units q*
using *inj-on-mult-units surj-on-mult-units one-time-pad-units coprime* **by** *simp*

Addition and multiplication map.

lemma *samp-uni-add-mult*:
assumes *coprime*: *coprime x* (*q*::*nat*)
and *xa*: *xa* < *q*
and *ya*: *ya* < *q*

and *map*: $(y + x * xa)$ *mod* $q = (y + x * ya)$ *mod* q
shows $xa = ya$
proof –
have $(y + x * xa)$ *mod* $q = (y + x * ya)$ *mod* $q \Longrightarrow xa$ *mod* $q = ya$ *mod* q
proof –
have $(y + x * xa)$ *mod* $q = (y + x * ya)$ *mod* $q \Longrightarrow [y + x*xa = y + x *ya]$ (*mod* q)
using *cong-def* **by** *blast*
also have $[y + x*xa = y + x *ya]$ (*mod* q) $\Longrightarrow [xa = ya]$ (*mod* q)
by(*simp add*: *cong-add-lcancel-nat*)(*simp add*: *coprime cong-mult-lcancel-nat*)
ultimately show *?thesis* **by**(*simp add*: *cong-def map*)
qed
also have xa *mod* $q = ya$ *mod* $q \Longrightarrow xa = ya$
by(*simp add*: *xa ya*)
ultimately show *?thesis* **by**(*simp add*: *map*)
qed

lemma *inj-on-add-mult*:
assumes *coprime*: *coprime* x (q::*nat*)
shows *inj-on* (λ b. $(y + x*b)$ *mod* q) {..<q}
apply(*auto simp add*: *inj-on-def*)
using *coprime* **by**(*simp only*: *samp-uni-add-mult*)

lemma *surj-on-add-mult*: **assumes** *coprime*: *coprime* x (q::*nat*) **and** *inj*: *inj-on* (λ b. $(y + x*b)$ *mod* q) {..<q}
shows (λ b. $(y + x*b)$ *mod* q) ‘ {..< q} = {..< q}
apply(*rule endo-inj-surj*) **using** *coprime inj* **by** *auto*

lemma *add-mult-one-time-pad*: **assumes** *coprime*: *coprime* x q
shows *map-spmf* (λ b. $(y + x*b)$ *mod* q) (*sample-uniform* q) = (*sample-uniform* q)
using *inj-on-add-mult surj-on-add-mult one-time-pad coprime* **by** *simp*

Subtraction Map.

lemma *inj-minus*:
assumes x: $(x :: nat) < q$
and *ya*: $ya < q$
and *map*: $(y + q - x)$ *mod* $q = (y + q - ya)$ *mod* q
shows $x = ya$
proof –
have $(y + q - x)$ *mod* $q = (y + q - ya)$ *mod* $q \Longrightarrow x$ *mod* $q = ya$ *mod* q
proof –
have $(y + q - x)$ *mod* $q = (y + q - ya)$ *mod* $q \Longrightarrow [y + q - x = y + q - ya]$ (*mod* q)
using *cong-def* **by** *blast*
moreover have $[y + q - x = y + q - ya]$ (*mod* q) $\Longrightarrow [q - x = q - ya]$ (*mod* q)
using *x ya cong-add-lcancel-nat* **by** *fastforce*
moreover have $[y + q - x = y + q - ya]$ (*mod* q) $\Longrightarrow [q + x = q + ya]$

(*mod q*)
 by (*metis add-diff-inverse-nat calculation*(*2*) *cong-add-lcancel-nat cong-add-rcancel-nat cong-sym less-imp-le-nat not-le x ya*)
 ultimately show *?thesis*
 by (*simp add*: *cong-def map*)
 qed
 moreover have *x mod q* = *ya mod q* ⟹ *x* = *ya*
 by(*simp add*: *x ya*)
 ultimately show *?thesis* **by**(*simp add*: *map*)
qed

lemma *inj-on-minus*: *inj-on* (λ(*b* :: *nat*). (*y* + (*q* − *b*)) *mod q*) {..<*q*}
 by(*auto simp add*: *inj-on-def inj-minus*)

lemma *surj-on-minus*:
 assumes *inj*: *inj-on* (λ(*b* :: *nat*). (*y* + (*q* − *b*)) *mod q*) {..<*q*}
 shows (λ(*b* :: *nat*). (*y* + (*q* − *b*)) *mod q*) ' {..< *q*} = {..< *q*}
 apply(*rule endo-inj-surj*)
 using *inj* **by** *auto*

lemma *samp-uni-minus-one-time-pad*:
 shows *map-spmf*(λ *b*. (*y* + (*q* − *b*)) *mod q*) (*sample-uniform q*) = (*sample-uniform q*)
 using *inj-on-minus surj-on-minus one-time-pad* **by** *simp*

lemma *not-coin-flip*: *map-spmf* (λ *a*. ¬ *a*) *coin-spmf* = *coin-spmf*
proof−
 have *inj-on Not* {*True*, *False*}
 by *simp*
 also have *Not* ' {*True*, *False*} = {*True*, *False*}
 by *auto*
 ultimately show *?thesis* **using** *one-time-pad*
 by (*simp add*: *UNIV-bool*)
qed

lemma *xor-uni-samp*: *map-spmf*(λ *b*. *y* ⊕ *b*) (*coin-spmf*) = *map-spmf*(λ *b*. *b*) (*coin-spmf*)
 (**is** *?lhs* = *?rhs*)
proof−
 have *rhs*: *?rhs* = *spmf-of-set* {*True*, *False*}
 by (*simp add*: *UNIV-bool insert-commute*)
 also have *map-spmf*(λ *b*. *y* ⊕ *b*) (*spmf-of-set* {*True*, *False*}) = *spmf-of-set*((λ *b*. *y* ⊕ *b*) ' {*True*, *False*})
 by (*simp add*: *xor-def*)
 also have (λ *b*. *xor y b*) ' {*True*, *False*} = {*True*, *False*}
 using *xor-def* **by** *auto*
 finally show *?thesis* **using** *rhs* **by**(*simp*)
qed

end

2 Semi-Honest Security

We follow the security definitions for the semi honest setting as described in [5]. In the semi honest model the parties are assumed not to deviate from the protocol transcript. Semi honest security guarantees that no information is leaked during the running of the protocol.

2.1 Security definitions

theory *Semi-Honest-Def* **imports**
  CryptHOL.CryptHOL
begin

2.1.1 Security for deterministic functionalities

locale *sim-det-def* =
  fixes *R1* :: *'msg1* ⇒ *'msg2* ⇒ *'view1 spmf*
    and *S1* :: *'msg1* ⇒ *'out1* ⇒ *'view1 spmf*
    and *R2* :: *'msg1* ⇒ *'msg2* ⇒ *'view2 spmf*
    and *S2* :: *'msg2* ⇒ *'out2* ⇒ *'view2 spmf*
    and *funct* :: *'msg1* ⇒ *'msg2* ⇒ (*'out1* × *'out2*) *spmf*
    and *protocol* :: *'msg1* ⇒ *'msg2* ⇒ (*'out1* × *'out2*) *spmf*
  assumes *lossless-R1*: *lossless-spmf* (*R1 m1 m2*)
    and *lossless-S1*: *lossless-spmf* (*S1 m1 out1*)
    and *lossless-R2*: *lossless-spmf* (*R2 m1 m2*)
    and *lossless-S2*: *lossless-spmf* (*S2 m2 out2*)
    and *lossless-funct*: *lossless-spmf* (*funct m1 m2*)
begin

type-synonym *'view' adversary-det* = *'view'* ⇒ *bool spmf*

definition *correctness m1 m2* ≡ (*protocol m1 m2* = *funct m1 m2*)

definition *adv-P1* :: *'msg1* ⇒ *'msg2* ⇒ *'view1 adversary-det* ⇒ *real*
  where *adv-P1 m1 m2 D* ≡ |(*spmf* (*R1 m1 m2* ⤜ *D*) *True*)
          − *spmf* (*funct m1 m2* ⤜ (λ (*o1*, *o2*). *S1 m1 o1* ⤜ *D*)) *True*|

definition *perfect-sec-P1 m1 m2* ≡ (*R1 m1 m2* = *funct m1 m2* ⤜ (λ (*s1*, *s2*). *S1 m1 s1*))

definition *adv-P2* :: *'msg1* ⇒ *'msg2* ⇒ *'view2 adversary-det* ⇒ *real*
  where *adv-P2 m1 m2 D* = |*spmf* (*R2 m1 m2* ⤜ (λ *view*. *D view*)) *True*
          − *spmf* (*funct m1 m2* ⤜ (λ (*o1*, *o2*). *S2 m2 o2* ⤜ (λ *view*. *D view*))) *True*|

definition *perfect-sec-P2 m1 m2* $\equiv$ (*R2 m1 m2* = *funct m1 m2* $\gg=$ (λ (*s1*, *s2*). *S2 m2 s2*))

We also define the security games (for Party 1 and 2) used in EasyCrypt to define semi honest security for Party 1. We then show the two definitions are equivalent.

definition *P1-game-alt* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ *'view1 adversary-det* $\Rightarrow$ *bool spmf*
  where *P1-game-alt m1 m2 D* = *do* {
    b $\leftarrow$ *coin-spmf*;
    (*out1*, *out2*) $\leftarrow$ *funct m1 m2*;
    rview :: *'view1* $\leftarrow$ *R1 m1 m2*;
    sview :: *'view1* $\leftarrow$ *S1 m1 out1*;
    b' $\leftarrow$ *D* (*if b then rview else sview*);
    return-spmf (*b* = *b'*)}

definition *adv-P1-game* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ *'view1 adversary-det* $\Rightarrow$ *real*
  where *adv-P1-game m1 m2 D* = |*2**(*spmf* (*P1-game-alt m1 m2 D*) *True*) − *1*|

We show the two definitions are equivalent

lemma *equiv-defs-P1*:
  assumes *lossless-D*: $\forall$ *view*. *lossless-spmf* ((*D*:: *'view1 adversary-det*) *view*)
  shows *adv-P1-game m1 m2 D* = *adv-P1 m1 m2 D*
  including *monad-normalisation*
proof−
  have *return-True-not-False*: *spmf* (*return-spmf* (*b*)) *True* = *spmf* (*return-spmf* (¬ *b*)) *False*
    for *b* **by**(*cases b*; *auto*)
  have *lossless-ideal*: *lossless-spmf* ((*funct m1 m2* $\gg=$ (λ(*out1*, *out2*). *S1 m1 out1* $\gg=$ (λ*sview*. *D sview* $\gg=$ (λ*b'*. *return-spmf* (*False* = *b'*))))))
    by(*simp add*: *lossless-S1 lossless-funct lossless-weight-spmfD split-def lossless-D*)
  have *return*: *spmf* (*funct m1 m2* $\gg=$ (λ(*o1*, *o2*). *S1 m1 o1* $\gg=$ *D*)) *True*
          = *spmf* (*funct m1 m2* $\gg=$ (λ(*o1*, *o2*). *S1 m1 o1* $\gg=$ (λ *view*. *D view* $\gg=$ (λ *b*. *return-spmf b*)))) *True*
    by *simp*
  have *2**(*spmf* (*P1-game-alt m1 m2 D*) *True*) − *1* = (*spmf* (*R1 m1 m2* $\gg=$ (λ*rview*. *D rview* $\gg=$ (λ(*b'*:: *bool*). *return-spmf* (*True* = *b'*))))) *True*
      − (*1* − (*spmf* (*funct m1 m2* $\gg=$ (λ(*out1*, *out2*). *S1 m1 out1* $\gg=$ (λ*sview*. *D sview* $\gg=$ (λ*b'*. *return-spmf* (*False* = *b'*)))))) *True*)
    by(*simp add*: *spmf-bind integral-spmf-of-set adv-P1-game-def P1-game-alt-def spmf-of-set*
        UNIV-bool bind-spmf-const lossless-R1 lossless-S1 lossless-funct lossless-weight-spmfD)
  hence *adv-P1-game m1 m2 D* = |(*spmf* (*R1 m1 m2* $\gg=$ (λ*rview*. *D rview* $\gg=$ (λ(*b'*:: *bool*). *return-spmf* (*True* = *b'*))))) *True*
      − (*1* − (*spmf* (*funct m1 m2* $\gg=$ (λ(*out1*, *out2*). *S1 m1 out1* $\gg=$ (λ*sview*. *D sview* $\gg=$ (λ*b'*. *return-spmf* (*False* = *b'*)))))) *True*)|
    using *adv-P1-game-def* **by** *simp*
  also have |(*spmf* (*R1 m1 m2* $\gg=$ (λ*rview*. *D rview* $\gg=$ (λ(*b'*:: *bool*). *return-spmf* (*True* = *b'*))))) *True*

− (1 − (spmf (funct m1 m2 ⤜ (λ(out1, out2). S1 m1 out1 ⤜ (λsview. D sview
⤜ (λb′. return-spmf (False = b′)))))) True)| = adv-P1 m1 m2 D
apply(*simp only*: *adv-P1-def spmf-False-conv-True*[*symmetric*] *lossless-ideal*; *simp*)
by(*simp only*: *return*)(*simp only*: *split-def spmf-bind return-True-not-False*)
ultimately show *?thesis* **by** *simp*
qed

definition *P2-game-alt* :: ′*msg1* ⇒ ′*msg2* ⇒ ′*view2 adversary-det* ⇒ *bool spmf*
where *P2-game-alt m1 m2 D* = *do* {
b ← *coin-spmf*;
(*out1*, *out2*) ← *funct m1 m2*;
rview :: ′*view2* ← *R2 m1 m2*;
sview :: ′*view2* ← *S2 m2 out2*;
b′ ← *D* (*if b then rview else sview*);
return-spmf (*b* = *b*′)}

definition *adv-P2-game* :: ′*msg1* ⇒ ′*msg2* ⇒ ′*view2 adversary-det* ⇒ *real*
where *adv-P2-game m1 m2 D* = |2∗(*spmf* (*P2-game-alt m1 m2 D*) *True*) − 1|

lemma *equiv-defs-P2*:
assumes *lossless-D*: ∀ *view*. *lossless-spmf* ((*D*:: ′*view2 adversary-det*) *view*)
shows *adv-P2-game m1 m2 D* = *adv-P2 m1 m2 D*
including *monad-normalisation*
proof−
have *return-True-not-False*: *spmf* (*return-spmf* (*b*)) *True* = *spmf* (*return-spmf* (¬ *b*)) *False*
for *b* **by**(*cases b*; *auto*)
have *lossless-ideal*: *lossless-spmf* ((*funct m1 m2* ⤜ (λ(*out1*, *out2*). *S2 m2 out2* ⤜ (λ*sview*. *D sview* ⤜ (λ*b*′. *return-spmf* (*False* = *b*′))))))
by(*simp add*: *lossless-S2 lossless-funct lossless-weight-spmfD split-def lossless-D*)
have *return*: *spmf* (*funct m1 m2* ⤜ (λ(*o1*, *o2*). *S2 m2 o2* ⤜ *D*)) *True* = *spmf* (*funct m1 m2* ⤜ (λ(*o1*, *o2*). *S2 m2 o2* ⤜ (λ *view*. *D view* ⤜ (λ *b*. *return-spmf b*)))) *True*
by *simp*
have
2∗(*spmf* (*P2-game-alt m1 m2 D*) *True*) − 1 = (*spmf* (*R2 m1 m2* ⤜ (λ*rview*. *D rview* ⤜ (λ(*b*′:: *bool*). *return-spmf* (*True* = *b*′))))) *True*
− (1 − (*spmf* (*funct m1 m2* ⤜ (λ(*out1*, *out2*). *S2 m2 out2* ⤜ (λ*sview*. *D sview* ⤜ (λ*b*′. *return-spmf* (*False* = *b*′)))))) *True*)
by(*simp add*: *spmf-bind integral-spmf-of-set adv-P1-game-def P2-game-alt-def spmf-of-set*
UNIV-bool bind-spmf-const lossless-R2 lossless-S2 lossless-funct lossless-weight-spmfD)
hence *adv-P2-game m1 m2 D* = |(*spmf* (*R2 m1 m2* ⤜ (λ*rview*. *D rview* ⤜ (λ(*b*′:: *bool*). *return-spmf* (*True* = *b*′))))) *True*

$-$ (*1* $-$ (*spmf* (*funct m1 m2* $\gg=$ (λ(*out1*, *out2*). *S2 m2 out2* $\gg=$ (λ*sview*. *D sview* $\gg=$ ($\lambda b'$. *return-spmf* (*False* = b')))))) *True*)|
using *adv-P2-game-def* **by** *simp*
also have |(*spmf* (*R2 m1 m2* $\gg=$ (λ*rview*. *D rview* $\gg=$ (λ(b':: *bool*). *return-spmf* (*True* = b'))))) *True*
$-$ (*1* $-$ (*spmf* (*funct m1 m2* $\gg=$ (λ(*out1*, *out2*). *S2 m2 out2* $\gg=$ (λ*sview*. *D sview* $\gg=$ ($\lambda b'$. *return-spmf* (*False* = b')))))) *True*)| = *adv-P2 m1 m2 D*
apply(*simp only*: *adv-P2-def spmf-False-conv-True*[*symmetric*] *lossless-ideal*; *simp*)
by(*simp only*: *return*)(*simp only*: *split-def spmf-bind return-True-not-False*)
ultimately show *?thesis* **by** *simp*
qed

end

2.1.2 Security definitions for non deterministic functionalities

locale *sim-non-det-def* =
fixes *R1* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ (*'view1* $\times$ (*'out1* $\times$ *'out2*)) *spmf*
and *S1* :: *'msg1* $\Rightarrow$ *'out1* $\Rightarrow$ *'view1 spmf*
and *Out1* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ *'out1* $\Rightarrow$ (*'out1* $\times$ *'out2*) *spmf* — takes the input of the other party so can form the outputs of parties
and *R2* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ (*'view2* $\times$ (*'out1* $\times$ *'out2*)) *spmf*
and *S2* :: *'msg2* $\Rightarrow$ *'out2* $\Rightarrow$ *'view2 spmf*
and *Out2* :: *'msg2* $\Rightarrow$ *'msg1* $\Rightarrow$ *'out2* $\Rightarrow$ (*'out1* $\times$ *'out2*) *spmf*
and *funct* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ (*'out1* $\times$ *'out2*) *spmf*
begin

type-synonym (*'view'*, *'out1'*, *'out2'*) *adversary-non-det* = (*'view'* $\times$ (*'out1'* $\times$ *'out2'*)) $\Rightarrow$ *bool spmf*

definition *Ideal1* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ *'out1* $\Rightarrow$ (*'view1* $\times$ (*'out1* $\times$ *'out2*)) *spmf*
where *Ideal1 m1 m2 out1* = *do* {
view1 :: *'view1* $\leftarrow$ *S1 m1 out1*;
out1 $\leftarrow$ *Out1 m1 m2 out1*;
return-spmf (*view1*, *out1*)}

definition *Ideal2* :: *'msg2* $\Rightarrow$ *'msg1* $\Rightarrow$ *'out2* $\Rightarrow$ (*'view2* $\times$ (*'out1* $\times$ *'out2*)) *spmf*
where *Ideal2 m2 m1 out2* = *do* {
view2 :: *'view2* $\leftarrow$ *S2 m2 out2*;
out2 $\leftarrow$ *Out2 m2 m1 out2*;
return-spmf (*view2*, *out2*)}

definition *adv-P1* :: *'msg1* $\Rightarrow$ *'msg2* $\Rightarrow$ (*'view1*, *'out1*, *'out2*) *adversary-non-det* $\Rightarrow$ *real*
where *adv-P1 m1 m2 D* $\equiv$ |(*spmf* (*R1 m1 m2* $\gg=$ (λ *view*. *D view*)) *True*) $-$ *spmf* (*funct m1 m2* $\gg=$ (λ (*o1*, *o2*). *Ideal1 m1 m2 o1* $\gg=$ (λ *view*. *D view*))) *True*|

definition *perfect-sec-P1 m1 m2* $\equiv$ (*R1 m1 m2* = *funct m1 m2* $\gg=$ (λ (*s1*, *s2*).

Ideal1 m1 m2 s1))

definition *adv-P2* :: *'msg1* ⇒ *'msg2* ⇒ (*'view2*, *'out1*, *'out2*) *adversary-non-det* ⇒ *real*
where *adv-P2 m1 m2 D* = |*spmf* (*R2 m1 m2* ⤜ (λ *view*. *D view*)) *True* − *spmf* (*funct m1 m2* ⤜ (λ (*o1*, *o2*). *Ideal2 m2 m1 o2* ⤜ (λ *view*. *D view*))) *True*|

definition *perfect-sec-P2 m1 m2* ≡ (*R2 m1 m2* = *funct m1 m2* ⤜ (λ (*s1*, *s2*). *Ideal2 m2 m1 s2*))

end

2.1.3 Secret sharing schemes

locale *secret-sharing-scheme* =
fixes *share* :: *'input-out* ⇒ (*'share* × *'share*) *spmf*
and *reconstruct* :: (*'share* × *'share*) ⇒ *'input-out spmf*
and *F* :: (*'input-out* ⇒ *'input-out* ⇒ *'input-out spmf*) *set*
begin

definition *sharing-correct input* ≡ (*share input* ⤜ (λ (*s1*,*s2*). *reconstruct* (*s1*,*s2*)) = *return-spmf input*)

definition *correct-share-eval input1 input2* ≡ (∀ *gate-eval* ∈ *F*.
∃ *gate-protocol* :: (*'share* × *'share*) ⇒ (*'share* × *'share*) ⇒ (*'share* × *'share*) *spmf*.
share input1 ⤜ (λ (*s1*,*s2*). *share input2*
⤜ (λ (*s3*,*s4*). *gate-protocol* (*s1*,*s3*) (*s2*,*s4*)
⤜ (λ (*S1*,*S2*). *reconstruct* (*S1*,*S2*)))) = *gate-eval input1 input2*)

end

end

2.2 Oblivious Transfer functionalities

Here we define the functionalities for 1-out-of-2 and 1-out-of-4 OT.

theory *OT-Functionalities* **imports**
CryptHOL.CryptHOL
begin

definition *funct-OT-12* :: (*'a* × *'a*) ⇒ *bool* ⇒ (*unit* × *'a*) *spmf*
where *funct-OT-12* $input_1$ σ = *return-spmf* (() , *if* σ *then* (*snd* $input_1$) *else* (*fst* $input_1$))

lemma *lossless-funct-OT-12*: *lossless-spmf* (*funct-OT-12 msgs* σ)
by(*simp add*: *funct-OT-12-def*)

definition *funct-OT-14* :: (*'a* × *'a* × *'a* × *'a*) ⇒ (*bool* × *bool*) ⇒ (*unit* × *'a*) *spmf*
where *funct-OT-14 M C* = *do* {
let (*c0*,*c1*) = *C*;
let (*m00*, *m01*, *m10*, *m11*) = *M*;
return-spmf ((),*if c0 then* (*if c1 then m11 else m10*) *else* (*if c1 then m01 else m00*))}

lemma *lossless-funct-14-OT*: *lossless-spmf* (*funct-OT-14 M C*)
by(*simp add*: *funct-OT-14-def split-def*)

end

2.3 ETP definitions

We define Extended Trapdoor Permutations (ETPs) following [5] and [2]. In particular we consider the property of Hard Core Predicates (HCPs).

theory *ETP* **imports**
CryptHOL.CryptHOL
begin

type-synonym (*'index*,*'range*) *dist2* = (*bool* × *'index* × *bool* × *bool*) ⇒ *bool spmf*

type-synonym (*'index*,*'range*) *advP2* = *'index* ⇒ *bool* ⇒ *bool* ⇒ (*'index*,*'range*) *dist2* ⇒ *'range* ⇒ *bool spmf*

locale *etp* =
fixes *I* :: (*'index* × *'trap*) *spmf* — samples index and trapdoor
and *domain* :: *'index* ⇒ *'range set*
and *range* :: *'index* ⇒ *'range set*
and *F* :: *'index* ⇒ (*'range* ⇒ *'range*) — permutation
and F_{inv} :: *'index* ⇒ *'trap* ⇒ *'range* ⇒ *'range* — must be efficiently computable
and *B* :: *'index* ⇒ *'range* ⇒ *bool* — hard core predicate
assumes *dom-eq-ran*: *y* ∈ *set-spmf I* ⟶ *domain* (*fst y*) = *range* (*fst y*)
and *finite-range*: *y* ∈ *set-spmf I* ⟶ *finite* (*range* (*fst y*))
and *non-empty-range*: *y* ∈ *set-spmf I* ⟶ *range* (*fst y*) ≠ {}
and *bij-betw*: *y* ∈ *set-spmf I* ⟶ *bij-betw* (*F* (*fst y*)) (*domain* (*fst y*)) (*range* (*fst y*))
and *lossless-I*: *lossless-spmf I*
and *F-f-inv*: *y* ∈ *set-spmf I* ⟶ *x* ∈ *range* (*fst y*) ⟶ F_{inv} (*fst y*) (*snd y*) (*F* (*fst y*) *x*) = *x*
begin

definition *S* :: *'index* ⇒ *'range spmf*
where *S* α = *spmf-of-set* (*range* α)

lemma *lossless-S*: *y* ∈ *set-spmf I* ⟶ *lossless-spmf* (*S* (*fst y*))
by(*simp add*: *lossless-spmf-def S-def finite-range non-empty-range*)

lemma *set-spmf-S* [*simp*]: *y* ∈ *set-spmf I* ⟶ *set-spmf* (*S* (*fst y*)) = *range* (*fst y*)

by (*simp add*: *S-def finite-range*)

lemma *f-inj-on*: $y \in$ *set-spmf I* $\longrightarrow$ *inj-on* (*F* (*fst y*)) (*range* (*fst y*))
by(*metis bij-betw-def bij-betw dom-eq-ran bij-betw-def bij-betw dom-eq-ran*)

lemma *range-f*: $y \in$ *set-spmf I* $\longrightarrow$ $x \in$ *range* (*fst y*) $\longrightarrow$ *F* (*fst y*) $x \in$ *range* (*fst y*)
by (*metis bij-betw bij-betw dom-eq-ran bij-betwE*)

lemma *f-inv-f* [*simp*]: $y \in$ *set-spmf I* $\longrightarrow$ $x \in$ *range* (*fst y*) $\longrightarrow$ F_{inv} (*fst y*) (*snd y*) (*F* (*fst y*) x) = x
by (*metis bij-betw bij-betw-inv-into-left dom-eq-ran F-f-inv*)

lemma *f-inv-f'* [*simp*]: $y \in$ *set-spmf I* $\longrightarrow$ $x \in$ *range* (*fst y*) $\longrightarrow$ *Hilbert-Choice.inv-into* (*range* (*fst y*)) (*F* (*fst y*)) (*F* (*fst y*) x) = x
by (*metis bij-betw bij-betw-inv-into-left bij-betw dom-eq-ran*)

lemma *B-F-inv-rewrite*: (B α (F_{inv} α τ $y_\sigma{}'$) = (B α (F_{inv} α τ $y_\sigma{}'$) = *m1*)) = *m1*
by *auto*

lemma *uni-set-samp*:
assumes $y \in$ *set-spmf I*
shows *map-spmf* (λ x. *F* (*fst y*) x) (*S* (*fst y*)) = (*S* (*fst y*))
(**is** *?lhs* = *?rhs*)
proof−
have *rhs*: *?rhs* = *spmf-of-set* (*range* (*fst y*))
unfolding *S-def* **by**(*simp*)
also have *map-spmf* (λ x. *F* (*fst y*) x) (*spmf-of-set* (*range* (*fst y*))) = *spmf-of-set* ((λ x. *F* (*fst y*) x) ‘ (*range* (*fst y*)))
using *f-inj-on assms*
by (*metis map-spmf-of-set-inj-on*)
also have (λ x. *F* (*fst y*) x) ‘ (*range* (*fst y*)) = *range* (*fst y*)
apply(*rule endo-inj-surj*)
using *bij-betw*
by (*auto simp add*: *bij-betw-def dom-eq-ran f-inj-on bij-betw finite-range assms*)

finally show *?thesis* **by**(*simp add*: *rhs*)
qed

We define the security property of the hard core predicate (HCP) using a game.

definition *HCP-game* :: (*'index,'range*) *advP2* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ (*'index,'range*) *dist2* $\Rightarrow$ *bool spmf*
where *HCP-game A* = (λ σ b_σ *D*. *do* {
(α, τ) $\leftarrow$ *I*;
$x \leftarrow S$ α;
$b' \leftarrow A$ α σ b_σ *D* x;

let $b = B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x)$;
return-spmf $(b = b')\})$

definition *HCP-adv A* σ b_σ *D* = |((*spmf* (*HCP-game A* σ b_σ *D*) *True*) − *1/2*)|

end

end

2.4 Oblivious transfer constructed from ETPs

Here we construct the OT protocol based on ETPs given in [5] (Chapter 4) and prove semi honest security for both parties. We show information theoretic security for Party 1 and reduce the security of Party 2 to the HCP assumption.

theory *ETP-OT* **imports**
HOL−Number-Theory.Cong
ETP
OT-Functionalities
Semi-Honest-Def
begin

type-synonym *'range viewP1* = ((*bool* × *bool*) × *'range* × *'range*) *spmf*
type-synonym *'range dist1* = ((*bool* × *bool*) × *'range* × *'range*) ⇒ *bool spmf*
type-synonym *'index viewP2* = (*bool* × *'index* × (*bool* × *bool*)) *spmf*
type-synonym *'index dist2* = (*bool* × *'index* × *bool* × *bool*) ⇒ *bool spmf*
type-synonym (*'index*, *'range*) *advP2* = *'index* ⇒ *bool* ⇒ *bool* ⇒ *'index dist2* ⇒ *'range* ⇒ *bool spmf*

lemma *if-False-True*: (*if x then False else* ¬ *False*) ⟷ (*if x then False else True*)
by *simp*

lemma *if-then-True* [*simp*]: (*if b then True else x*) ⟷ (¬ *b* ⟶ *x*)
by *simp*

lemma *if-else-True* [*simp*]: (*if b then x else True*) ⟷ (*b* ⟶ *x*)
by *simp*

lemma *inj-on-Not* [*simp*]: *inj-on Not A*
by(*auto simp add*: *inj-on-def*)

locale *ETP-base* = *etp*: *etp I domain range F* F_{inv} *B*
for *I* :: (*'index* × *'trap*) *spmf* — samples index and trapdoor
and *domain* :: *'index* ⇒ *'range set*
and *range* :: *'index* ⇒ *'range set*
and *B* :: *'index* ⇒ *'range* ⇒ *bool* — hard core predicate
and *F* :: *'index* ⇒ *'range* ⇒ *'range*
and F_{inv} :: *'index* ⇒ *'trap* ⇒ *'range* ⇒ *'range*

begin

The probabilistic program that defines the protocol.

definition *protocol* :: (*bool* × *bool*) ⇒ *bool* ⇒ (*unit* × *bool*) *spmf*
  where *protocol* $input_1$ σ = *do* {
    let (b_σ, $b_\sigma{}'$) = $input_1$;
    (α :: *'index*, τ :: *'trap*) ← *I*;
    x_σ :: *'range* ← *etp.S* α;
    $y_\sigma{}'$ :: *'range* ← *etp.S* α;
    let (y_σ :: *'range*) = *F* α x_σ;
    let (x_σ :: *'range*) = F_{inv} α τ y_σ;
    let ($x_\sigma{}'$ :: *'range*) = F_{inv} α τ $y_\sigma{}'$;
    let (β_σ :: *bool*) = *xor* (*B* α x_σ) b_σ;
    let ($\beta_\sigma{}'$ :: *bool*) = *xor* (*B* α $x_\sigma{}'$) $b_\sigma{}'$;
    return-spmf ((), *if* σ *then xor* (*B* α $x_\sigma{}'$) $\beta_\sigma{}'$ *else xor* (*B* α x_σ) β_σ)}

lemma *correctness*: *protocol* (*m0*,*m1*) *c* = *funct-OT-12* (*m0*,*m1*) *c*
proof−
  have (*B* α (F_{inv} α τ $y_\sigma{}'$) = (*B* α (F_{inv} α τ $y_\sigma{}'$) = *m1*)) = *m1*
    for α τ $y_\sigma{}'$ **by** *auto*
  then show *?thesis*
    by(*auto simp add*: *protocol-def funct-OT-12-def Let-def etp.B-F-inv-rewrite bind-spmf-const etp.lossless-S local.etp.lossless-I lossless-weight-spmfD split-def cong*: *bind-spmf-cong*)
qed

Party 1 views

definition *R1* :: (*bool* × *bool*) ⇒ *bool* ⇒ *'range viewP1*
  where *R1* $input_1$ σ = *do* {
    let (b_0, b_1) = $input_1$;
    (α, τ) ← *I*;
    x_σ ← *etp.S* α;
    $y_\sigma{}'$ ← *etp.S* α;
    let y_σ = *F* α x_σ;
    return-spmf ((b_0, b_1), *if* σ *then* $y_\sigma{}'$ *else* y_σ, *if* σ *then* y_σ *else* $y_\sigma{}'$)}

lemma *lossless-R1*: *lossless-spmf* (*R1 msgs* σ)
  by(*simp add*: *R1-def local.etp.lossless-I split-def etp.lossless-S Let-def*)

definition *S1* :: (*bool* × *bool*) ⇒ *unit* ⇒ *'range viewP1*
  where *S1* == (λ $input_1$ (). *do* {
    let (b_0, b_1) = $input_1$;
    (α, τ) ← *I*;
    y_0 :: *'range* ← *etp.S* α;
    y_1 ← *etp.S* α;
    return-spmf ((b_0, b_1), y_0, y_1)})

lemma *lossless-S1*: *lossless-spmf* (*S1 msgs* ())
  by(*simp add*: *S1-def local.etp.lossless-I split-def etp.lossless-S*)

Party 2 views

definition *R2* :: (*bool* × *bool*) ⇒ *bool* ⇒ *'index viewP2*
  where *R2 msgs* σ = *do* {
    let (*b0*,*b1*) = *msgs*;
    (α, τ) ← *I*;
    x_σ ← *etp.S* α;
    $y_\sigma{}'$ ← *etp.S* α;
    let y_σ = *F* α x_σ;
    let x_σ = F_{inv} α τ y_σ;
    let $x_\sigma{}'$ = F_{inv} α τ $y_\sigma{}'$;
    let β_σ = (*B* α x_σ) ⊕ (*if* σ *then b1 else b0*) ;
    let $\beta_\sigma{}'$ = (*B* α $x_\sigma{}'$) ⊕ (*if* σ *then b0 else b1*);
    return-spmf (σ, α,(β_σ, $\beta_\sigma{}'$))}

lemma *lossless-R2*: *lossless-spmf* (*R2 msgs* σ)
  by(*simp add*: *R2-def split-def local.etp.lossless-I etp.lossless-S*)

definition *S2* :: *bool* ⇒ *bool* ⇒ *'index viewP2*
  where *S2* σ b_σ = *do* {
    (α, τ) ← *I*;
    x_σ ← *etp.S* α;
    $y_\sigma{}'$ ← *etp.S* α;
    let $x_\sigma{}'$ = F_{inv} α τ $y_\sigma{}'$;
    let β_σ = (*B* α x_σ) ⊕ b_σ;
    let $\beta_\sigma{}'$ = *B* α $x_\sigma{}'$;
    return-spmf (σ, α, (β_σ, $\beta_\sigma{}'$))}

lemma *lossless-S2*: *lossless-spmf* (*S2* σ b_σ)
  by(*simp add*: *S2-def local.etp.lossless-I etp.lossless-S split-def*)

Security for Party 1

We have information theoretic security for Party 1.

lemma *P1-security*: *R1* $input_1$ σ = *funct-OT-12 x y* ⋙ (λ (*s1*, *s2*). *S1* $input_1$ *s1*)
  including *monad-normalisation*
proof−
   have *R1* $input_1$ σ = *do* {
    let (*b0*,*b1*) = $input_1$;
    (α, τ) ← *I*;
    $y_\sigma{}'$ :: *'range* ← *etp.S* α;
    y_σ ← *map-spmf* (λ x_σ. *F* α x_σ) (*etp.S* α);
    return-spmf ((*b0*,*b1*), *if* σ *then* $y_\sigma{}'$ *else* y_σ, *if* σ *then* y_σ *else* $y_\sigma{}'$)}
     by(*simp add*: *bind-map-spmf o-def Let-def R1-def*)
   also have ... = *do* {
    let (*b0*,*b1*) = $input_1$;
    (α, τ) ← *I*;
    $y_\sigma{}'$ :: *'range* ← *etp.S* α;
    y_σ ← *etp.S* α;

return-spmf ((*b0*,*b1*), *if* σ *then* $y_\sigma{}'$ *else* y_σ, *if* σ *then* y_σ *else* $y_\sigma{}'$)}
by(*simp add*: *etp.uni-set-samp Let-def split-def cong*: *bind-spmf-cong*)
also have ... = *funct-OT-12 x y* $\gg\!=$ (λ (*s1*, *s2*). *S1* $input_1$ *s1*)
by(*cases* σ; *simp add*: *S1-def R1-def Let-def funct-OT-12-def*)
ultimately show *?thesis* **by** *auto*
qed

The adversary used in proof of security for party 2

definition $\mathcal{A}$ :: (*'index*, *'range*) *advP2*
where $\mathcal{A}$ α σ b_σ *D2 x* = *do* {
$\beta_\sigma{}'$ ← *coin-spmf*;
x_σ ← *etp.S* α;
let β_σ = (*B* α x_σ) $\oplus$ b_σ;
d ← *D2*(σ, α, β_σ, $\beta_\sigma{}'$);
return-spmf(*if d then* $\beta_\sigma{}'$ *else* ¬ $\beta_\sigma{}'$)}

lemma *lossless-*$\mathcal{A}$:
assumes ∀ *view*. *lossless-spmf* (*D2 view*)
shows *y* ∈ *set-spmf I* ⟶ *lossless-spmf* ($\mathcal{A}$ (*fst y*) σ b_σ *D2 x*)
by(*simp add*: $\mathcal{A}$*-def etp.lossless-S assms*)

lemma *assm-bound-funct-OT-12*:
assumes *etp.HCP-adv* $\mathcal{A}$ σ (*if* σ *then b1 else b0*) *D* ≤ *HCP-ad*
shows |*spmf* (*funct-OT-12* (*b0*,*b1*) σ $\gg\!=$ (λ (*out1*,*out2*).
etp.HCP-game $\mathcal{A}$ σ *out2 D*)) *True* − *1*/*2*| ≤ *HCP-ad*
(**is** *?lhs* ≤ *HCP-ad*)
proof−
have *?lhs* = |*spmf* (*etp.HCP-game* $\mathcal{A}$ σ (*if* σ *then b1 else b0*) *D*) *True* − *1*/*2*|
by(*simp add*: *funct-OT-12-def*)
thus *?thesis* **using** *assms etp.HCP-adv-def* **by** *simp*
qed

lemma *assm-bound-funct-OT-12-collapse*:
assumes ∀ b_σ. *etp.HCP-adv* $\mathcal{A}$ σ b_σ *D* ≤ *HCP-ad*
shows |*spmf* (*funct-OT-12 m1* σ $\gg\!=$ (λ (*out1*,*out2*). *etp.HCP-game* $\mathcal{A}$ σ *out2 D*)) *True* − *1*/*2*| ≤ *HCP-ad*
using *assm-bound-funct-OT-12 surj-pair assms* **by** *metis*

To prove security for party 2 we split the proof on the cases on party 2's input

lemma *R2-S2-False*:
assumes ((*if* σ *then b0 else b1*) = *False*)
shows *spmf* (*R2* (*b0*,*b1*) σ $\gg\!=$ (*D2* :: (*bool* × *'index* × *bool* × *bool*) ⇒ *bool spmf*)) *True*
= *spmf* (*funct-OT-12* (*b0*,*b1*) σ $\gg\!=$ (λ (*out1*,*out2*). *S2* σ *out2* $\gg\!=$ *D2*)) *True*
proof−
have σ ⟹ ¬ *b0* **using** *assms* **by** *simp*
moreover have ¬ σ ⟹ ¬ *b1* **using** *assms* **by** *simp*

ultimately show *?thesis*
by(*auto simp add*: *R2-def S2-def split-def local.etp.F-f-inv assms funct-OT-12-def cong*: *bind-spmf-cong-simp*)
qed

lemma *R2-S2-True*:
assumes $((if\ \sigma\ then\ b0\ else\ b1) = True)$
and *lossless-D*: $\forall\ a.\ lossless\text{-}spmf\ (D2\ a)$
shows $|(spmf\ (bind\text{-}spmf\ (R2\ (b0,b1)\ \sigma)\ D2)\ True) - spmf\ (funct\text{-}OT\text{-}12\ (b0,b1)\ \sigma \gg= (\lambda\ (out1,\ out2).\ S2\ \sigma\ out2 \gg= (\lambda\ view.\ D2\ view)))\ True|$
$= |2*((spmf\ (etp.HCP\text{-}game\ \mathcal{A}\ \sigma\ (if\ \sigma\ then\ b1\ else\ b0)\ D2)\ True) - 1/2)|$
proof−
have $(spmf\ (funct\text{-}OT\text{-}12\ (b0,b1)\ \sigma \gg= (\lambda\ (out1,\ out2).\ S2\ \sigma\ out2 \gg= D2))\ True$
$- spmf\ (bind\text{-}spmf\ (R2\ (b0,b1)\ \sigma)\ D2)\ True)$
$= 2 * ((spmf\ (etp.HCP\text{-}game\ \mathcal{A}\ \sigma\ (if\ \sigma\ then\ b1\ else\ b0)\ D2)\ True) - 1/2)$
proof−
have $((spmf\ (etp.HCP\text{-}game\ \mathcal{A}\ \sigma\ (if\ \sigma\ then\ b1\ else\ b0)\ D2)\ True) - 1/2) =$
$1/2*(spmf\ (bind\text{-}spmf\ (S2\ \sigma\ (if\ \sigma\ then\ b1\ else\ b0))\ D2)\ True$
$- spmf\ (bind\text{-}spmf\ (R2\ (b0,b1)\ \sigma)\ D2)\ True)$
including *monad-normalisation*
proof−
have *σ-true-b0-true*: $\sigma \Longrightarrow b0 = True$ **using** *assms*(*1*) **by** *simp*
have *σ-false-b1-true*: $\neg\ \sigma \Longrightarrow b1$ **using** *assms*(*1*) **by** *simp*
have *return-True-False*: $spmf\ (return\text{-}spmf\ (\neg\ d))\ True = spmf\ (return\text{-}spmf\ d)\ False$
for *d* **by**(*cases d*; *simp*)
define *HCP-game-true* **where** *HCP-game-true* $== \lambda\ \sigma\ b_\sigma.\ do\ \{$
$(\alpha,\ \tau) \leftarrow I;$
$x_\sigma \leftarrow etp.S\ \alpha;$
$x \leftarrow (etp.S\ \alpha);$
$let\ \beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma;$
$let\ \beta_\sigma{}' = B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x);$
$d \leftarrow D2(\sigma,\ \alpha,\ \beta_\sigma,\ \beta_\sigma{}');$
$let\ b' = (if\ d\ then\ \beta_\sigma{}'\ else\ \neg\ \beta_\sigma{}');$
$let\ b = B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x);$
$return\text{-}spmf\ (b = b')\}$
define *HCP-game-false* **where** *HCP-game-false* $== \lambda\ \sigma\ b_\sigma.\ do\ \{$
$(\alpha,\ \tau) \leftarrow I;$
$x_\sigma \leftarrow etp.S\ \alpha;$
$x \leftarrow (etp.S\ \alpha);$
$let\ \beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma;$
$let\ \beta_\sigma{}' = \neg\ B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x);$
$d \leftarrow D2(\sigma,\ \alpha,\ \beta_\sigma,\ \beta_\sigma{}');$
$let\ b' = (if\ d\ then\ \beta_\sigma{}'\ else\ \neg\ \beta_\sigma{}');$
$let\ b = B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x);$

return-spmf $(b = b')\}$
define *HCP-game-$\mathcal{A}$* **where** *HCP-game-$\mathcal{A}$* $== \lambda\ \sigma\ b_\sigma.$ *do* $\{$
$\beta_\sigma{}' \leftarrow$ *coin-spmf*;
$(\alpha,\ \tau) \leftarrow I$;
$x \leftarrow$ *etp.S* α;
$x' \leftarrow$ *etp.S* α;
$d \leftarrow D2\ (\sigma,\ \alpha,\ (B\ \alpha\ x) \oplus b_\sigma,\ \beta_\sigma{}')$;
let $b' =$ (*if* d *then* $\beta_\sigma{}'$ *else* $\neg\ \beta_\sigma{}'$);
return-spmf $(B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x') = b')\}$
define *S2D* **where** *S2D* $== \lambda\ \sigma\ b_\sigma$. *do* $\{$
$(\alpha,\ \tau) \leftarrow I$;
$x_\sigma \leftarrow$ *etp.S* α;
$y_\sigma{}' \leftarrow$ *etp.S* α;
let $x_\sigma{}' = F_{inv}\ \alpha\ \tau\ y_\sigma{}'$;
let $\beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma$;
let $\beta_\sigma{}' = B\ \alpha\ x_\sigma{}'$;
d :: *bool* $\leftarrow D2(\sigma,\ \alpha,\ \beta_\sigma,\ \beta_\sigma{}')$;
return-spmf $d\}$
define *R2D* **where** *R2D* $== \lambda$ *msgs* σ. *do* $\{$
let $(b0, b1) =$ *msgs*;
$(\alpha,\ \tau) \leftarrow I$;
$x_\sigma \leftarrow$ *etp.S* α;
$y_\sigma{}' \leftarrow$ *etp.S* α;
let $y_\sigma = F\ \alpha\ x_\sigma$;
let $x_\sigma = F_{inv}\ \alpha\ \tau\ y_\sigma$;
let $x_\sigma{}' = F_{inv}\ \alpha\ \tau\ y_\sigma{}'$;
let $\beta_\sigma = (B\ \alpha\ x_\sigma) \oplus$ (*if* σ *then* $b1$ *else* $b0$) ;
let $\beta_\sigma{}' = (B\ \alpha\ x_\sigma{}') \oplus$ (*if* σ *then* $b0$ *else* $b1$);
b :: *bool* $\leftarrow D2(\sigma,\ \alpha,(\beta_\sigma,\ \beta_\sigma{}'))$;
return-spmf $b\}$
define *D-true* **where** *D-true* $== \lambda\sigma\ b_\sigma.$ *do* $\{$
$(\alpha,\ \tau) \leftarrow I$;
$x_\sigma \leftarrow$ *etp.S* α;
$x \leftarrow$ (*etp.S* α);
let $\beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma$;
let $\beta_\sigma{}' = B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x)$;
d :: *bool* $\leftarrow D2(\sigma,\ \alpha,\ \beta_\sigma,\ \beta_\sigma{}')$;
return-spmf $d\}$
define *D-false* **where** *D-false* $== \lambda\ \sigma\ b_\sigma.$ *do* $\{$
$(\alpha,\ \tau) \leftarrow I$;
$x_\sigma \leftarrow$ *etp.S* α;
$x \leftarrow$ *etp.S* α;
let $\beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma$;
let $\beta_\sigma{}' = \neg\ B\ \alpha\ (F_{inv}\ \alpha\ \tau\ x)$;
d :: *bool* $\leftarrow D2(\sigma,\ \alpha,\ \beta_\sigma,\ \beta_\sigma{}')$;
return-spmf $d\}$
have *lossless-D-false*: *lossless-spmf* (*D-false* σ (*if* σ *then* $b1$ *else* $b0$))
apply(*auto simp add*: *D-false-def lossless-D local.etp.lossless-I*)
using *local.etp.lossless-S* **by** *auto*

have *spmf* (*etp.HCP-game* $\mathcal{A}$ σ (*if* σ *then* *b1* *else* *b0*) *D2*) *True* = *spmf* (*HCP-game*-$\mathcal{A}$ σ (*if* σ *then* *b1* *else* *b0*)) *True*
apply(*simp add*: *etp.HCP-game-def HCP-game*-$\mathcal{A}$*-def* $\mathcal{A}$*-def split-def etp.F-f-inv*)
by(*rewrite bind-commute-spmf*[**where** *q* = *coin-spmf*]; *rewrite bind-commute-spmf*[**where** *q* = *coin-spmf*]; *rewrite bind-commute-spmf*[**where** *q* = *coin-spmf*]; *auto*)+
also have ... = *spmf* (*bind-spmf* (*map-spmf Not coin-spmf*) (λb. *if* *b* *then* *HCP-game-true* σ (*if* σ *then* *b1* *else* *b0*) *else* *HCP-game-false* σ (*if* σ *then* *b1* *else* *b0*))) *True*
unfolding *HCP-game*-$\mathcal{A}$*-def HCP-game-true-def HCP-game-false-def* $\mathcal{A}$*-def* *Let-def*
apply(*simp add*: *split-def cong*: *if-cong*)
supply [[*simproc del*: *monad-normalisation*]]
apply(*subst if-distrib*[**where** *f* = *bind-spmf* - **for** *f*, *symmetric*]; *simp cong*: *bind-spmf-cong add*: *if-distribR*)+
apply(*rewrite* **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** ⌑ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
apply(*fold map-spmf-conv-bind-spmf*)
apply(*rule conjI*; *rule impI*; *simp*)
apply(*simp only*: *spmf-bind*)
apply(*rule Bochner-Integration.integral-cong*[*OF refl*])+
apply *clarify*
subgoal for r r_σ α τ
apply(*simp only*: *UNIV-bool spmf-of-set integral-spmf-of-set*)
apply(*simp cong*: *if-cong split del*: *if-split*)
apply(*cases B r* (F_{inv} r r_σ τ))
by *auto*
apply(*rewrite* **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** ⌑ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
apply(*simp only*: *spmf-bind*)
apply(*rule Bochner-Integration.integral-cong*[*OF refl*])+
apply *clarify*
subgoal for r r_σ α τ
apply(*simp only*: *UNIV-bool spmf-of-set integral-spmf-of-set*)
apply(*simp cong*: *if-cong split del*: *if-split*)
apply(*cases B r* (F_{inv} r r_σ τ))
by *auto*
done
also have ... = *1/2**(*spmf* (*HCP-game-true* σ (*if* σ *then* *b1* *else* *b0*)) *True*) + *1/2**(*spmf* (*HCP-game-false* σ (*if* σ *then* *b1* *else* *b0*)) *True*)
by(*simp add*: *spmf-bind UNIV-bool spmf-of-set integral-spmf-of-set*)
also have ... = *1/2**(*spmf* (*D-true* σ (*if* σ *then* *b1* *else* *b0*)) *True*) +

1/2∗(*spmf* (*D-false* σ (*if* σ *then b1 else b0*)) *False*)
proof−
have *spmf* (*I* ⪢= (λ(α, τ). *etp.S* α ⪢= (λ$x_σ$. *etp.S* α ⪢= (λ*x*. *D2* (σ, α, *B* α $x_σ$ = (¬ (*if* σ *then b1 else b0*)), ¬ *B* α (F_{inv} α τ *x*)) ⪢= (λ*d*. *return-spmf* (¬ *d*)))))) *True*
= *spmf* (*I* ⪢= (λ(α, τ). *etp.S* α ⪢= (λ$x_σ$. *etp.S* α ⪢= (λ*x*. *D2* (σ, α, *B* α $x_σ$ = (¬ (*if* σ *then b1 else b0*)), ¬ *B* α (F_{inv} α τ *x*)))))) *False*
(**is** *?lhs* = *?rhs*)
proof−
have *?lhs* = *spmf* (*I* ⪢= (λ(α, τ). *etp.S* α ⪢= (λ$x_σ$. *etp.S* α ⪢= (λ*x*. *D2* (σ, α, *B* α $x_σ$ = (¬ (*if* σ *then b1 else b0*)), ¬ *B* α (F_{inv} α τ *x*)) ⪢= (λ*d*. *return-spmf* (*d*)))))) *False*
by(*simp only*: *split-def return-True-False spmf-bind*)
then show *?thesis* **by** *simp*
qed
then show *?thesis* **by**(*simp add*: *HCP-game-true-def HCP-game-false-def Let-def D-true-def D-false-def if-distrib*[**where** *f*=(=) -] *cong*: *if-cong*)
qed
also have ... = *1/2*∗((*spmf* (*D-true* σ (*if* σ *then b1 else b0*)) *True*) + (*1* − *spmf* (*D-false* σ (*if* σ *then b1 else b0*)) *True*))
by(*simp add*: *spmf-False-conv-True lossless-D-false*)
also have ... = *1/2* + *1/2*∗ (*spmf* (*D-true* σ (*if* σ *then b1 else b0*)) *True*) − *1/2*∗(*spmf* (*D-false* σ (*if* σ *then b1 else b0*)) *True*)
by(*simp*)
also have ... = *1/2* + *1/2*∗ (*spmf* (*S2D* σ (*if* σ *then b1 else b0*)) *True*) − *1/2*∗(*spmf* (*R2D* (*b0*,*b1*) σ) *True*)
apply(*auto simp add*: *local.etp.F-f-inv S2D-def R2D-def D-true-def D-false-def assms split-def cong*: *bind-spmf-cong-simp*)
apply(*simp add*: *σ-true-b0-true*)
by(*simp add*: *σ-false-b1-true*)
ultimately show *?thesis* **by**(*simp add*: *S2D-def R2D-def R2-def S2-def split-def*)
qed
then show *?thesis* **by**(*auto simp add*: *funct-OT-12-def*)
qed
thus *?thesis* **by** *simp*
qed

lemma *P2-adv-bound*:
assumes *lossless-D*: ∀ *a*. *lossless-spmf* (*D2 a*)
shows |(*spmf* (*bind-spmf* (*R2* (*b0*,*b1*) σ) *D2*) *True*) − *spmf* (*funct-OT-12* (*b0*,*b1*) σ ⪢= (λ (*out1*, *out2*). *S2* σ *out2* ⪢= (λ *view*. *D2 view*))) *True*|
≤ |*2*∗((*spmf* (*etp.HCP-game* $\mathcal{A}$ σ (*if* σ *then b1 else b0*) *D2*) *True*) − *1/2*)|
by(*cases* (*if* σ *then b0 else b1*); *auto simp add*: *R2-S2-False R2-S2-True assms*)

sublocale *OT-12*: *sim-det-def R1 S1 R2 S2 funct-OT-12 protocol*
unfolding *sim-det-def-def*
by(*simp add*: *lossless-R1 lossless-S1 lossless-R2 lossless-S2 funct-OT-12-def*)

lemma *correct*: *OT-12.correctness m1 m2*
unfolding *OT-12.correctness-def*
by (*metis prod.collapse correctness*)

lemma *P1-security-inf-the*: *OT-12.perfect-sec-P1 m1 m2*
unfolding *OT-12.perfect-sec-P1-def* **using** *P1-security* **by** *simp*

lemma *P2-security*:
assumes $\forall$ *a. lossless-spmf* (*D a*)
and $\forall$ b_σ. *etp.HCP-adv* $\mathcal{A}$ *m2* b_σ *D* $\leq$ *HCP-ad*
shows *OT-12.adv-P2 m1 m2 D* $\leq$ *2* $*$ *HCP-ad*
proof−
have *spmf* (*etp.HCP-game* $\mathcal{A}$ σ (*if* σ *then b1 else b0*) *D*) *True* = *spmf* (*funct-OT-12* (*b0*,*b1*) σ $\gg=$ (λ (*out1*, *out2*). *etp.HCP-game* $\mathcal{A}$ σ *out2 D*)) *True*
for σ *b0 b1*
by(*simp add*: *funct-OT-12-def*)
hence *OT-12.adv-P2 m1 m2 D* $\leq$ |*2*$*$((*spmf* (*funct-OT-12 m1 m2* $\gg=$ (λ (*out1*, *out2*). *etp.HCP-game* $\mathcal{A}$ *m2 out2 D*)) *True*) − *1*/*2*)|
unfolding *OT-12.adv-P2-def* **using** *P2-adv-bound assms surj-pair prod.collapse* **by** *metis*
moreover have |*2*$*$((*spmf* (*funct-OT-12 m1 m2* $\gg=$ (λ (*out1*, *out2*). *etp.HCP-game* $\mathcal{A}$ *m2 out2 D*)) *True*) − *1*/*2*)| $\leq$ |*2*$*$*HCP-ad*|
proof −
have ($\exists$ *r*. |(*1*::*real*) / *r*| $\neq$ *1* / |*r*|) $\vee$ *2* / |*1* / (*spmf* (*funct-OT-12 m1 m2* $\gg=$ (λ(*x*, *y*). ((λ*u b*. *etp.HCP-game* $\mathcal{A}$ *m2 b D*)::*unit* $\Rightarrow$ *bool* $\Rightarrow$ *bool spmf*) *x y*)) *True* − *1* / *2*)|
$\leq$ *HCP-ad* / (*1* / *2*)
using *assm-bound-funct-OT-12-collapse assms* **by** *auto*
then show *?thesis*
by *fastforce*
qed
moreover have *HCP-ad* $\geq$ *0*
using *assms*(*2*) *local.etp.HCP-adv-def* **by** *auto*
ultimately show *?thesis* **by** *argo*
qed

end

We also consider the asymptotic case for security proofs

locale *ETP-sec-para* =
fixes *I* :: *nat* $\Rightarrow$ (*'index* $\times$ *'trap*) *spmf*
and *domain* :: *'index* $\Rightarrow$ *'range set*
and *range* :: *'index* $\Rightarrow$ *'range set*
and *f* :: *'index* $\Rightarrow$ (*'range* $\Rightarrow$ *'range*)
and *F* :: *'index* $\Rightarrow$ *'range* $\Rightarrow$ *'range*
and F_{inv} :: *'index* $\Rightarrow$ *'trap* $\Rightarrow$ *'range* $\Rightarrow$ *'range*
and *B* :: *'index* $\Rightarrow$ *'range* $\Rightarrow$ *bool*
assumes *ETP-base*: $\bigwedge$ *n*. *ETP-base* (*I n*) *domain range F* F_{inv}

begin

sublocale *ETP-base* (*I n*) *domain range*
using *ETP-base* **by** *simp*

lemma *correct-asym*: *OT-12.correctness n m1 m2*
by(*simp add*: *correct*)

lemma *P1-sec-asym*: *OT-12.perfect-sec-P1 n m1 m2*
using *P1-security-inf-the* **by** *simp*

lemma *P2-sec-asym*:
assumes ∀ *a*. *lossless-spmf* (*D a*)
and *HCP-adv-neg*: *negligible* (λ *n*. *etp-advantage n*)
and *etp-adv-bound*: ∀ b_σ *n*. *etp.HCP-adv n* $\mathcal{A}$ *m2* b_σ *D* ≤ *etp-advantage n*
shows *negligible* (λ *n*. *OT-12.adv-P2 n m1 m2 D*)
proof−
have *negligible* (λ *n*. *2* ∗ *etp-advantage n*) **using** *HCP-adv-neg*
by (*simp add*: *negligible-cmultI*)
moreover have |*OT-12.adv-P2 n m1 m2 D*| = *OT-12.adv-P2 n m1 m2 D* **for** *n* **unfolding** *OT-12.adv-P2-def* **by** *simp*
moreover have *OT-12.adv-P2 n m1 m2 D* ≤ *2* ∗ *etp-advantage n* **for** *n* **using** *assms P2-security* **by** *blast*
ultimately show *?thesis*
using *assms negligible-le HCP-adv-neg P2-security* **by** *presburger*
qed

end

end

2.4.1 RSA instantiation

It is known that the RSA collection forms an ETP. Here we instantitate our proof of security for OT that uses a general ETP for RSA. We use the proof of the general construction of OT. The main proof effort here is in showing the RSA collection meets the requirements of an ETP, mainly this involves showing the RSA mapping is a bijection.

theory *ETP-RSA-OT* **imports**
ETP-OT
Number-Theory-Aux
Uniform-Sampling
begin

type-synonym *index* = (*nat* × *nat*)
type-synonym *trap* = *nat*
type-synonym *range* = *nat*
type-synonym *domain* = *nat*

type-synonym *viewP1* = ((*bool* × *bool*) × *nat* × *nat*) *spmf*
type-synonym *viewP2* = (*bool* × *index* × (*bool* × *bool*)) *spmf*
type-synonym *dist2* = (*bool* × *index* × *bool* × *bool*) ⇒ *bool spmf*
type-synonym *advP2* = *index* ⇒ *bool* ⇒ *bool* ⇒ *dist2* ⇒ *bool spmf*

locale *rsa-base* =
  fixes *prime-set* :: *nat set* — the set of primes used
    and *B* :: *index* ⇒ *nat* ⇒ *bool*
  assumes *prime-set-ass*: *prime-set* ⊆ {*x*. *prime x* ∧ *x* > *2*}
    and *finite-prime-set*: *finite prime-set*
    and *prime-set-gt-2*: *card prime-set* > *2*
begin

lemma *prime-set-non-empty*: *prime-set* ≠ {}
  using *prime-set-gt-2* **by** *auto*

definition *coprime-set* :: *nat* ⇒ *nat set*
  where *coprime-set N* ≡ {*x*. *coprime x N* ∧ *x* > *1* ∧ *x* < *N*}

lemma *coprime-set-non-empty*:
  assumes *N* > *2*
  shows *coprime-set N* ≠ {}
  by(*simp add*: *coprime-set-def*; *metis assms*(*1*) *Suc-lessE coprime-Suc-right-nat lessI numeral-2-eq-2*)

definition *sample-coprime* :: *nat* ⇒ *nat spmf*
  where *sample-coprime N* = *spmf-of-set* (*coprime-set* (*N*))

lemma *sample-coprime-e-gt-1*:
  assumes *e* ∈ *set-spmf* (*sample-coprime N*)
  shows *e* > *1*
  using *assms* **by**(*simp add*: *sample-coprime-def coprime-set-def*)

lemma *lossless-sample-coprime*:
  assumes ¬ *prime N*
    and *N* > *2*
  shows *lossless-spmf* (*sample-coprime N*)
proof−
  have *coprime-set N* ≠ {}
    by(*simp add*: *coprime-set-non-empty assms*)
  also have *finite* (*coprime-set N*)
    by(*simp add*: *coprime-set-def*)
  ultimately show *?thesis* **by**(*simp add*: *sample-coprime-def*)
qed

lemma *set-spmf-sample-coprime*:
  shows *set-spmf* (*sample-coprime N*) = {*x*. *coprime x N* ∧ *x* > *1* ∧ *x* < *N*}
  by(*simp add*: *sample-coprime-def coprime-set-def*)

definition *sample-primes* :: *nat spmf*
  where *sample-primes* = *spmf-of-set prime-set*

lemma *lossless-sample-primes*:
  shows *lossless-spmf sample-primes*
    by(*simp add*: *sample-primes-def prime-set-non-empty finite-prime-set*)

lemma *set-spmf-sample-primes*:
  shows *set-spmf sample-primes* ⊆ {*x*. *prime x* ∧ *x* > *2*}
  by(*auto simp add*: *sample-primes-def prime-set-ass finite-prime-set*)

lemma *mem-samp-primes-gt-2*:
  shows *x* ∈ *set-spmf sample-primes* ⟹ *x* > *2*
  apply (*simp add*: *finite-prime-set sample-primes-def*)
  using *prime-set-ass* **by** *blast*

lemma *mem-samp-primes-prime*:
  shows *x* ∈ *set-spmf sample-primes* ⟹ *prime x*
  apply (*simp add*: *finite-prime-set sample-primes-def prime-set-ass*)
  using *prime-set-ass* **by** *blast*

definition *sample-primes-excl* :: *nat set* ⇒ *nat spmf*
  where *sample-primes-excl P* = *spmf-of-set* (*prime-set* − *P*)

lemma *lossless-sample-primes-excl*:
  shows *lossless-spmf* (*sample-primes-excl* {*P*})
  apply(*simp add*: *sample-primes-excl-def finite-prime-set*)
  using *prime-set-gt-2 subset-singletonD* **by** *fastforce*

definition *sample-set-excl* :: *nat set* ⇒ *nat set* ⇒ *nat spmf*
  where *sample-set-excl Q P* = *spmf-of-set* (*Q* − *P*)

lemma *set-spmf-sample-set-excl* [*simp*]:
  assumes *finite* (*Q* − *P*)
  shows *set-spmf* (*sample-set-excl Q P*) = (*Q* − *P*)
  unfolding *sample-set-excl-def*
  by (*metis set-spmf-of-set assms*)+

lemma *lossless-sample-set-excl*:
  assumes *finite Q*
    and *card Q* > *2*
  shows *lossless-spmf* (*sample-set-excl Q* {*P*})
  unfolding *sample-set-excl-def*
  using *assms subset-singletonD* **by** *fastforce*

lemma *mem-samp-primes-excl-gt-2*:
  shows *x* ∈ *set-spmf* (*sample-set-excl prime-set* {*y*}) ⟹ *x* > *2*
  apply(*simp add*: *finite-prime-set sample-set-excl-def prime-set-ass*)
  using *prime-set-ass* **by** *blast*

lemma *mem-samp-primes-excl-prime* :
shows $x \in$ *set-spmf* (*sample-set-excl prime-set* $\{y\}$) $\Longrightarrow$ *prime* x
apply (*simp add*: *finite-prime-set sample-set-excl-def*)
using *prime-set-ass* **by** *blast*

lemma *sample-coprime-lem*:
assumes $x \in$ *set-spmf sample-primes*
and $y \in$ *set-spmf* (*sample-set-excl prime-set* $\{x\}$)
shows *lossless-spmf* (*sample-coprime* $((x - Suc\ 0) * (y - Suc\ 0))$)
proof−
have *gt-2*: $x > 2$ $y > 2$
using *mem-samp-primes-gt-2 assms mem-samp-primes-excl-gt-2* **by** *auto*
have $\neg$ *prime* $((x-1)*(y-1))$
proof−
have *prime x prime y*
using *mem-samp-primes-prime mem-samp-primes-excl-prime assms* **by** *auto*
then show *?thesis* **using** *prod-not-prime gt-2* **by** *simp*
qed
also have $((x-1)*(y-1)) > 2$
by (*metis* (*no-types*, *lifting*) *gt-2 One-nat-def Suc-diff-1 assms*(*1*) *assms*(*2*)
calculation
divisors-zero less-2-cases nat-1-eq-mult-iff nat-neq-iff not-numeral-less-one
numeral-2-eq-2
prime-gt-0-nat rsa-base.mem-samp-primes-excl-prime rsa-base.mem-samp-primes-prime
rsa-base-axioms two-is-prime-nat)
ultimately show *?thesis* **using** *lossless-sample-coprime* **by** *simp*
qed

definition I :: (*index* $\times$ *trap*) *spmf*
where I = *do* {
$P \leftarrow$ *sample-primes*;
$Q \leftarrow$ *sample-set-excl prime-set* $\{P\}$;
let $N = P*Q$;
let $N' = (P-1)*(Q-1)$;
$e \leftarrow$ *sample-coprime* N';
let d = *nat* ((*fst* (*bezw* e N')) *mod* N');
return-spmf $((N, e), d)$}

lemma *lossless-I*: *lossless-spmf I*
by(*auto simp add*: *I-def lossless-sample-primes lossless-sample-set-excl finite-prime-set*
prime-set-gt-2 Let-def sample-coprime-lem)

lemma *set-spmf-I-N*:
assumes $((N,e),d) \in$ *set-spmf I*
obtains P Q **where** $N = P * Q$
and $P \neq Q$
and *prime* P
and *prime* Q

  and *coprime e* ((*P* − *1*)∗(*Q* − *1*))
  and *d* = *nat* (*fst* (*bezw e* ((*P*−*1*)∗(*Q*−*1*))) *mod int* ((*P*−*1*)∗(*Q*−*1*)))
 using *assms* **apply**(*auto simp add*: *I-def Let-def*)
 using *finite-prime-set mem-samp-primes-prime sample-set-excl-def rsa-base-axioms sample-primes-def*
 by (*simp add*: *set-spmf-sample-coprime*)

lemma *set-spmf-I-e-d*:
 ‹*e* > *1*› ‹*d* > *1*› **if** ‹((*N*, *e*), *d*) ∈ *set-spmf I*›
proof −
 from *that* **obtain** *M* **where**
  e: ‹*e* ∈ *set-spmf* (*sample-coprime M*)›
  and *d*: ‹*d* = *nat* (*fst* (*bezw e M*) *mod M*)›
  by (*auto simp add*: *I-def Let-def*)
 from *e set-spmf-sample-coprime* [*of M*]
 have ‹*coprime e M*› ‹*1* < *e*› ‹*e* < *M*›
  by *simp-all*
 then have ‹*2* < *M*›
  by *simp*
 from ‹*1* < *e*› **show** ‹*e* > *1*›**.**
 from *d* ‹*coprime e M*› *bezw-inverse* [*of e M*]
 have *eq1*: ‹[*e* ∗ *d* = *1*] (*mod M*)›
  by *simp*
 with ‹*2* < *M*› **have** *d* ≠ *0*
  by (*metis cong-0-1-nat mult-0-right not-numeral-less-one*)
 moreover have *d* ≠ *1*
  using ‹*1* < *e*› *eq1* ‹*e* < *M*› *cong-less-modulus-unique-nat* **by** *fastforce*
 ultimately show ‹*d* > *1*›
  by *linarith*
qed

definition *domain* :: *index* ⇒ *nat set*
 where *domain index* ≡ {..< *fst index*}

definition *range* :: *index* ⇒ *nat set*
 where *range index* ≡ {..< *fst index*}

lemma *finite-range*: *finite* (*range index*)
 by(*simp add*: *range-def*)

lemma *dom-eq-ran*: *domain index* = *range index*
 by(*simp add*: *range-def domain-def*)

definition *F* :: *index* ⇒ (*nat* ⇒ *nat*)
 where *F index x* = *x* ^ (*snd index*) *mod* (*fst index*)

definition F_{inv} :: *index* ⇒ *trap* ⇒ *nat* ⇒ *nat*
 where F_{inv} α τ *y* = *y* ^ τ *mod* (*fst* α)

We must prove the RSA function is a bijection

```
lemma rsa-bijection:
  assumes coprime: coprime e ((P−1)∗(Q−1))
    and prime-P: prime (P::nat)
    and prime-Q: prime Q
    and P-neq-Q: P ≠ Q
    and x-lt-pq: x < P ∗ Q
    and y-lt-pd: y < P ∗ Q
    and rsa-map-eq: x ^ e mod (P ∗ Q) = y ^ e mod (P ∗ Q)
  shows x = y
proof−
  have flt-xP: [x ^ P = x] (mod P)
    using fermat-little prime-P by blast
  have flt-yP: [y ^ P = y] (mod P)
    using fermat-little prime-P by blast
  have flt-xQ: [x ^ Q = x] (mod Q)
    using fermat-little prime-Q by blast
  have flt-yQ: [y ^ Q = y] (mod Q)
    using fermat-little prime-Q by blast
  show ?thesis
  proof(cases y ≥ x)
    case True
    hence ye-gt-xe: y^e ≥ x^e
      by (simp add: power-mono)
    have x-y-exp-e: [x^e = y^e] (mod P)
      by (metis assms(7) cong-refl mod-mult-cong-right)
    obtain d where d:  [e∗d = 1] (mod (P−1)) ∧ d ≠ 0
      using ex-inverse assms by blast
    then obtain k where k: e∗d = 1 + k∗(P−1)
      using ex-k-mod assms by blast
    hence xk-yk: [x^(1 + k∗(P−1)) = y^(1 + k∗(P−1))] (mod P)
      by(metis k power-mult x-y-exp-e cong-pow)
    have xk-x: [x^(1 + k∗(P−1)) = x] (mod P)
    proof(induct k)
      case 0
      then show ?case by simp
    next
      case (Suc k)
      assume  asm: [x ^(1 + k ∗ (P − 1)) = x] (mod P)
      then show ?case
      proof−
        have exp-rewrite: (k ∗ (P − 1) + P) = (1 + (k + 1) ∗ (P − 1))
       by (smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1
prime-P prime-gt-1-nat semiring-normalization-rules(3))
        have [x ∗ x ^(k ∗ (P − 1)) = x] (mod P) using asm by simp
        hence [x ^(k ∗ (P − 1)) ∗ x ^ P = x] (mod P) using flt-xP
          by (metis cong-scalar-right cong-trans mult.commute)
        hence [x ^(k ∗ (P − 1) + P) = x] (mod P)
          by (simp add: power-add)
        hence [x ^(1 + (k + 1) ∗ (P − 1)) = x] (mod P)
```

using *exp-rewrite* **by** *argo*
thus *?thesis* **by** *simp*
qed
qed
have *yk-y*: $[y\hat{}(1 + k*(P-1)) = y] \ (mod\ P)$
proof(*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
assume *asm*: $[y \hat{} (1 + k * (P - 1)) = y] \ (mod\ P)$
then show *?case*
proof−
have *exp-rewrite*: $(k * (P - 1) + P) = (1 + (k + 1) * (P - 1))$
by (*smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1 prime-P prime-gt-1-nat semiring-normalization-rules*(*3*))
have $[y * y \hat{} (k * (P - 1)) = y] \ (mod\ P)$ **using** *asm* **by** *simp*
hence $[y \hat{} (k * (P - 1)) * y \hat{} P = y] \ (mod\ P)$ **using** *flt-yP*
by (*metis cong-scalar-right cong-trans mult.commute*)
hence $[y \hat{} (k * (P - 1) + P) = y] \ (mod\ P)$
by (*simp add*: *power-add*)
hence $[y \hat{} (1 + (k + 1) * (P - 1)) = y] \ (mod\ P)$
using *exp-rewrite* **by** *argo*
thus *?thesis* **by** *simp*
qed
qed
have $[x\hat{}e = y\hat{}e] \ (mod\ Q)$
by (*metis assms*(*7*) *cong-refl mod-mult-cong-left*)
obtain d' **where** d': $[e*d' = 1] \ (mod\ (Q-1)) \wedge d' \neq 0$
by (*metis mult.commute ex-inverse prime-P prime-Q P-neq-Q coprime*)
then obtain k' **where** k': $e*d' = 1 + k'*(Q-1)$
by(*metis ex-k-mod mult.commute prime-P prime-Q P-neq-Q coprime*)
hence *xk-yk'*: $[x\hat{}(1 + k'*(Q-1)) = y\hat{}(1 + k'*(Q-1))] \ (mod\ Q)$
by(*metis k' power-mult* ‹$[x \hat{} e = y \hat{} e] \ (mod\ Q)$› *cong-pow*)
have *xk-x'*: $[x\hat{}(1 + k'*(Q-1)) = x] \ (mod\ Q)$
proof(*induct k'*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k'*)
assume *asm*: $[x \hat{} (1 + k' * (Q - 1)) = x] \ (mod\ Q)$
then show *?case*
proof−
have *exp-rewrite*: $(k' * (Q - 1) + Q) = (1 + (k' + 1) * (Q - 1))$
by (*smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1 prime-Q prime-gt-1-nat semiring-normalization-rules*(*3*))
have $[x * x \hat{} (k' * (Q - 1)) = x] \ (mod\ Q)$ **using** *asm* **by** *simp*
hence $[x \hat{} (k' * (Q - 1)) * x \hat{} Q = x] \ (mod\ Q)$ **using** *flt-xQ*
by (*metis cong-scalar-right cong-trans mult.commute*)

```
      hence [x ^ (k' * (Q − 1) + Q) = x] (mod Q)
        by (simp add: power-add)
      hence [x ^ (1 + (k' + 1) * (Q − 1)) = x] (mod Q)
        using exp-rewrite by argo
      thus ?thesis by simp
    qed
  qed
  have yk-y': [y^(1 + k'*(Q−1)) = y] (mod Q)
  proof(induct k')
    case 0
    then show ?case by simp
  next
    case (Suc k')
    assume asm: [y ^ (1 + k' * (Q − 1)) = y] (mod Q)
    then show ?case
    proof−
      have exp-rewrite: (k' * (Q − 1) + Q) = (1 + (k' + 1) * (Q − 1))
      by (smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1
prime-Q prime-gt-1-nat semiring-normalization-rules(3))
      have [y * y ^ (k' * (Q − 1)) = y] (mod Q) using asm by simp
      hence [y ^ (k' * (Q − 1)) * y ^ Q = y] (mod Q) using flt-yQ
        by (metis cong-scalar-right cong-trans mult.commute)
      hence [y ^ (k' * (Q − 1) + Q) = y] (mod Q)
        by (simp add: power-add)
      hence [y ^ (1 + (k' + 1) * (Q − 1)) = y] (mod Q)
        using exp-rewrite by argo
      thus ?thesis by simp
    qed
  qed
  have P-dvd-xy: P dvd (y − x)
  proof−
    have [x = y] (mod P)
      by (meson cong-sym-eq cong-trans xk-x xk-yk yk-y)
  thus ?thesis
    using cong-altdef-nat cong-sym True by blast
  qed
  have Q-dvd-xy: Q dvd (y − x)
  proof−
    have [x = y] (mod Q)
      by (meson cong-sym cong-trans xk-x' xk-yk' yk-y')
  thus ?thesis
    using cong-altdef-nat cong-sym True by blast
  qed
  show ?thesis
  proof−
  have P*Q dvd (y − x) using P-dvd-xy  Q-dvd-xy
    by (simp add: assms divides-mult primes-coprime)
  then have [x = y] (mod P*Q)
    by (simp add: cong-altdef-nat cong-sym True)
```

hence *x mod P∗Q = y mod P∗Q*
using *cong-less-modulus-unique-nat x-lt-pq y-lt-pd* **by** *blast*
then show *?thesis*
using ‹[*x* = *y*] (*mod P* ∗ *Q*)› *cong-less-modulus-unique-nat x-lt-pq y-lt-pd* **by** *blast*
qed
next
case *False*
hence *ye-gt-xe*: $x\hat{}e \geq y\hat{}e$
by (*simp add*: *power-mono*)
have *pow-eq*: [$x\hat{}e = y\hat{}e$] (*mod P∗Q*)
using *rsa-map-eq unique-euclidean-semiring-class.cong-def* **by** *blast*
then have *PQ-dvd-ye-xe*: (*P∗Q*) *dvd* ($x\hat{}e - y\hat{}e$)
using *cong-altdef-nat False ye-gt-xe cong-sym* **by** *blast*
then have [$x\hat{}e = y\hat{}e$] (*mod P*)
using *cong-modulus-mult-nat pow-eq* **by** *blast*
obtain *d* **where** *d*: [*e∗d* = *1*] (*mod* (*P*−*1*)) ∧ *d* ≠ *0* **using** *ex-inverse assms* **by** *blast*
then obtain *k* **where** *k*: *e∗d* = *1* + *k∗*(*P*−*1*) **using** *ex-k-mod assms* **by** *blast*
have *xk-yk*: [$x\hat{}(1 + k*(P-1)) = y\hat{}(1 + k*(P-1))$] (*mod P*)
proof−
have [$(x\hat{}e)\hat{}d = (y\hat{}e)\hat{}d$] (*mod P*)
using ‹[$x \hat{} e = y \hat{} e$] (*mod P*)› *cong-pow* **by** *blast*
then have [$x\hat{}(e*d) = y\hat{}(e*d)$] (*mod P*)
by (*simp add*: *power-mult*)
thus *?thesis* **using** *k* **by** *simp*
qed
have *xk-x*: [$x\hat{}(1 + k*(P-1)) = x$] (*mod P*)
proof(*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
assume *asm*: [$x \hat{} (1 + k * (P - 1)) = x$] (*mod P*)
then show *?case*
proof−
have *exp-rewrite*: $(k * (P - 1) + P) = (1 + (k + 1) * (P - 1))$
by (*smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1 prime-P prime-gt-1-nat semiring-normalization-rules*(*3*))
have [$x * x \hat{} (k * (P - 1)) = x$] (*mod P*) **using** *asm* **by** *simp*
hence [$x \hat{} (k * (P - 1)) * x \hat{} P = x$] (*mod P*) **using** *flt-xP*
by (*metis cong-scalar-right cong-trans mult.commute*)
hence [$x \hat{} (k * (P - 1) + P) = x$] (*mod P*)
by (*simp add*: *power-add*)
hence [$x \hat{} (1 + (k + 1) * (P - 1)) = x$] (*mod P*)
using *exp-rewrite* **by** *argo*
thus *?thesis* **by** *simp*
qed
qed

have *yk-y*: [y^(1 + k*(P−1)) = y] (*mod P*)
proof(*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
assume *asm*: [y ^(1 + k * (P − 1)) = y] (*mod P*)
then show *?case*
proof−
have *exp-rewrite*: (k * (P − 1) + P) = (1 + (k + 1) * (P − 1))
by (*smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1 prime-P prime-gt-1-nat semiring-normalization-rules*(*3*))
have [y * y ^(k * (P − 1)) = y] (*mod P*) **using** *asm* **by** *simp*
hence [y ^(k * (P − 1)) * y ^P = y] (*mod P*) **using** *flt-yP*
by (*metis cong-scalar-right cong-trans mult.commute*)
hence [y ^(k * (P − 1) + P) = y] (*mod P*)
by (*simp add*: *power-add*)
hence [y ^(1 + (k + 1) * (P − 1)) = y] (*mod P*)
using *exp-rewrite* **by** *argo*
thus *?thesis* **by** *simp*
qed
qed
have *P-dvd-xy*: *P dvd* (*x* − *y*)
proof−
have [*x* = *y*] (*mod P*)
by (*meson cong-sym-eq cong-trans xk-x xk-yk yk-y*)
thus *?thesis*
using *cong-altdef-nat cong-sym False* **by** *simp*
qed
have [x^e = y^e] (*mod Q*)
using *cong-modulus-mult-nat pow-eq PQ-dvd-ye-xe cong-dvd-modulus-nat dvd-triv-right* **by** *blast*
obtain d' **where** d': [e*d' = 1] (*mod* (Q−1)) ∧ $d' \neq 0$
by (*metis mult.commute ex-inverse prime-P prime-Q coprime P-neq-Q*)
then obtain k' **where** k': e*d' = 1 + k'*(Q−1)
by(*metis ex-k-mod mult.commute prime-P prime-Q coprime P-neq-Q*)
have *xk-yk'*: [x^(1 + k'*(Q−1)) = y^(1 + k'*(Q−1))] (*mod Q*)
proof−
have [(x^e)^d' = (y^e)^d'] (*mod Q*)
using ‹[x ^ e = y ^ e] (*mod Q*)› *cong-pow* **by** *blast*
then have [x^(e*d') = y^(e*d')] (*mod Q*)
by (*simp add*: *power-mult*)
thus *?thesis* **using** k'
by *simp*
qed
have *xk-x'*: [x^(1 + k'*(Q−1)) = x] (*mod Q*)
proof(*induct* k')
case *0*
then show *?case* **by** *simp*

next
case (*Suc* k')
assume *asm*: $[x \,\hat{}\, (1 + k' * (Q - 1)) = x]$ $(mod\ Q)$
then show *?case*
proof−
have *exp-rewrite*: $(k' * (Q - 1) + Q) = (1 + (k' + 1) * (Q - 1))$
by (*smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1 prime-Q prime-gt-1-nat semiring-normalization-rules*(*3*))
have $[x * x \,\hat{}\, (k' * (Q - 1)) = x]$ $(mod\ Q)$ **using** *asm* **by** *simp*
hence $[x \,\hat{}\, (k' * (Q - 1)) * x \,\hat{}\, Q = x]$ $(mod\ Q)$ **using** *flt-xQ*
by (*metis cong-scalar-right cong-trans mult.commute*)
hence $[x \,\hat{}\, (k' * (Q - 1) + Q) = x]$ $(mod\ Q)$
by (*simp add*: *power-add*)
hence $[x \,\hat{}\, (1 + (k' + 1) * (Q - 1)) = x]$ $(mod\ Q)$
using *exp-rewrite* **by** *argo*
thus *?thesis* **by** *simp*
qed
qed
have *yk-y'*: $[y\hat{}(1 + k'*(Q-1)) = y]$ $(mod\ Q)$
proof(*induct* k')
case *0*
then show *?case* **by** *simp*
next
case (*Suc* k')
assume *asm*: $[y \,\hat{}\, (1 + k' * (Q - 1)) = y]$ $(mod\ Q)$
then show *?case*
proof−
have *exp-rewrite*: $(k' * (Q - 1) + Q) = (1 + (k' + 1) * (Q - 1))$
by (*smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1 prime-Q prime-gt-1-nat semiring-normalization-rules*(*3*))
have $[y * y \,\hat{}\, (k' * (Q - 1)) = y]$ $(mod\ Q)$ **using** *asm* **by** *simp*
hence $[y \,\hat{}\, (k' * (Q - 1)) * y \,\hat{}\, Q = y]$ $(mod\ Q)$ **using** *flt-yQ*
by (*metis cong-scalar-right cong-trans mult.commute*)
hence $[y \,\hat{}\, (k' * (Q - 1) + Q) = y]$ $(mod\ Q)$
by (*simp add*: *power-add*)
hence $[y \,\hat{}\, (1 + (k' + 1) * (Q - 1)) = y]$ $(mod\ Q)$
using *exp-rewrite* **by** *argo*
thus *?thesis* **by** *simp*
qed
qed
have *Q-dvd-xy*: Q *dvd* $(x - y)$
proof−
have $[x = y]$ $(mod\ Q)$
by (*meson cong-sym-eq cong-trans xk-x' xk-yk' yk-y'*)
thus *?thesis*
using *cong-altdef-nat cong-sym False* **by** *simp*
qed
show *?thesis*
proof−

```
      have P∗Q dvd (x − y)
       using P-dvd-xy Q-dvd-xy by (simp add: assms divides-mult primes-coprime)
      hence 1: [x = y] (mod P∗Q)
        using False cong-altdef-nat linear by blast
      hence x mod P∗Q = y mod P∗Q
        using cong-less-modulus-unique-nat x-lt-pq y-lt-pd by blast
      thus ?thesis
        using 1 cong-less-modulus-unique-nat x-lt-pq y-lt-pd by blast
    qed
  qed
qed

lemma rsa-bij-betw:
  assumes coprime e ((P − 1)∗(Q − 1))
    and prime P
    and prime Q
    and P ≠ Q
  shows bij-betw (F ((P ∗ Q), e)) (range ((P ∗ Q), e)) (range ((P ∗ Q), e))
proof−
  have PQ-not-0: prime P ⟶ prime Q ⟶ P ∗ Q ≠ 0
  using assms by auto
  have inj-on (λx. x ^ snd (P ∗ Q, e) mod fst (P ∗ Q, e)) {..<fst (P ∗ Q, e)}
    apply(simp add: inj-on-def)
    using rsa-bijection assms by blast
  moreover have (λx. x ^ snd (P ∗ Q, e) mod fst (P ∗ Q, e)) ‘ {..<fst (P ∗ Q,
e)} = {..<fst (P ∗ Q, e)}
    apply(simp add: assms(2) assms(3) prime-gt-0-nat PQ-not-0)
     apply(rule endo-inj-surj; auto simp add: assms(2) assms(3) image-subsetI
prime-gt-0-nat PQ-not-0 inj-on-def)
    using rsa-bijection assms by blast
  ultimately show ?thesis
  unfolding bij-betw-def F-def range-def by blast
qed

lemma bij-betw1:
  assumes ((N,e),d) ∈ set-spmf I
  shows bij-betw (F ((N), e)) (range ((N), e)) (range ((N), e))
proof−
  obtain P Q where N = P ∗ Q and bij-betw (F ((P∗Q), e)) (range ((P∗Q), e))
(range ((P∗Q), e))
  proof−
    obtain P Q where prime P and prime Q and N = P ∗ Q and P ≠ Q and
coprime e ((P − 1)∗(Q − 1))
      using set-spmf-I-N assms by blast
    then show ?thesis
      using rsa-bij-betw that by blast
  qed
  thus ?thesis by blast
qed
```

```
lemma rsa-inv:
  assumes d: d = nat (fst (bezw e ((P−1)∗(Q−1))) mod int ((P−1)∗(Q−1)))
    and coprime: coprime e ((P−1)∗(Q−1))
    and prime-P: prime (P::nat)
    and prime-Q: prime Q
    and P-neq-Q: P ≠ Q
    and e-gt-1: e > 1
    and d-gt-1: d > 1
  shows ((((x) ^ e) mod (P∗Q)) ^ d) mod (P∗Q) = x mod (P∗Q)
proof(cases x = 0 ∨ x = 1)
  case True
  then show ?thesis
    by (metis assms(6) assms(7) le-simps(1) nat-power-eq-Suc-0-iff neq0-conv
not-one-le-zero numeral-nat(7) power-eq-0-iff power-mod)
next
  case False
  hence x-gt-1: x > 1 by simp
  define z where z = (x ^ e) ^ d − x
  hence z-gt-0: z > 0
  proof−
    have (x ^ e) ^ d − x = x ^(e ∗ d) − x
      by (simp add: power-mult)
    also have ... > 0
      by (metis x-gt-1 e-gt-1 d-gt-1 le-neq-implies-less less-one linorder-not-less
n-less-m-mult-n not-less-zero numeral-nat(7) power-increasing-iff power-one-right
zero-less-diff)
    ultimately show ?thesis using z-def by argo
  qed
  hence [z = 0] (mod P)
  proof(cases [x = 0] (mod P))
    case True
    then show ?thesis
     by (metis Suc-lessD e-gt-1 d-gt-1 cong-0-iff dvd-minus-self dvd-power dvd-trans
One-nat-def z-def)
  next
    case False
    have [e ∗ d = 1] (mod ((P − 1) ∗ (Q − 1)))
      by (metis d bezw-inverse coprime coprime-imp-gcd-eq-1 nat-int)
    hence [e ∗ d = 1] (mod (P − 1))
      using assms cong-modulus-mult-nat by blast
    then obtain k where k: e∗d = 1 + k∗(P−1)
      using ex-k-mod assms by force
    hence x ^(e ∗ d) = x ∗ ((x ^(P − 1)) ^ k)
      by (metis power-add power-one-right mult.commute power-mult)
    hence [x ^(e ∗ d) = x ∗ ((x ^(P − 1)) ^ k)] (mod P)
      using cong-def by simp
    moreover have [x ^(P − 1) = 1] (mod P)
      using prime-P fermat-theorem False
```

```
      by (simp add: cong-0-iff)
    moreover have [x ^ (e ∗ d) = x ∗ ((1) ^ k)] (mod P)
      by (metis ‹x ^ (e ∗ d) = x ∗ (x ^ (P − 1)) ^ k› calculation(2) cong-pow
cong-scalar-left)
    hence [x ^ (e ∗ d) = x] (mod P) by simp
    thus ?thesis using z-def z-gt-0
      by (simp add: cong-diff-iff-cong-0-nat power-mult)
  qed
  moreover have [z = 0] (mod Q)
  proof(cases [x = 0] (mod Q))
    case True
    then show ?thesis
      by (metis cong-0-iff cong-modulus-mult-nat dvd-def dvd-minus-self power-eq-if
power-mult x-gt-1 z-def)
  next
    case False
    have [e ∗ d = 1] (mod ((P − 1) ∗ (Q − 1)))
      by (metis d bezw-inverse coprime coprime-imp-gcd-eq-1 nat-int)
    hence [e ∗ d = 1] (mod (Q − 1))
      using assms cong-modulus-mult-nat mult.commute by metis
    then obtain k where k: e∗d = 1 + k∗(Q−1)
      using ex-k-mod assms by force
    hence x ^ (e ∗ d) = x ∗ ((x ^ (Q − 1)) ^ k)
      by (metis power-add power-one-right mult.commute power-mult)
    hence [x ^ (e ∗ d) = x ∗ ((x ^ (Q − 1)) ^ k)] (mod P)
      using cong-def by simp
    moreover have [x ^ (Q − 1) = 1] (mod Q)
        using prime-Q fermat-theorem False
        by (simp add: cong-0-iff)
    moreover have [x ^ (e ∗ d) = x ∗ ((1) ^ k)] (mod Q)
      by (metis ‹x ^ (e ∗ d) = x ∗ (x ^ (Q − 1)) ^ k› calculation(2) cong-pow
cong-scalar-left)
    hence [x ^ (e ∗ d) = x] (mod Q) by simp
    thus ?thesis using z-def z-gt-0
      by (simp add: cong-diff-iff-cong-0-nat power-mult)
  qed
  ultimately have Q dvd (x ^ e) ^ d − x
                  P dvd (x ^ e) ^ d − x
    using z-def assms cong-0-iff by blast +
  hence P ∗ Q dvd ((x ^ e) ^ d − x)
    using assms divides-mult primes-coprime-nat by blast
  hence [(x ^ e) ^ d = x] (mod (P ∗ Q))
    using z-gt-0 cong-altdef-nat z-def by auto
  thus ?thesis
    by (simp add: unique-euclidean-semiring-class.cong-def power-mod)
qed
```

lemma *rsa-inv-set-spmf-I*:

assumes $((N, e), d) \in$ *set-spmf I*
shows $((((x{::}nat) \mathbin{\char`^} e)\ mod\ N) \mathbin{\char`^} d)\ mod\ N = x\ mod\ N$
proof−
obtain *P Q* **where** $N = P * Q$ **and** $d = nat\ (fst\ (bezw\ e\ ((P-1)*(Q-1)))\ mod\ int\ ((P-1)*(Q-1)))$
and *prime P*
and *prime Q*
and *coprime e* $((P - 1)*(Q - 1))$
and $P \neq Q$
using *assms set-spmf-I-N*
by *blast*
moreover have $e > 1$ **and** $d > 1$ **using** *set-spmf-I-e-d assms* **by** *auto*
ultimately show *?thesis* **using** *rsa-inv* **by** *blast*
qed

sublocale *etp-rsa*: *etp I domain range F* F_{inv}
unfolding *etp-def* **apply**(*auto simp add*: *etp-def dom-eq-ran finite-range bij-betw1 lossless-I*)
apply (*metis fst-conv lessThan-iff mem-simps*(*2*) *nat-0-less-mult-iff prime-gt-0-nat range-def set-spmf-I-N*)
apply(*auto simp add*: *F-def* F_{inv}*-def*) **using** *rsa-inv-set-spmf-I*
by (*simp add*: *range-def*)

sublocale *etp*: *ETP-base I domain range B F* F_{inv}
unfolding *ETP-base-def*
by (*simp add*: *etp-rsa.etp-axioms*)

After proving the RSA collection is an ETP the proofs of security come easily from the general proofs.

lemma *correctness-rsa*: *etp.OT-12.correctness m1 m2*
by (*rule local.etp.correct*)

lemma *P1-security-rsa*: *etp.OT-12.perfect-sec-P1 m1 m2*
by(*rule local.etp.P1-security-inf-the*)

lemma *P2-security-rsa*:
assumes $\forall$ *a. lossless-spmf* (*D a*)
and $\bigwedge b_\sigma$. *local.etp-rsa.HCP-adv etp.*$\mathcal{A}$ *m2* b_σ *D* $\leq$ *HCP-ad*
shows *etp.OT-12.adv-P2 m1 m2 D* $\leq 2 *$ *HCP-ad*
by(*simp add*: *local.etp.P2-security assms*)

end

locale *rsa-asym* =
fixes *prime-set* :: *nat* $\Rightarrow$ *nat set*
and *B* :: *index* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
assumes *rsa-proof-assm*: $\bigwedge$ *n. rsa-base* (*prime-set n*)
begin

sublocale *rsa-base* (*prime-set n*) *B*
  using *local.rsa-proof-assm* **by** *simp*

lemma *correctness-rsa-asymp*:
  shows *etp.OT-12.correctness n m1 m2*
  by(*rule correctness-rsa*)

lemma *P1-sec-asymp*: *etp.OT-12.perfect-sec-P1 n m1 m2*
  by(*rule local.P1-security-rsa*)

lemma *P2-sec-asym*:
  assumes ∀ *a. lossless-spmf* (*D a*)
    and *HCP-adv-neg*: *negligible* (λ *n. hcp-advantage n*)
   and *hcp-adv-bound*: ∀ b_σ *n. local.etp-rsa.HCP-adv n etp.*$\mathcal{A}$ *m2* b_σ *D* ≤ *hcp-advantage n*
  shows *negligible* (λ *n. etp.OT-12.adv-P2 n m1 m2 D*)
proof−
  have *negligible* (λ *n. 2* ∗ *hcp-advantage n*) **using** *HCP-adv-neg*
    by (*simp add*: *negligible-cmultI*)
  moreover have |*etp.OT-12.adv-P2 n m1 m2 D*| = *etp.OT-12.adv-P2 n m1 m2 D*
   for *n* **unfolding** *sim-det-def.adv-P2-def local.etp.OT-12.adv-P2-def* **by** *linarith*
  moreover have *etp.OT-12.adv-P2 n m1 m2 D* ≤ *2* ∗ *hcp-advantage n* **for** *n*
    using *P2-security-rsa assms* **by** *blast*
  ultimately show *?thesis*
    using *assms negligible-le* **by** *presburger*
qed

end

end

2.5 Noar Pinkas OT

Here we prove security for the Noar Pinkas OT from [7].

theory *Noar-Pinkas-OT* **imports**
  Cyclic-Group-Ext
  Game-Based-Crypto.Diffie-Hellman
  OT-Functionalities
  Semi-Honest-Def
  Uniform-Sampling
begin

locale *np-base* =
  fixes $\mathcal{G}$ :: *'grp cyclic-group* (**structure**)
  assumes *finite-group*: *finite* (*carrier* $\mathcal{G}$)
    and *or-gt-0*: *0* < *order* $\mathcal{G}$
    and *prime-order*: *prime* (*order* $\mathcal{G}$)
begin

lemma *prime-field*: $a < (order\ \mathcal{G}) \Longrightarrow a \neq 0 \Longrightarrow coprime\ a\ (order\ \mathcal{G})$
by(*metis dvd-imp-le neq0-conv not-le prime-imp-coprime prime-order coprime-commute*)

lemma *weight-sample-uniform-units*: *weight-spmf* (*sample-uniform-units* (*order* $\mathcal{G}$)) = *1*
using *lossless-spmf-def lossless-sample-uniform-units prime-order prime-gt-1-nat*
by *auto*

definition *protocol* :: ('*grp* × '*grp*) ⇒ *bool* ⇒ (*unit* × '*grp*) *spmf*
where *protocol M v* = *do* {
let (*m0*,*m1*) = *M*;
a :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
b :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
let c_v = (*a*∗*b*) *mod* (*order* $\mathcal{G}$);
c_v' :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
r0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w0 = (**g** [^] *a*) [^] *s0* ⊗ **g** [^] *r0*;
let z0' = ((**g** [^] (*if v then* c_v' *else* c_v)) [^] *s0*) ⊗ ((**g** [^] *b*) [^] *r0*);
r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w1 = (**g** [^] *a*) [^] *s1* ⊗ **g** [^] *r1*;
let z1' = ((**g** [^] ((*if v then* c_v *else* c_v'))) [^] *s1*) ⊗ ((**g** [^] *b*) [^] *r1*);
let enc-m0 = *z0'* ⊗ *m0*;
let enc-m1 = *z1'* ⊗ *m1*;
let out-2 = (*if v then enc-m1* ⊗ *inv* (*w1* [^] *b*) *else enc-m0* ⊗ *inv* (*w0* [^] *b*));
return-spmf ((), *out-2*)}

lemma *lossless-protocol*: *lossless-spmf* (*protocol M* σ)
apply(*auto simp add*: *protocol-def Let-def split-def lossless-sample-uniform-units or-gt-0*)
using *prime-order prime-gt-1-nat lossless-sample-uniform-units* **by** *simp*

type-synonym '*grp*' *view1* = (('*grp*' × '*grp*') × ('*grp*' × '*grp*' × '*grp*' × '*grp*')) *spmf*

type-synonym '*grp*' *dist-adversary* = (('*grp*' × '*grp*') × '*grp*' × '*grp*' × '*grp*' × '*grp*') ⇒ *bool spmf*

definition *R1* :: ('*grp* × '*grp*) ⇒ *bool* ⇒ '*grp view1*
where *R1 msgs* σ = *do* {
let (*m0*, *m1*) = *msgs*;
a ← *sample-uniform* (*order* $\mathcal{G}$);
b ← *sample-uniform* (*order* $\mathcal{G}$);
let c_σ = *a*∗*b*;
c_σ' ← *sample-uniform* (*order* $\mathcal{G}$);
return-spmf (*msgs*, (**g** [^] *a*, **g** [^] *b*, (*if* σ *then* **g** [^] c_σ' *else* **g** [^] c_σ), (*if* σ *then* **g** [^] c_σ *else* **g** [^] c_σ')))}

lemma *lossless-R1*: *lossless-spmf* (*R1 M σ*)
  by(*simp add*: *R1-def Let-def lossless-sample-uniform-units or-gt-0*)

definition *inter* :: ('*grp* × '*grp*) ⇒ '*grp view1*
  where *inter msgs* = *do* {
    a ← *sample-uniform* (*order* $\mathcal{G}$);
    b ← *sample-uniform* (*order* $\mathcal{G}$);
    c ← *sample-uniform* (*order* $\mathcal{G}$);
    d ← *sample-uniform* (*order* $\mathcal{G}$);
    return-spmf (*msgs*, **g** [^] *a*, **g** [^] *b*, **g** [^] *c*, **g** [^] *d*)}

definition *S1* :: ('*grp* × '*grp*) ⇒ *unit* ⇒ '*grp view1*
  where *S1 msgs out1* = *do* {
    let (*m0*, *m1*) = *msgs*;
    a ← *sample-uniform* (*order* $\mathcal{G}$);
    b ← *sample-uniform* (*order* $\mathcal{G}$);
    c ← *sample-uniform* (*order* $\mathcal{G}$);
    return-spmf (*msgs*, (**g** [^] *a*, **g** [^] *b*, **g** [^] *c*, **g** [^] (*a*∗*b*)))}

lemma *lossless-S1*: *lossless-spmf* (*S1 M out1*)
  by(*simp add*: *S1-def Let-def lossless-sample-uniform-units or-gt-0*)

fun *R1-inter-adversary* :: '*grp dist-adversary* ⇒ ('*grp* × '*grp*) ⇒ '*grp* ⇒ '*grp* ⇒ '*grp* ⇒ *bool spmf*
  where *R1-inter-adversary* $\mathcal{A}$ *msgs α β γ* = *do* {
    c ← *sample-uniform* (*order* $\mathcal{G}$);
    $\mathcal{A}$ (*msgs*, *α*, *β*, *γ*, **g** [^] *c*)}

fun *inter-S1-adversary* :: '*grp dist-adversary* ⇒ ('*grp* × '*grp*) ⇒ '*grp* ⇒ '*grp* ⇒ '*grp* ⇒ *bool spmf*
  where *inter-S1-adversary* $\mathcal{A}$ *msgs α β γ* = *do* {
    c ← *sample-uniform* (*order* $\mathcal{G}$);
    $\mathcal{A}$ (*msgs*, *α*, *β*, **g** [^] *c*, *γ*)}

sublocale *ddh*: *ddh* $\mathcal{G}$ **.**

definition *R2* :: ('*grp* × '*grp*) ⇒ *bool* ⇒ (*bool* × '*grp* × '*grp* × '*grp* × '*grp* × '*grp* × '*grp* × '*grp*) *spmf*
  where *R2 M v* = *do* {
  let (*m0*,*m1*) = *M*;
  a :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
  b :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
  let c_v = (*a*∗*b*) *mod* (*order* $\mathcal{G}$);
  c_v' :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
  r0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
  s0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
  let w0 = (**g** [^] *a*) [^] *s0* ⊗ **g** [^] *r0*;
  let z = ((**g** [^] c_v') [^] *s0*) ⊗ ((**g** [^] *b*) [^] *r0*);

r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w1 = (**g** [⌉] *a*) [⌉] *s1* ⊗ **g** [⌉] *r1*;
let z′ = ((**g** [⌉] (c_v)) [⌉] *s1*) ⊗ ((**g** [⌉] *b*) [⌉] *r1*);
let enc-m = *z* ⊗ (*if v then m0 else m1*);
let enc-m′ = *z′* ⊗ (*if v then m1 else m0*) ;
return-spmf(*v*, **g** [⌉] *a*, **g** [⌉] *b*, **g** [⌉] c_v, *w0*, *enc-m*, *w1*, *enc-m′*)}

lemma *lossless-R2*: *lossless-spmf* (*R2 M σ*)
apply(*simp add*: *R2-def Let-def split-def lossless-sample-uniform-units or-gt-0*)
using *prime-order prime-gt-1-nat lossless-sample-uniform-units* **by** *simp*

definition *S2* :: *bool* ⇒ *′grp* ⇒ (*bool* × *′grp* × *′grp* × *′grp* × *′grp* × *′grp* × *′grp* × *′grp*) *spmf*
where *S2 v m* = *do* {
a :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
b :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
let c_v = (*a*∗*b*) *mod* (*order* $\mathcal{G}$);
r0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w0 = (**g** [⌉] *a*) [⌉] *s0* ⊗ **g** [⌉] *r0*;
r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w1 = (**g** [⌉] *a*) [⌉] *s1* ⊗ **g** [⌉] *r1*;
let z′ = ((**g** [⌉] (c_v)) [⌉] *s1*) ⊗ ((**g** [⌉] *b*) [⌉] *r1*);
s′ ← *sample-uniform* (*order* $\mathcal{G}$);
let enc-m = **g** [⌉] *s′*;
let enc-m′ = *z′* ⊗ *m* ;
return-spmf(*v*, **g** [⌉] *a*, **g** [⌉] *b*, **g** [⌉] c_v, *w0*, *enc-m*, *w1*, *enc-m′*)}

lemma *lossless-S2*: *lossless-spmf* (*S2 σ out2*)
apply(*simp add*: *S2-def Let-def lossless-sample-uniform-units or-gt-0*)
using *prime-order prime-gt-1-nat lossless-sample-uniform-units* **by** *simp*

sublocale *sim-def*: *sim-det-def R1 S1 R2 S2 funct-OT-12 protocol*
unfolding *sim-det-def-def*
by(*auto simp add*: *lossless-R1 lossless-S1 lossless-R2 lossless-S2 lossless-protocol lossless-funct-OT-12*)

end

locale *np* = *np-base* + *cyclic-group* $\mathcal{G}$
begin

lemma *protocol-inverse*:
assumes *m0* ∈ *carrier* $\mathcal{G}$ *m1* ∈ *carrier* $\mathcal{G}$
shows ((**g** [⌉] ((*a*∗*b*) *mod* (*order* $\mathcal{G}$))) [⌉] (*s1* :: *nat*)) ⊗ ((**g** [⌉] *b*) [⌉] (*r1*::*nat*)) ⊗ (*if v then m0 else m1*) ⊗ *inv* (((**g** [⌉] *a*) [⌉] *s1* ⊗ **g** [⌉] *r1*) [⌉] *b*)
= (*if v then m0 else m1*)

(**is** *?lhs* = *?rhs*)
proof−
have *1*: (*a*∗*b*)∗(*s1*) + *b*∗*r1* =((*a*::*nat*)∗(*s1*) + *r1*)∗*b* **using** *mult.commute mult.assoc add-mult-distrib* **by** *auto*
have *?lhs* =
((**g** [^] (*a*∗*b*)) [^] *s1*) ⊗ ((**g** [^] *b*) [^] *r1*) ⊗ (*if v then m0 else m1*) ⊗ *inv* (((**g** [^] *a*) [^] *s1* ⊗ **g** [^] *r1*) [^] *b*)
by(*simp add*: *pow-generator-mod*)
also have ... = (**g** [^] ((*a*∗*b*)∗(*s1*))) ⊗ ((**g** [^] (*b*∗*r1*))) ⊗ ((*if v then m0 else m1*) ⊗ *inv* (((**g** [^] ((*a*∗(*s1*) + *r1*)∗*b*)))))
by(*auto simp add*: *nat-pow-pow nat-pow-mult assms cyclic-group-assoc*)
also have ... = **g** [^] ((*a*∗*b*)∗(*s1*)) ⊗ **g** [^] (*b*∗*r1*) ⊗ ((*inv* (((**g** [^] ((*a*∗(*s1*) + *r1*)∗*b*))))) ⊗ (*if v then m0 else m1*))
by(*simp add*: *nat-pow-mult cyclic-group-commute assms*)
also have ... = (**g** [^] ((*a*∗*b*)∗(*s1*) + *b*∗*r1*) ⊗ *inv* (((**g** [^] ((*a*∗(*s1*) + *r1*)∗*b*))))) ⊗ (*if v then m0 else m1*)
by(*simp add*: *nat-pow-mult cyclic-group-assoc assms*)
also have ... = (**g** [^] ((*a*∗*b*)∗(*s1*) + *b*∗*r1*) ⊗ *inv* (((**g** [^] (((*a*∗*b*)∗(*s1*) + *r1*∗*b*)))))) ⊗ (*if v then m0 else m1*)
using *1* **by** (*simp add*: *mult.commute*)
ultimately show *?thesis*
using *l-cancel-inv assms* **by** (*simp add*: *mult.commute*)
qed

lemma *correctness*:
assumes *m0* ∈ *carrier* $\mathcal{G}$ *m1* ∈ *carrier* $\mathcal{G}$
shows *sim-def.correctness* (*m0*,*m1*) *σ*
proof−
have *protocol* (*m0*, *m1*) *σ* = *funct-OT-12* (*m0*, *m1*) *σ*
proof−
have *protocol* (*m0*, *m1*) *σ* = *do* {
a :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
b :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let out-2 = ((**g** [^] ((*a*∗*b*) *mod* (*order* $\mathcal{G}$))) [^] *s1*) ⊗ ((**g** [^] *b*) [^] *r1*) ⊗ (*if σ then m1 else m0*) ⊗ *inv* (((**g** [^] *a*) [^] *s1* ⊗ **g** [^] *r1*) [^] *b*);
return-spmf ((), *out-2*)}
by(*simp add*: *protocol-def lossless-sample-uniform-units bind-spmf-const weight-sample-uniform-units or-gt-0*)
also have ... = *do* {
let out-2 = (*if σ then m1 else m0*);
return-spmf ((), *out-2*)}
by(*simp add*: *protocol-inverse assms lossless-sample-uniform-units bind-spmf-const weight-sample-uniform-units or-gt-0*)
ultimately show *?thesis* **by**(*simp add*: *Let-def funct-OT-12-def*)
qed
thus *?thesis*
by(*simp add*: *sim-def.correctness-def*)

qed

lemma *security-P1*:
  shows *sim-def.adv-P1 msgs* σ *D* $\leq$ *ddh.advantage* (*R1-inter-adversary D msgs*) + *ddh.advantage* (*inter-S1-adversary D msgs*)
    (**is** *?lhs* $\leq$ *?rhs*)
proof(*cases* σ)
  case *True*
  have *R1 msgs* σ = *S1 msgs out1* **for** *out1*
    by(*simp add*: *R1-def S1-def True*)
  then have *sim-def.adv-P1 msgs* σ *D* = *0*
    by(*simp add*: *sim-def.adv-P1-def funct-OT-12-def*)
  also have *ddh.advantage A* $\geq$ *0* **for** *A* **using** *ddh.advantage-def* **by** *simp*
  ultimately show *?thesis* **by** *simp*
next
  case *False*
  have *bounded-advantage*: $|(a :: real) - b| = e1 \Longrightarrow |b - c| = e2 \Longrightarrow |a - c| \leq e1 + e2$
    for *a b e1 c e2* **by** *simp*
  also have *R1-inter-dist*: |*spmf* (*R1 msgs False* ⪢= *D*) *True* − *spmf* ((*inter msgs*) ⪢= *D*) *True*| = *ddh.advantage* (*R1-inter-adversary D msgs*)
    unfolding *R1-def inter-def ddh.advantage-def ddh.ddh-0-def ddh.ddh-1-def Let-def split-def* **by**(*simp*)
  also have *inter-S1-dist*: |*spmf* ((*inter msgs*) ⪢= *D*) *True* − *spmf* (*S1 msgs out1* ⪢= *D*) *True*| = *ddh.advantage* (*inter-S1-adversary D msgs*)
    for *out1* **including** *monad-normalisation*
    by(*simp add*: *S1-def inter-def ddh.advantage-def ddh.ddh-0-def ddh.ddh-1-def*)
  ultimately have |*spmf* (*R1 msgs False* ⪢= (λ*view. D view*)) *True* − *spmf* (*S1 msgs out1* ⪢= (λ*view. D view*)) *True*| $\leq$ *?rhs*
    for *out1* **using** *R1-inter-dist* **by** *auto*
  thus *?thesis* **by**(*simp add*: *sim-def.adv-P1-def funct-OT-12-def False*)
qed

lemma *add-mult-one-time-pad*:
  assumes *s0* < *order* $\mathcal{G}$
    and *s0* $\neq$ *0*
  shows *map-spmf* (λ $c_v{}'$. (((*b** r0*) + (*s0* * $c_v{}'$)) *mod*(*order* $\mathcal{G}$))) (*sample-uniform* (*order* $\mathcal{G}$)) = *sample-uniform* (*order* $\mathcal{G}$)
proof−
  have *gcd s0* (*order* $\mathcal{G}$) = *1*
    using *assms prime-field* **by** *simp*
  thus *?thesis*
    using *add-mult-one-time-pad* **by** *force*
qed

lemma *security-P2*:
  assumes *m0* $\in$ *carrier* $\mathcal{G}$ *m1* $\in$ *carrier* $\mathcal{G}$
  shows *sim-def.perfect-sec-P2* (*m0*,*m1*) σ
proof−

```
have R2 (m0, m1) σ = S2 σ (if σ then m1 else m0)
  including monad-normalisation
proof−
  have R2 (m0, m1) σ = do {
    a :: nat ← sample-uniform (order 𝒢);
    b :: nat ← sample-uniform (order 𝒢);
    let c_v = (a∗b) mod (order 𝒢);
    c_v' :: nat ← sample-uniform (order 𝒢);
    r0 :: nat ← sample-uniform-units (order 𝒢);
    s0 :: nat ← sample-uniform-units (order 𝒢);
    let w0 = (g [^] a) [^] s0 ⊗ g [^] r0;
    let s' = (((b∗ r0) + ((c_v')∗(s0))) mod(order 𝒢));
    let z = g [^] s' ;
    r1 :: nat ← sample-uniform-units (order 𝒢);
    s1 :: nat ← sample-uniform-units (order 𝒢);
    let w1 = (g [^] a) [^] s1 ⊗ g [^] r1;
    let z' = ((g [^] (c_v)) [^] s1) ⊗ ((g [^] b) [^] r1);
    let enc-m = z ⊗ (if σ then m0 else m1);
    let enc-m' = z' ⊗ (if σ then m1 else m0) ;
    return-spmf(σ, g [^] a, g [^] b, g [^] c_v, w0, enc-m, w1, enc-m')}
   by(simp add: R2-def nat-pow-pow nat-pow-mult pow-generator-mod add.commute)

  also have ... = do {
    a :: nat ← sample-uniform (order 𝒢);
    b :: nat ← sample-uniform (order 𝒢);
    let c_v = (a∗b) mod (order 𝒢);
    r0 :: nat ← sample-uniform-units (order 𝒢);
    s0 :: nat ← sample-uniform-units (order 𝒢);
    let w0 = (g [^] a) [^] s0 ⊗ g [^] r0;
   s' ← map-spmf (λ c_v'. (((b∗ r0) + ((c_v')∗(s0))) mod(order 𝒢))) (sample-uniform
(order 𝒢));
    let z = g [^] s';
    r1 :: nat ← sample-uniform-units (order 𝒢);
    s1 :: nat ← sample-uniform-units (order 𝒢);
    let w1 = (g [^] a) [^] s1 ⊗ g [^] r1;
    let z' = ((g [^] (c_v)) [^] s1) ⊗ ((g [^] b) [^] r1);
    let enc-m = z ⊗ (if σ then m0 else m1);
    let enc-m' = z' ⊗ (if σ then m1 else m0) ;
    return-spmf(σ, g [^] a, g [^] b, g [^] c_v, w0, enc-m, w1, enc-m')}
    by(simp add: bind-map-spmf o-def Let-def)
  also have ... = do {
    a :: nat ← sample-uniform (order 𝒢);
    b :: nat ← sample-uniform (order 𝒢);
    let c_v = (a∗b) mod (order 𝒢);
    r0 :: nat ← sample-uniform-units (order 𝒢);
    s0 :: nat ← sample-uniform-units (order 𝒢);
    let w0 = (g [^] a) [^] s0 ⊗ g [^] r0;
    s' ← (sample-uniform (order 𝒢));
    let z = g [^] s';
```

r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w1 = (**g** [^] *a*) [^] *s1* ⊗ **g** [^] *r1*;
let z′ = ((**g** [^] (c_v)) [^] *s1*) ⊗ ((**g** [^] *b*) [^] *r1*);
let enc-m = *z* ⊗ (*if σ then m0 else m1*);
let enc-m′ = *z′* ⊗ (*if σ then m1 else m0*) ;
return-spmf(σ, **g** [^] *a*, **g** [^] *b*, **g** [^] c_v, *w0*, *enc-m*, *w1*, *enc-m′*)}
by(*simp add*: *add-mult-one-time-pad Let-def mult.commute cong*: *bind-spmf-cong-simp*)
also have ... = *do* {
a :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
b :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
let c_v = (*a*∗*b*) *mod* (*order* $\mathcal{G}$);
r0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w0 = (**g** [^] *a*) [^] *s0* ⊗ **g** [^] *r0*;
r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w1 = (**g** [^] *a*) [^] *s1* ⊗ **g** [^] *r1*;
let z′ = ((**g** [^] (c_v)) [^] *s1*) ⊗ ((**g** [^] *b*) [^] *r1*);
enc-m ← *map-spmf* (λ *s′*. **g** [^] *s′* ⊗ (*if σ then m0 else m1*)) (*sample-uniform* (*order* $\mathcal{G}$));
let enc-m′ = *z′* ⊗ (*if σ then m1 else m0*) ;
return-spmf(σ, **g** [^] *a*, **g** [^] *b*, **g** [^] c_v, *w0*, *enc-m*, *w1*, *enc-m′*)}
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
a :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
b :: *nat* ← *sample-uniform* (*order* $\mathcal{G}$);
let c_v = (*a*∗*b*) *mod* (*order* $\mathcal{G}$);
r0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s0 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w0 = (**g** [^] *a*) [^] *s0* ⊗ **g** [^] *r0*;
r1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
s1 :: *nat* ← *sample-uniform-units* (*order* $\mathcal{G}$);
let w1 = (**g** [^] *a*) [^] *s1* ⊗ **g** [^] *r1*;
let z′ = ((**g** [^] (c_v)) [^] *s1*) ⊗ ((**g** [^] *b*) [^] *r1*);
enc-m ← *map-spmf* (λ *s′*. **g** [^] *s′*) (*sample-uniform* (*order* $\mathcal{G}$));
let enc-m′ = *z′* ⊗ (*if σ then m1 else m0*) ;
return-spmf(σ, **g** [^] *a*, **g** [^] *b*, **g** [^] c_v, *w0*, *enc-m*, *w1*, *enc-m′*)}
by(*simp add*: *sample-uniform-one-time-pad assms*)
ultimately show *?thesis* **by**(*simp add*: *S2-def Let-def bind-map-spmf o-def*)
qed
thus *?thesis*
by(*simp add*: *sim-def.perfect-sec-P2-def funct-OT-12-def*)
qed

end

locale *np-asymp* =
fixes $\mathcal{G}$ :: *security* ⇒ *′grp cyclic-group*

assumes *np*: $\bigwedge\eta$. *np* ($\mathcal{G}$ η)
begin

sublocale *np* $\mathcal{G}$ η **for** η **by**(*simp add*: *np*)

theorem *correctness-asymp*:
assumes *m0* $\in$ *carrier* ($\mathcal{G}$ η) *m1* $\in$ *carrier* ($\mathcal{G}$ η)
shows *sim-def.correctness* η (*m0*, *m1*) σ
by(*simp add*: *correctness assms*)

theorem *security-P1-asymp*:
assumes *negligible* (λ η. *ddh.advantage* η (*inter-S1-adversary* η *D msgs*))
and *negligible* (λ η. *ddh.advantage* η (*R1-inter-adversary* η *D msgs*))
shows *negligible* (λ η. *sim-def.adv-P1* η *msgs* σ *D*)
proof−
have *sim-def.adv-P1* η *msgs* σ *D* $\leq$ *ddh.advantage* η (*R1-inter-adversary* η *D msgs*) + *ddh.advantage* η (*inter-S1-adversary* η *D msgs*)
for η
using *security-P1* **by** *simp*
moreover have *negligible* (λ η. *ddh.advantage* η (*R1-inter-adversary* η *D msgs*) + *ddh.advantage* η (*inter-S1-adversary* η *D msgs*))
using *assms*
by (*simp add*: *negligible-plus*)
ultimately show *?thesis*
using *negligible-le sim-def.adv-P1-def* **by** *auto*
qed

theorem *security-P2-asymp*:
assumes *m0* $\in$ *carrier* ($\mathcal{G}$ η) *m1* $\in$ *carrier* ($\mathcal{G}$ η)
shows *sim-def.perfect-sec-P2* η (*m0*,*m1*) σ
by(*simp add*: *security-P2 assms*)

end

end

2.6 1-out-of-2 OT to 1-out-of-4 OT

Here we construct a protocol that achieves 1-out-of-4 OT from 1-out-of-2 OT. We follow the protocol for constructing 1-out-of-n OT from 1-out-of-2 OT from [2]. We assume the security properties on 1-out-of-2 OT.

theory *OT14* **imports**
Semi-Honest-Def
OT-Functionalities
Uniform-Sampling
begin

type-synonym *input1* = *bool* $\times$ *bool* $\times$ *bool* $\times$ *bool*
type-synonym *input2* = *bool* $\times$ *bool*

type-synonym *'v-OT121' view1* = (*input1* × (*bool* × *bool* × *bool* × *bool* × *bool* × *bool*) × *'v-OT121'* × *'v-OT121'* × *'v-OT121'*)
type-synonym *'v-OT122' view2* = (*input2* × (*bool* × *bool* × *bool* × *bool*) × *'v-OT122'* × *'v-OT122'* × *'v-OT122'*)

locale *ot14-base* =
  fixes *S1-OT12* :: (*bool* × *bool*) ⇒ *unit* ⇒ *'v-OT121 spmf* — simulator for party 1 in OT12
    and *R1-OT12* :: (*bool* × *bool*) ⇒ *bool* ⇒ *'v-OT121 spmf* — real view for party 1 in OT12
    and *adv-OT12* :: *real*
    and *S2-OT12* :: *bool* ⇒ *bool* ⇒ *'v-OT122 spmf*
    and *R2-OT12* :: (*bool* × *bool*) ⇒ *bool* ⇒ *'v-OT122 spmf*
    and *protocol-OT12* :: (*bool* × *bool*) ⇒ *bool* ⇒ (*unit* × *bool*) *spmf*
  assumes *ass-adv-OT12*: *sim-det-def.adv-P1 R1-OT12 S1-OT12 funct-OT12* (*m0*,*m1*) *c D* ≤ *adv-OT12* — bound the advantage of OT12 for party 1
    and *inf-th-OT12-P2*: *sim-det-def.perfect-sec-P2 R2-OT12 S2-OT12 funct-OT12* (*m0*,*m1*) *σ* — information theoretic security for party 2
    and *correct*: *protocol-OT12 msgs b* = *funct-OT-12 msgs b*
    and *lossless-R1-12*: *lossless-spmf* (*R1-OT12 m c*)
    and *lossless-S1-12*: *lossless-spmf* (*S1-OT12 m out1*)
    and *lossless-S2-12*: *lossless-spmf* (*S2-OT12 c out2*)
    and *lossless-R2-12*: *lossless-spmf* (*R2-OT12 M c*)
    and *lossless-funct-OT12*: *lossless-spmf* (*funct-OT12* (*m0*,*m1*) *c*)
    and *lossless-protocol-OT12*: *lossless-spmf* (*protocol-OT12 M C*)
begin

sublocale *OT-12-sim*: *sim-det-def R1-OT12 S1-OT12 R2-OT12 S2-OT12 funct-OT-12 protocol-OT12*
  unfolding *sim-det-def-def*
  by(*simp add*: *lossless-R1-12 lossless-S1-12 lossless-funct-OT12 lossless-R2-12 lossless-S2-12*)

lemma *OT-12-P1-assms-bound'*: |*spmf* (*bind-spmf* (*R1-OT12* (*m0*,*m1*) *c*) (λ *view*. ((*D*::*'v-OT121* ⇒ *bool spmf*) *view*))) *True*
    − *spmf* (*bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*D view*))) *True*| ≤ *adv-OT12*
proof−
  have *sim-det-def.adv-P1 R1-OT12 S1-OT12 funct-OT-12* (*m0*,*m1*) *c D* =
      |*spmf* (*bind-spmf* (*R1-OT12* (*m0*,*m1*) *c*) (λ *view*. (*D view*))) *True*
        − *spmf* (*funct-OT-12* (*m0*,*m1*) *c* ⤜ (λ ((*out1*::*unit*), (*out2*::*bool*)).
          S1-OT12 (*m0*,*m1*) *out1* ⤜ (λ *view*. *D view*))) *True*|
    using *sim-det-def.adv-P1-def*
    using *OT-12-sim.adv-P1-def* **by** *auto*
  also have ... = |*spmf* (*bind-spmf* (*R1-OT12* (*m0*,*m1*) *c*) (λ *view*. ((*D*::*'v-OT121* ⇒ *bool spmf*) *view*))) *True*

− *spmf* (*bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*D view*))) *True*|

  by(*simp add*: *funct-OT-12-def*)
 ultimately show *?thesis*
  by(*metis ass-adv-OT12*)
qed

lemma *OT-12-P2-assm*: *R2-OT12* (*m0*,*m1*) σ = *funct-OT-12* (*m0*,*m1*) σ ≫= (λ (*out1*, *out2*). *S2-OT12* σ *out2*)
 using *inf-th-OT12-P2 OT-12-sim.perfect-sec-P2-def* **by** *blast*

definition *protocol-14-OT* :: *input1* ⇒ *input2* ⇒ (*unit* × *bool*) *spmf*
 where *protocol-14-OT M C* = *do* {
  let (*c0*,*c1*) = *C*;
  let (*m00*, *m01*, *m10*, *m11*) = *M*;
  S0 ← *coin-spmf*;
  S1 ← *coin-spmf*;
  S2 ← *coin-spmf*;
  S3 ← *coin-spmf*;
  S4 ← *coin-spmf*;
  S5 ← *coin-spmf*;
  let a0 = *S0* ⊕ *S2* ⊕ *m00*;
  let a1 = *S0* ⊕ *S3* ⊕ *m01*;
  let a2 = *S1* ⊕ *S4* ⊕ *m10*;
  let a3 = *S1* ⊕ *S5* ⊕ *m11*;
  (-,*Si*) ← *protocol-OT12* (*S0*, *S1*) *c0*;
  (-,*Sj*) ← *protocol-OT12* (*S2*, *S3*) *c1*;
  (-,*Sk*) ← *protocol-OT12* (*S4*, *S5*) *c1*;
  let s2 = *Si* ⊕ (*if c0 then Sk else Sj*) ⊕ (*if c0 then* (*if c1 then a3 else a2*) *else* (*if c1 then a1 else a0*));
  return-spmf ((), *s2*)}

lemma *lossless-protocol-14-OT*: *lossless-spmf* (*protocol-14-OT M C*)
 by(*simp add*: *protocol-14-OT-def lossless-protocol-OT12 split-def*)

definition *R1-14* :: *input1* ⇒ *input2* ⇒ ′*v-OT121 view1 spmf*
 where *R1-14 msgs choice* = *do* {
  let (*m00*, *m01*, *m10*, *m11*) = *msgs*;
  let (*c0*, *c1*) = *choice*;
  S0 :: *bool* ← *coin-spmf*;
  S1 :: *bool* ← *coin-spmf*;
  S2 :: *bool* ← *coin-spmf*;
  S3 :: *bool* ← *coin-spmf*;
  S4 :: *bool* ← *coin-spmf*;
  S5 :: *bool* ← *coin-spmf*;
  a :: ′*v-OT121* ← *R1-OT12* (*S0*, *S1*) *c0*;
  b :: ′*v-OT121* ← *R1-OT12* (*S2*, *S3*) *c1*;
  c :: ′*v-OT121* ← *R1-OT12* (*S4*, *S5*) *c1*;
  return-spmf (*msgs*, (*S0*, *S1*, *S2*, *S3*, *S4*, *S5*), *a*, *b*, *c*)}

lemma *lossless-R1-14*: *lossless-spmf* (*R1-14 msgs C*)
by(*simp add*: *R1-14-def split-def lossless-R1-12*)

definition *R1-14-interm1* :: *input1* ⇒ *input2* ⇒ *'v-OT121 view1 spmf*
where *R1-14-interm1 msgs choice* = *do* {
let (*m00*, *m01*, *m10*, *m11*) = *msgs*;
let (*c0*, *c1*) = *choice*;
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a :: *'v-OT121* ← *S1-OT12* (*S0*, *S1*) ();
b :: *'v-OT121* ← *R1-OT12* (*S2*, *S3*) *c1*;
c :: *'v-OT121* ← *R1-OT12* (*S4*, *S5*) *c1*;
return-spmf (*msgs*, (*S0*, *S1*, *S2*, *S3*, *S4*, *S5*), *a*, *b*, *c*)}

lemma *lossless-R1-14-interm1*: *lossless-spmf* (*R1-14-interm1 msgs C*)
by(*simp add*: *R1-14-interm1-def split-def lossless-R1-12 lossless-S1-12*)

definition *R1-14-interm2* :: *input1* ⇒ *input2* ⇒ *'v-OT121 view1 spmf*
where *R1-14-interm2 msgs choice* = *do* {
let (*m00*, *m01*, *m10*, *m11*) = *msgs*;
let (*c0*, *c1*) = *choice*;
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a :: *'v-OT121* ← *S1-OT12* (*S0*, *S1*) ();
b :: *'v-OT121* ← *S1-OT12* (*S2*, *S3*) ();
c :: *'v-OT121* ← *R1-OT12* (*S4*, *S5*) *c1*;
return-spmf (*msgs*, (*S0*, *S1*, *S2*, *S3*, *S4*, *S5*), *a*, *b*, *c*)}

lemma *lossless-R1-14-interm2*: *lossless-spmf* (*R1-14-interm2 msgs C*)
by(*simp add*: *R1-14-interm2-def split-def lossless-R1-12 lossless-S1-12*)

definition *S1-14* :: *input1* ⇒ *unit* ⇒ *'v-OT121 view1 spmf*
where *S1-14 msgs* - = *do* {
let (*m00*, *m01*, *m10*, *m11*) = *msgs*;
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;

a :: *'v-OT121* ← *S1-OT12* (*S0*, *S1*) ();
b :: *'v-OT121* ← *S1-OT12* (*S2*, *S3*) ();
c :: *'v-OT121* ← *S1-OT12* (*S4*, *S5*) ();
return-spmf (*msgs*, (*S0*, *S1*, *S2*, *S3*, *S4*, *S5*), *a*, *b*, *c*)}

lemma *lossless-S1-14*: *lossless-spmf* (*S1-14 m out*)
by(*simp add*: *S1-14-def lossless-S1-12*)

lemma *reduction-step1*:
shows ∃ *A1*. |*spmf* (*bind-spmf* (*R1-14 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c0*) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True*|
including *monad-normalisation*
proof−
define *A1'* **where** *A1'* == λ (*view* :: *'v-OT121*) (*m0*,*m1*). *do* {
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
b :: *'v-OT121* ← *R1-OT12* (*S2*, *S3*) *c1*;
c :: *'v-OT121* ← *R1-OT12* (*S4*, *S5*) *c1*;
let R = (*M*, (*m0*,*m1*, *S2*, *S3*, *S4*, *S5*), *view*, *b*, *c*);
D R}
have |*spmf* (*bind-spmf* (*R1-14 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c0*) (λ *view*. (*A1' view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1' view* (*m0*,*m1*))))) *True*|
apply(*simp add*: *pair-spmf-alt-def R1-14-def R1-14-interm1-def A1'-def Let-def split-def*)
apply(*subst bind-commute-spmf*[*of S1-OT12 - -*])
apply(*subst bind-commute-spmf*[*of S1-OT12 - -*])
apply(*subst bind-commute-spmf*[*of S1-OT12 - -*])
apply(*subst bind-commute-spmf*[*of S1-OT12 - -*])
apply(*subst bind-commute-spmf*[*of S1-OT12 - -*])
by *auto*
then show *?thesis* **by** *auto*
qed

lemma *reduction-step1'*:
shows |*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c0*) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True*|
≤ *adv-OT12*

(**is** *?lhs* ≤ *adv-OT12*)
proof−
have *int1*: *integrable* (*measure-spmf* (*pair-spmf coin-spmf coin-spmf*)) (λ*x*. *spmf* (*case x of* (*m0*, *m1*) ⇒ *R1-OT12* (*m0*, *m1*) *c0* ⤜ (λ*view*. *A1 view* (*m0*, *m1*))) *True*)
and *int2*: *integrable* (*measure-spmf* (*pair-spmf coin-spmf coin-spmf*)) (λ*x*. *spmf* (*case x of* (*m0*, *m1*) ⇒ *S1-OT12* (*m0*, *m1*) () ⤜ (λ*view*. *A1 view* (*m0*, *m1*))) *True*)
by(*rule measure-spmf.integrable-const-bound*[**where** *B=1*]; *simp add*: *pmf-le-1*)+
have *?lhs* =
|*LINT x*|*measure-spmf* (*pair-spmf coin-spmf coin-spmf*). *spmf* (*case x of* (*m0*, *m1*) ⇒ *R1-OT12* (*m0*, *m1*) *c0* ⤜ (λ*view*. *A1 view* (*m0*, *m1*))) *True*
− *spmf* (*case x of* (*m0*, *m1*) ⇒ *S1-OT12* (*m0*, *m1*) () ⤜ (λ*view*. *A1 view* (*m0*, *m1*))) *True*|
apply(*subst* (*1 2*) *spmf-bind*) **using** *int1 int2* **by** *simp*
also have ... ≤ *LINT x*|*measure-spmf* (*pair-spmf coin-spmf coin-spmf*).
|*spmf* (*R1-OT12 x c0* ⤜ (λ*view*. *A1 view x*)) *True* − *spmf* (*S1-OT12 x* () ⤜ (λ*view*. *A1 view x*)) *True*|
by(*rule integral-abs-bound*[*THEN order-trans*]; *simp add*: *split-beta*)
ultimately have *?lhs* ≤ *LINT x*|*measure-spmf* (*pair-spmf coin-spmf coin-spmf*).

|*spmf* (*R1-OT12 x c0* ⤜ (λ*view*. *A1 view x*)) *True* − *spmf* (*S1-OT12 x* () ⤜ (λ*view*. *A1 view x*)) *True*|
by *simp*
also have *LINT x*|*measure-spmf* (*pair-spmf coin-spmf coin-spmf*).
|*spmf* (*R1-OT12 x c0* ⤜ (λ*view*::'*v-OT121*. *A1 view x*)) *True*
− *spmf* (*S1-OT12 x* () ⤜ (λ*view*::'*v-OT121*. *A1 view x*)) *True*|
≤ *adv-OT12*
apply(*rule integral-mono*[*THEN order-trans*])
apply(*rule measure-spmf.integrable-const-bound*[**where** *B=2*])
apply *clarsimp*
apply(*rule abs-triangle-ineq4*[*THEN order-trans*])
subgoal for *m0 m1*
using *pmf-le-1*[*of R1-OT12* (*m0*, *m1*) *c0* ⤜ (λ*view*. *A1 view* (*m0*, *m1*)) *Some True*]
pmf-le-1[*of S1-OT12* (*m0*, *m1*) () ⤜ (λ*view*. *A1 view* (*m0*, *m1*)) *Some True*]
by *simp*
apply *simp*
apply(*rule measure-spmf.integrable-const*)
apply *clarify*
apply(*rule OT-12-P1-assms-bound'*[*rule-format*])
by *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *reduction-step2*:
shows ∃ *A1*. |*spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True*| =

|spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf (R1-OT12 (m0,m1) c1) (λ view. (A1 view (m0,m1))))) True −
spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf (S1-OT12 (m0,m1) ()) (λ view. (A1 view (m0,m1))))) True|

proof −
define A1′ **where** A1′ == λ (view :: ′v-OT121) (m0,m1). do {
S2 :: bool ← coin-spmf;
S3 :: bool ← coin-spmf;
S4 :: bool ← coin-spmf;
S5 :: bool ← coin-spmf;
a :: ′v-OT121 ← S1-OT12 (S2,S3) ();
c :: ′v-OT121 ← R1-OT12 (S4, S5) c1;
let R = (M, (S2,S3, m0, m1, S4, S5), a, view, c);
D R}
have |spmf (bind-spmf (R1-14-interm1 M (c0, c1)) D) True − spmf (bind-spmf (R1-14-interm2 M (c0, c1)) D) True| =
|spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf (R1-OT12 (m0,m1) c1) (λ view. (A1′ view (m0,m1))))) True −
spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf (S1-OT12 (m0,m1) ()) (λ view. (A1′ view (m0,m1))))) True|
proof −
have (bind-spmf (R1-14-interm1 M (c0, c1)) D) = (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf (R1-OT12 (m0,m1) c1) (λ view. (A1′ view (m0,m1)))))
unfolding R1-14-interm1-def R1-14-interm2-def A1′-def Let-def split-def
apply(simp add: pair-spmf-alt-def)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
including monad-normalisation **by**(simp)
also have (bind-spmf (R1-14-interm2 M (c0, c1)) D) = (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf (S1-OT12 (m0,m1) ()) (λ view. (A1′ view (m0,m1)))))
unfolding R1-14-interm1-def R1-14-interm2-def A1′-def Let-def split-def
apply(simp add: pair-spmf-alt-def)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** - = ⌑ bind-commute-spmf)
apply(rewrite **in** bind-spmf - ⌑ **in** - = ⌑ bind-commute-spmf)
by(simp)

ultimately show *?thesis* **by** *simp*
qed
then show *?thesis* **by** *auto*
qed

lemma *reduction-step3*:
shows ∃ *A1*. |*spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*S1-14 M out*) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True*|
proof−
define *A1′* **where** *A1′* == λ (*view* :: ′*v-OT121*) (*m0*,*m1*). *do* {
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a :: ′*v-OT121* ← *S1-OT12* (*S2*,*S3*) ();
b :: ′*v-OT121* ← *S1-OT12* (*S4*, *S5*) ();
let R = (*M*, (*S2*,*S3*, *S4*, *S5*,*m0*, *m1*), *a*, *b*, *view*);
D R}
have |*spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*S1-14 M out*) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A1′ view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1′ view* (*m0*,*m1*))))) *True*|
proof−
have (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) = (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A1′ view* (*m0*,*m1*)))))
unfolding *R1-14-interm2-def A1′-def Let-def split-def*
apply(*simp add*: *pair-spmf-alt-def*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
including *monad-normalisation* **by**(*simp*)
also have (*bind-spmf* (*S1-14 M out*) *D*) = (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1′ view* (*m0*,*m1*)))))
unfolding *S1-14-def Let-def A1′-def split-def*
apply(*simp add*: *pair-spmf-alt-def*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤

bind-commute-spmf)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** - = ¤ *bind-commute-spmf*)
apply(*rewrite* **in** ¤ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** ¤ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** ¤ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** ¤ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** ¤ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** ¤ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** *bind-spmf* - ¤ **in** ¤ = - *bind-commute-spmf*)
including *monad-normalisation* **by**(*simp*)
ultimately show *?thesis* **by** *simp*
qed
then show *?thesis* **by** *auto*
qed

lemma *reduction-P1-interm*:
shows |*spmf* (*bind-spmf* (*R1-14 M* (*c0*,*c1*)) (*D*)) *True* − *spmf* (*bind-spmf* (*S1-14 M out*) (*D*)) *True*| ≤ *3* ∗ *adv-OT12*
(**is** *?lhs* ≤ *?rhs*)
proof−
have *lhs*: *?lhs* ≤ |*spmf* (*bind-spmf* (*R1-14 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True*| +
|*spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True*| +
|*spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*S1-14 M out*) *D*) *True*|
by *simp*
obtain *A1* **where** *A1*: |*spmf* (*bind-spmf* (*R1-14 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c0*) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True*|
using *reduction-step1* **by** *blast*
obtain *A2* **where** *A2*: |*spmf* (*bind-spmf* (*R1-14-interm1 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A2 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*,

m1). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A2 view* (*m0*,*m1*))))) *True*|
using *reduction-step2* **by** *blast*
obtain *A3* **where** *A3*: |*spmf* (*bind-spmf* (*R1-14-interm2 M* (*c0*, *c1*)) *D*) *True* − *spmf* (*bind-spmf* (*S1-14 M out*) *D*) *True*| =
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A3 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A3 view* (*m0*,*m1*))))) *True*|
using *reduction-step3* **by** *blast*
have *lhs-bound*: *?lhs* ≤ |*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c0*) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True*| +
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A2 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A2 view* (*m0*,*m1*))))) *True*| +
|*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A3 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A3 view* (*m0*,*m1*))))) *True*|
using *A1 A2 A3 lhs* **by** *simp*
have *bound1*: |*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c0*) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A1 view* (*m0*,*m1*))))) *True*|
≤ *adv-OT12*
and *bound2*: |*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A2 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A2 view* (*m0*,*m1*))))) *True*|
≤ *adv-OT12*
and *bound3*: |*spmf* (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*R1-OT12* (*m0*,*m1*) *c1*) (λ *view*. (*A3 view* (*m0*,*m1*))))) *True* −
spmf (*bind-spmf* (*pair-spmf coin-spmf coin-spmf*) (λ(*m0*, *m1*). *bind-spmf* (*S1-OT12* (*m0*,*m1*) ()) (λ *view*. (*A3 view* (*m0*,*m1*))))) *True*| ≤ *adv-OT12*
using *reduction-step1′* **by** *auto*
thus *?thesis*
using *reduction-step1′ lhs-bound* **by** *argo*
qed

lemma *reduction-P1*: |*spmf* (*bind-spmf* (*R1-14 M* (*c0*,*c1*)) (*D*)) *True*
− *spmf* (*funct-OT-14 M* (*c0*,*c1*) ⤜ (λ (*out1*,*out2*). *S1-14 M out1* ⤜ (λ *view*. *D view*))) *True*|
≤ *3* ∗ *adv-OT12*
by(*simp add*: *funct-OT-14-def split-def Let-def reduction-P1-interm*)

Party 2 security.

lemma *coin-coin*: *map-spmf* (λ *S0*. *S0* ⊕ *S3* ⊕ *m1*) *coin-spmf* = *coin-spmf*

(**is** *?lhs* = *?rhs*)
proof−
have *lhs*: *?lhs* = *map-spmf* (λ *S0*. *S0* ⊕ (*S3* ⊕ *m1*)) *coin-spmf* **by** *blast*
also have *op-eq*: ... = *map-spmf* ((⊕) (*S3* ⊕ *m1*)) *coin-spmf*
by (*metis xor-bool-def*)
also have ... = *?rhs*
using *xor-uni-samp* **by** *fastforce*
ultimately show *?thesis*
using *op-eq* **by** *auto*
qed

lemma *coin-coin'*: *map-spmf* (λ *S3*. *S0* ⊕ *S3* ⊕ *m1*) *coin-spmf* = *coin-spmf*
proof−
have *map-spmf* (λ *S3*. *S0* ⊕ *S3* ⊕ *m1*) *coin-spmf* = *map-spmf* (λ *S3*. *S3* ⊕ *S0* ⊕ *m1*) *coin-spmf*
by (*metis xor-left-commute*)
thus *?thesis* **using** *coin-coin* **by** *simp*
qed

definition *R2-14*:: *input1* ⇒ *input2* ⇒ *'v-OT122 view2 spmf*
where *R2-14 M C* = *do* {
let (*m0*,*m1*,*m2*,*m3*) = *M*;
let (*c0*,*c1*) = *C*;
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
let a0 = *S0* ⊕ *S2* ⊕ *m0*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
let a2 = *S1* ⊕ *S4* ⊕ *m2*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *'v-OT122* ← *R2-OT12* (*S0*,*S1*) *c0*;
b :: *'v-OT122* ← *R2-OT12* (*S2*,*S3*) *c1*;
c :: *'v-OT122* ← *R2-OT12* (*S4*,*S5*) *c1*;
return-spmf (*C*, (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}

lemma *lossless-R2-14*: *lossless-spmf* (*R2-14 M C*)
by(*simp add*: *R2-14-def split-def lossless-R2-12*)

definition *S2-14* :: *input2* ⇒ *bool* ⇒ *'v-OT122 view2 spmf*
where *S2-14 C out* = *do* {
let ((*c0*::*bool*),(*c1*::*bool*)) = *C*;
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;

S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
a1 :: *bool* ← *coin-spmf*;
a2 :: *bool* ← *coin-spmf*;
a3 :: *bool* ← *coin-spmf*;
let a0′ = (*if* ((¬ *c0*) ∧ (¬ *c1*)) *then* (*S0* ⊕ *S2* ⊕ *out*) *else a0*);
let a1′ = (*if* ((¬ *c0*) ∧ *c1*) *then* (*S0* ⊕ *S3* ⊕ *out*) *else a1*);
let a2′ = (*if* (*c0* ∧ (¬ *c1*)) *then* (*S1* ⊕ *S4* ⊕ *out*) *else a2*);
let a3′ = (*if* (*c0* ∧ *c1*) *then* (*S1* ⊕ *S5* ⊕ *out*) *else a3*);
a :: *′v-OT122* ← *S2-OT12* (*c0*::*bool*) (*if c0 then S1 else S0*);
b :: *′v-OT122* ← *S2-OT12* (*c1*::*bool*) (*if c1 then S3 else S2*);
c :: *′v-OT122* ← *S2-OT12* (*c1*::*bool*) (*if c1 then S5 else S4*);
return-spmf ((*c0*,*c1*), (*a0′*,*a1′*,*a2′*,*a3′*), *a*,*b*,*c*)}

lemma *lossless-S2-14*: *lossless-spmf* (*S2-14 c out*)
by(*simp add*: *S2-14-def lossless-S2-12 split-def*)

lemma *P2-OT-14-FT*: *R2-14* (*m0*,*m1*,*m2*,*m3*) (*False*,*True*) = *funct-OT-14* (*m0*,*m1*,*m2*,*m3*) (*False*,*True*) ⤜ (λ (*out1*, *out2*). *S2-14* (*False*,*True*) *out2*)
including *monad-normalisation*
proof−
have *R2-14* (*m0*,*m1*,*m2*,*m3*) (*False*,*True*) = *do* {
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *map-spmf* (λ *S2*. *S0* ⊕ *S2* ⊕ *m0*) *coin-spmf*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
a2 ← *map-spmf* (λ *S4*. *S1* ⊕ *S4* ⊕ *m2*) *coin-spmf*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *′v-OT122* ← *S2-OT12 False S0*;
b :: *′v-OT122* ← *S2-OT12 True S3*;
c :: *′v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*False*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
by(*simp add*: *bind-map-spmf o-def Let-def R2-14-def inf-th-OT12-P2 funct-OT-12-def OT-12-P2-assm*)
also have ... = *do* {
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
a2 ← *coin-spmf*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *′v-OT122* ← *S2-OT12 False S0*;
b :: *′v-OT122* ← *S2-OT12 True S3*;
c :: *′v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*False*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}

using *coin-coin'* **by** *simp*
also have ... = *do* {
S0 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
a2 :: *bool* ← *coin-spmf*;
a3 ← *map-spmf* (λ *S1*. *S1* ⊕ *S5* ⊕ *m3*) *coin-spmf*;
a :: *'v-OT122* ← *S2-OT12 False S0*;
b :: *'v-OT122* ← *S2-OT12 True S3*;
c :: *'v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*False*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
S0 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
a2 :: *bool* ← *coin-spmf*;
a3 ← *coin-spmf*;
a :: *'v-OT122* ← *S2-OT12 False S0*;
b :: *'v-OT122* ← *S2-OT12 True S3*;
c :: *'v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*False*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
using *coin-coin* **by** *simp*
ultimately show *?thesis*
by(*simp add*: *funct-OT-14-def S2-14-def bind-spmf-const*)
qed

lemma *P2-OT-14-TT*: *R2-14* (*m0*,*m1*,*m2*,*m3*) (*True*,*True*) = *funct-OT-14* (*m0*,*m1*,*m2*,*m3*) (*True*,*True*) ⤜ (λ (*out1*, *out2*). *S2-14* (*True*,*True*) *out2*)
including *monad-normalisation*
proof−
have *R2-14* (*m0*,*m1*,*m2*,*m3*) (*True*,*True*) = *do* {
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *map-spmf* (λ *S2*. *S0* ⊕ *S2* ⊕ *m0*) *coin-spmf*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
a2 ← *map-spmf* (λ *S4*. *S1* ⊕ *S4* ⊕ *m2*) *coin-spmf*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *'v-OT122* ← *S2-OT12 True S1*;
b :: *'v-OT122* ← *S2-OT12 True S3*;
c :: *'v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*True*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
by(*simp add*: *bind-map-spmf o-def R2-14-def inf-th-OT12-P2 funct-OT-12-def*

OT-12-P2-assm Let-def)
also have ... = *do* {
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
let a1 = *S0* ⊕ *S3* ⊕ *m1*;
a2 ← *coin-spmf*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *'v-OT122* ← *S2-OT12 True S1*;
b :: *'v-OT122* ← *S2-OT12 True S3*;
c :: *'v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*True*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
using *coin-coin'* **by** *simp*
also have ... = *do* {
S1 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
a1 :: *bool* ← *map-spmf* (λ *S0*. *S0* ⊕ *S3* ⊕ *m1*) *coin-spmf*;
a2 ← *coin-spmf*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *'v-OT122* ← *S2-OT12 True S1*;
b :: *'v-OT122* ← *S2-OT12 True S3*;
c :: *'v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*True*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
S1 :: *bool* ← *coin-spmf*;
S3 :: *bool* ← *coin-spmf*;
S5 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
a1 :: *bool* ← *coin-spmf*;
a2 ← *coin-spmf*;
let a3 = *S1* ⊕ *S5* ⊕ *m3*;
a :: *'v-OT122* ← *S2-OT12 True S1*;
b :: *'v-OT122* ← *S2-OT12 True S3*;
c :: *'v-OT122* ← *S2-OT12 True S5*;
return-spmf ((*True*,*True*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
using *coin-coin* **by** *simp*
ultimately show *?thesis*
by(*simp add*: *funct-OT-14-def S2-14-def bind-spmf-const*)
qed

lemma *P2-OT-14-FF*: *R2-14* (*m0*,*m1*,*m2*,*m3*) (*False*, *False*) = *funct-OT-14* (*m0*,*m1*,*m2*,*m3*) (*False*, *False*) ⤜ (λ (*out1*, *out2*). *S2-14* (*False*, *False*) *out2*)
including *monad-normalisation*
proof−

```
have R2-14 (m0,m1,m2,m3) (False,False) =  do {
  S0 :: bool ← coin-spmf;
  S1 :: bool ← coin-spmf;
  S2 :: bool ← coin-spmf;
  S4 :: bool ← coin-spmf;
  let a0 = S0 ⊕ S2 ⊕ m0;
  a1 :: bool ← map-spmf (λ S3. S0 ⊕ S3 ⊕ m1) coin-spmf;
  let a2 = S1 ⊕ S4 ⊕ m2;
  a3 ← map-spmf (λ S5. S1 ⊕ S5 ⊕ m3) coin-spmf;
  a :: 'v-OT122 ← S2-OT12 False S0;
  b :: 'v-OT122 ← S2-OT12 False S2;
  c :: 'v-OT122 ← S2-OT12 False S4;
  return-spmf ((False,False), (a0,a1,a2,a3), a,b,c)}
  by(simp add: bind-map-spmf o-def R2-14-def inf-th-OT12-P2 funct-OT-12-def
OT-12-P2-assm Let-def)
also have ... = do {
  S0 :: bool ← coin-spmf;
  S1 :: bool ← coin-spmf;
  S2 :: bool ← coin-spmf;
  S4 :: bool ← coin-spmf;
  let a0 = S0 ⊕ S2 ⊕ m0;
  a1 :: bool ← coin-spmf;
  let a2 = S1 ⊕ S4 ⊕ m2;
  a3 ← coin-spmf;
  a :: 'v-OT122 ← S2-OT12 False S0;
  b :: 'v-OT122 ← S2-OT12 False S2;
  c :: 'v-OT122 ← S2-OT12 False S4;
  return-spmf ((False,False), (a0,a1,a2,a3), a,b,c)}
  using coin-coin' by simp
also have ... = do {
  S0 :: bool ← coin-spmf;
  S2 :: bool ← coin-spmf;
  S4 :: bool ← coin-spmf;
  let a0 = S0 ⊕ S2 ⊕ m0;
  a1 :: bool ← coin-spmf;
  a2 :: bool ← map-spmf (λ S1. S1 ⊕ S4 ⊕ m2) coin-spmf;
  a3 ← coin-spmf;
  a :: 'v-OT122 ← S2-OT12 False S0;
  b :: 'v-OT122 ← S2-OT12 False S2;
  c :: 'v-OT122 ← S2-OT12 False S4;
  return-spmf ((False,False), (a0,a1,a2,a3), a,b,c)}
  by(simp add: bind-map-spmf o-def Let-def)
also have ... = do {
  S0 :: bool ← coin-spmf;
  S2 :: bool ← coin-spmf;
  S4 :: bool ← coin-spmf;
  let a0 = S0 ⊕ S2 ⊕ m0;
  a1 :: bool ← coin-spmf;
  a2 :: bool ← coin-spmf;
```

a3 ← *coin-spmf*;
a :: ′*v-OT122* ← *S2-OT12 False S0*;
b :: ′*v-OT122* ← *S2-OT12 False S2*;
c :: ′*v-OT122* ← *S2-OT12 False S4*;
return-spmf ((*False*,*False*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
using *coin-coin* **by** *simp*
ultimately show *?thesis*
by(*simp add*: *funct-OT-14-def S2-14-def bind-spmf-const*)
qed

lemma *P2-OT-14-TF*: *R2-14* (*m0*,*m1*,*m2*,*m3*) (*True*,*False*) = *funct-OT-14* (*m0*,*m1*,*m2*,*m3*) (*True*,*False*) ≫= (λ (*out1*, *out2*). *S2-14* (*True*,*False*) *out2*)
including *monad-normalisation*
proof−
have *R2-14* (*m0*,*m1*,*m2*,*m3*) (*True*,*False*) = *do* {
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
let a0 = *S0* ⊕ *S2* ⊕ *m0*;
a1 :: *bool* ← *map-spmf* (λ *S3*. *S0* ⊕ *S3* ⊕ *m1*) *coin-spmf*;
let a2 = *S1* ⊕ *S4* ⊕ *m2*;
a3 ← *map-spmf* (λ *S5*. *S1* ⊕ *S5* ⊕ *m3*) *coin-spmf*;
a :: ′*v-OT122* ← *S2-OT12 True S1*;
b :: ′*v-OT122* ← *S2-OT12 False S2*;
c :: ′*v-OT122* ← *S2-OT12 False S4*;
return-spmf ((*True*,*False*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
apply(*simp add*: *R2-14-def inf-th-OT12-P2 OT-12-P2-assm funct-OT-12-def Let-def*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** ⌑ = - *bind-commute-spmf*)
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
S0 :: *bool* ← *coin-spmf*;
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
let a0 = *S0* ⊕ *S2* ⊕ *m0*;
a1 :: *bool* ← *coin-spmf*;
let a2 = *S1* ⊕ *S4* ⊕ *m2*;
a3 ← *coin-spmf*;
a :: ′*v-OT122* ← *S2-OT12 True S1*;
b :: ′*v-OT122* ← *S2-OT12 False S2*;
c :: ′*v-OT122* ← *S2-OT12 False S4*;
return-spmf ((*True*,*False*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
using *coin-coin*′ **by** *simp*
also have ... = *do* {

S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *map-spmf* (λ *S0*. *S0* ⊕ *S2* ⊕ *m0*) *coin-spmf*;
a1 :: *bool* ← *coin-spmf*;
let a2 = *S1* ⊕ *S4* ⊕ *m2*;
a3 ← *coin-spmf*;
a :: *'v-OT122* ← *S2-OT12 True S1*;
b :: *'v-OT122* ← *S2-OT12 False S2*;
c :: *'v-OT122* ← *S2-OT12 False S4*;
return-spmf ((*True*,*False*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
S1 :: *bool* ← *coin-spmf*;
S2 :: *bool* ← *coin-spmf*;
S4 :: *bool* ← *coin-spmf*;
a0 :: *bool* ← *coin-spmf*;
a1 :: *bool* ← *coin-spmf*;
let a2 = *S1* ⊕ *S4* ⊕ *m2*;
a3 ← *coin-spmf*;
a :: *'v-OT122* ← *S2-OT12 True S1*;
b :: *'v-OT122* ← *S2-OT12 False S2*;
c :: *'v-OT122* ← *S2-OT12 False S4*;
return-spmf ((*True*,*False*), (*a0*,*a1*,*a2*,*a3*), *a*,*b*,*c*)}
using *coin-coin* **by** *simp*
ultimately show *?thesis*
apply(*simp add*: *funct-OT-14-def S2-14-def bind-spmf-const*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
apply(*rewrite* **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** *bind-spmf* - ⌑ **in** - = ⌑ *bind-commute-spmf*)
by *simp*
qed

lemma *P2-sec-OT-14-split*: *R2-14* (*m0*,*m1*,*m2*,*m3*) (*c0*,*c1*) = *funct-OT-14* (*m0*,*m1*,*m2*,*m3*) (*c0*,*c1*) ⤜ (λ (*out1*, *out2*). *S2-14* (*c0*,*c1*) *out2*)
by(*cases c0*; *cases c1*; *auto simp add*: *P2-OT-14-FF P2-OT-14-TF P2-OT-14-FT P2-OT-14-TT*)

lemma *P2-sec-OT-14*: *R2-14 M C* = *funct-OT-14 M C* ⤜ (λ (*out1*, *out2*). *S2-14 C out2*)
by(*metis P2-sec-OT-14-split surj-pair*)

sublocale *OT-14*: *sim-det-def R1-14 S1-14 R2-14 S2-14 funct-OT-14 protocol-14-OT*
unfolding *sim-det-def-def*
by(*simp add*: *lossless-R1-14 lossless-S1-14 lossless-funct-14-OT lossless-R2-14 lossless-S2-14*)

lemma *correctness-OT-14*:

shows *funct-OT-14 M C = protocol-14-OT M C*
proof−
have *S1 = (S5 = (S1 = (S5 = d))) = d* **for** *S1 S5 d* **by** *auto*
thus *?thesis*
by(*cases fst C*; *cases snd C*; *simp add*: *funct-OT-14-def protocol-14-OT-def correct funct-OT-12-def lossless-funct-OT-12 bind-spmf-const split-def*)
qed

lemma *OT-14-correct*: *OT-14.correctness M C*
unfolding *OT-14.correctness-def*
using *correctness-OT-14* **by** *auto*

lemma *OT-14-P2-sec*: *OT-14.perfect-sec-P2 m1 m2*
unfolding *OT-14.perfect-sec-P2-def*
using *P2-sec-OT-14* **by** *blast*

lemma *OT-14-P1-sec*: *OT-14.adv-P1 m1 m2 D* $\leq$ *3* $*$ *adv-OT12*
unfolding *OT-14.adv-P1-def*
by (*metis reduction-P1 surj-pair*)

end

locale *OT-14-asymp = sim-det-def +*
fixes *S1-OT12* :: *nat* ⇒ (*bool* × *bool*) ⇒ *unit* ⇒ *'v-OT121 spmf*
and *R1-OT12* :: *nat* ⇒ (*bool* × *bool*) ⇒ *bool* ⇒ *'v-OT121 spmf*
and *adv-OT12* :: *nat* ⇒ *real*
and *S2-OT12* :: *nat* ⇒ *bool* ⇒ *bool* ⇒ *'v-OT122 spmf*
and *R2-OT12* :: *nat* ⇒ (*bool* × *bool*) ⇒ *bool* ⇒ *'v-OT122 spmf*
and *protocol-OT12* :: (*bool* × *bool*) ⇒ *bool* ⇒ (*unit* × *bool*) *spmf*
assumes *ot14-base*: ⋀ (*n*::*nat*). *ot14-base* (*S1-OT12 n*) (*R1-12-0T n*) (*adv-OT12 n*) (*S2-OT12 n*) (*R2-12OT n*) (*protocol-OT12*)
begin

sublocale *ot14-base* (*S1-OT12 n*) (*R1-12-0T n*) (*adv-OT12 n*) (*S2-OT12 n*) (*R2-12OT n*) **using** *local.ot14-base* **by** *simp*

lemma *OT-14-P1-sec*: *OT-14.adv-P1* (*R1-12-0T n*) *n m1 m2 D* $\leq$ *3* $*$ (*adv-OT12 n*)
unfolding *OT-14.adv-P1-def* **using** *reduction-P1 surj-pair* **by** *metis*

theorem *OT-14-P1-asym-sec*: *negligible* (λ *n*. *OT-14.adv-P1* (*R1-12-0T n*) *n m1 m2 D*) **if** *negligible* (λ *n*. *adv-OT12 n*)
proof−
have *adv-neg*: *negligible* (λ*n*. *3* $*$ *adv-OT12 n*) **using** *that negligible-cmultI* **by** *simp*
have |*OT-14.adv-P1* (*R1-12-0T n*) *n m1 m2 D*| $\leq$ |*3* $*$ (*adv-OT12 n*)| **for** *n*
proof −
have |*OT-14.adv-P1* (*R1-12-0T n*) *n m1 m2 D*| $\leq$ *3* $*$ *adv-OT12 n*
using *OT-14.adv-P1-def OT-14-P1-sec* **by** *auto*

then show *?thesis*
by (*meson abs-ge-self order-trans*)
qed
thus *?thesis* **using** *OT-14-P1-sec negligible-le adv-neg*
by (*metis* (*no-types*, *lifting*) *negligible-absI*)
qed

theorem *OT-14-P2-asym-sec*: *OT-14.perfect-sec-P2 R2-OT12 n m1 m2*
using *OT-14-P2-sec* **by** *simp*

end

end

2.7 1-out-of-4 OT to GMW

We prove security for the gates of the GMW protocol in the semi honest model. We assume security on 1-out-of-4 OT.

theory *GMW* **imports**
OT14
begin

type-synonym *share-1* = *bool*
type-synonym *share-2* = *bool*

type-synonym *shares-1* = *bool list*
type-synonym *shares-2* = *bool list*

type-synonym *msgs-14-OT* = (*bool* × *bool* × *bool* × *bool*)
type-synonym *choice-14-OT* = (*bool* × *bool*)

type-synonym *share-wire* = (*share-1* × *share-2*)

locale *gmw-base* =
fixes *S1-14-OT* :: *msgs-14-OT* ⇒ *unit* ⇒ *'v-14-OT1 spmf* — simulated view for party 1 of OT14
and *R1-14-OT* :: *msgs-14-OT* ⇒ *choice-14-OT* ⇒ *'v-14-OT1 spmf* — real view for party 1 of OT14
and *S2-14-OT* :: *choice-14-OT* ⇒ *bool* ⇒ *'v-14-OT2 spmf*
and *R2-14-OT* :: *msgs-14-OT* ⇒ *choice-14-OT* ⇒ *'v-14-OT2 spmf*
and *protocol-14-OT* :: *msgs-14-OT* ⇒ *choice-14-OT* ⇒ (*unit* × *bool*) *spmf*
and *adv-14-OT* :: *real*
assumes *P1-OT-14-adv-bound*: *sim-det-def.adv-P1 R1-14-OT S1-14-OT funct-14-OT M C D* ≤ *adv-14-OT* — bound the advantage of party 1 in the 1-out-of-4 OT
and *P2-OT-12-inf-theoretic*: *sim-det-def.perfect-sec-P2 R2-14-OT S2-14-OT funct-14-OT M C* — information theoretic security for party 2 in the 1-out-of-4 OT
and *correct-14*: *funct-OT-14 msgs C* = *protocol-14-OT msgs C* — correctness of the 1-out-of-4 OT

and *lossless-R1-14-OT*: *lossless-spmf* (*R1-14-OT* (*m1*,*m2*,*m3*,*m4*) (*c0*,*c1*))
and *lossless-R2-14-OT*: *lossless-spmf* (*R2-14-OT* (*m1*,*m2*,*m3*,*m4*) (*c0*,*c1*))
and *lossless-S1-14-OT*: *lossless-spmf* (*S1-14-OT* (*m1*,*m2*,*m3*,*m4*) ())
and *lossless-S2-14-OT*: *lossless-spmf* (*S2-14-OT* (*c0*,*c1*) *b*)
and *lossless-protocol-14-OT*: *lossless-spmf* (*protocol-14-OT S C*)
and *lossless-funct-14-OT*: *lossless-spmf* (*funct-14-OT M C*)
begin

lemma *funct-14*: *funct-OT-14* (*m00*,*m01*,*m10*,*m11*) (*c0*,*c1*)
= *return-spmf* ((),*if c0 then* (*if c1 then m11 else m10*) *else* (*if c1 then m01 else m00*))
by(*simp add*: *funct-OT-14-def*)

sublocale *OT-14-sim*: *sim-det-def R1-14-OT S1-14-OT R2-14-OT S2-14-OT funct-14-OT protocol-14-OT*
unfolding *sim-det-def-def*
by(*simp add*: *lossless-R1-14-OT lossless-S1-14-OT lossless-funct-14-OT lossless-R2-14-OT lossless-S2-14-OT*)

lemma *inf-th-14-OT-P4*: *R2-14-OT msgs C* = (*funct-OT-14 msgs C* ⤜ (λ (*s1*, *s2*). *S2-14-OT C s2*))
using *P2-OT-12-inf-theoretic sim-det-def.perfect-sec-P2-def OT-14-sim.perfect-sec-P2-def*
by *auto*

lemma *ass-adv-14-OT*: |*spmf* (*bind-spmf* (*S1-14-OT msgs* ()) (λ *view*. (*D view*))) *True* −
spmf (*bind-spmf* (*R1-14-OT msgs* (*c0*,*c1*)) (λ *view*. (*D view*))) *True* | ≤ *adv-14-OT*
(**is** *?lhs* ≤ *adv-14-OT*)
proof−
have *funct-OT-14* (*m0*,*m1*,*m2*,*m3*) (*c0*, *c1*) ⤜ (λ(*o1*, *o2*). *S1-14-OT* (*m0*,*m1*,*m2*,*m3*) () ⤜ *D*) = *S1-14-OT* (*m0*,*m1*,*m2*,*m3*) () ⤜ *D*
for *m0 m1 m2 m3* **by**(*simp add*: *funct-14*)
hence *funct*: *funct-OT-14 msgs* (*c0*, *c1*) ⤜ (λ(*o1*, *o2*). *S1-14-OT msgs* () ⤜ *D*) = *S1-14-OT msgs* () ⤜ *D*
by (*metis prod-cases4*)
have *?lhs* = |*spmf* (*bind-spmf* (*R1-14-OT msgs* (*c0*,*c1*)) (λ *view*. (*D view*))) *True*
− *spmf* (*bind-spmf* (*S1-14-OT msgs* ()) (λ *view*. (*D view*))) *True*|
by *linarith*
hence ... = |(*spmf* (*R1-14-OT msgs* (*c0*,*c1*) ⤜ (λ *view*. *D view*)) *True*)
− *spmf* (*funct-OT-14 msgs* (*c0*,*c1*) ⤜ (λ (*o1*, *o2*). *S1-14-OT msgs o1* ⤜ (λ *view*. *D view*))) *True*|
by(*simp add*: *funct*)
thus *?thesis* **using** *P1-OT-14-adv-bound sim-det-def.adv-P1-def*
by (*simp add*: *OT-14-sim.adv-P1-def abs-minus-commute*)
qed

The sharing scheme

definition *share* :: *bool* ⇒ *share-wire spmf*
  where *share x* = *do* {
    a_1 ← *coin-spmf*;
    let $b_1 = x \oplus a_1$;
    return-spmf (a_1, b_1)}

lemma *lossless-share* [*simp*]: *lossless-spmf* (*share x*)
  by(*simp add*: *share-def*)

definition *reconstruct* :: (*share-1* × *share-2*) ⇒ *bool spmf*
  where *reconstruct shares* = *do* {
    let (*a*,*b*) = *shares*;
    return-spmf (*a* ⊕ *b*)}

lemma *lossless-reconstruct* [*simp*]: *lossless-spmf* (*reconstruct s*)
  by(*simp add*: *reconstruct-def split-def*)

lemma *reconstruct-share* : (*bind-spmf* (*share x*) *reconstruct*) = (*return-spmf x*)
proof−
  have *y* = (*y* = *x*) = *x* **for** *y* **by** *auto*
  thus *?thesis*
    by(*auto simp add*: *share-def reconstruct-def bind-spmf-const eq-commute*)
qed

lemma (*reconstruct* (*s1*,*s2*) ⤜ (λ *rec*. *share rec* ⤜ (λ *shares*. *reconstruct shares*)))
= *return-spmf* (*s1* ⊕ *s2*)
  apply(*simp add*: *reconstruct-share reconstruct-def share-def*)
  apply(*cases s1*; *cases s2*) **by**(*auto simp add*: *bind-spmf-const*)

definition *xor-evaluate* :: *bool* ⇒ *bool* ⇒ *bool spmf*
  where *xor-evaluate A B* = *return-spmf* (*A* ⊕ *B*)

definition *xor-funct* :: *share-wire* ⇒ *share-wire* ⇒ (*bool* × *bool*) *spmf*
  where *xor-funct A B* = *do* {
    let (*a1*, *b1*) = *A*;
    let (*a2*,*b2*) = *B*;
    return-spmf (*a1* ⊕ *a2*, *b1* ⊕ *b2*)}

lemma *lossless-xor-funct*: *lossless-spmf* (*xor-funct A B*)
  by(*simp add*: *xor-funct-def split-def*)

definition *xor-protocol* :: *share-wire* ⇒ *share-wire* ⇒ (*bool* × *bool*) *spmf*
  where *xor-protocol A B* = *do* {
    let (*a1*, *b1*) = *A*;
    let (*a2*,*b2*) = *B*;
    return-spmf (*a1* ⊕ *a2*, *b1* ⊕ *b2*)}

lemma *lossless-xor-protocol*: *lossless-spmf* (*xor-protocol A B*)
  by(*simp add*: *xor-protocol-def split-def*)

lemma *share-xor-reconstruct*:
shows *share x* ⤜ (λ *w1*. *share y* ⤜ (λ *w2*. *xor-protocol w1 w2*
⤜ (λ (*a*, *b*). *reconstruct* (*a*, *b*)))) = *xor-evaluate x y*
proof −
have (*ya* = (¬ *yb*)) = ((*x* = (¬ *ya*)) = (*y* = (¬ *yb*))) = (*x* = (¬ *y*)) **for** *ya yb*
by *auto*
then show *?thesis*
by(*simp add*: *share-def xor-protocol-def reconstruct-def xor-evaluate-def bind-spmf-const*)
qed

definition *R1-xor* :: (*bool* × *bool*) ⇒ (*bool* × *bool*) ⇒ (*bool* × *bool*) *spmf*
where *R1-xor A B* = *return-spmf A*

lemma *lossless-R1-xor*: *lossless-spmf* (*R1-xor A B*)
by(*simp add*: *R1-xor-def*)

definition *S1-xor* :: (*bool* × *bool*) ⇒ *bool* ⇒ (*bool* × *bool*) *spmf*
where *S1-xor A out* = *return-spmf A*

lemma *lossless-S1-xor*: *lossless-spmf* (*S1-xor A out*)
by(*simp add*: *S1-xor-def*)

lemma *P1-xor-inf-th*: *R1-xor A B* = *xor-funct A B* ⤜ (λ (*out1*, *out2*). *S1-xor A out1*)
by(*simp add*: *R1-xor-def xor-funct-def S1-xor-def split-def*)

definition *R2-xor* :: (*bool* × *bool*) ⇒ (*bool* × *bool*) ⇒ (*bool* × *bool*) *spmf*
where *R2-xor A B* = *return-spmf B*

lemma *lossless-R2-xor*: *lossless-spmf* (*R2-xor A B*)
by(*simp add*: *R2-xor-def*)

definition *S2-xor* :: (*bool* × *bool*) ⇒ *bool* ⇒ (*bool* × *bool*) *spmf*
where *S2-xor B out* = *return-spmf B*

lemma *lossless-S2-xor*: *lossless-spmf* (*S2-xor A out*)
by(*simp add*: *S2-xor-def*)

lemma *P2-xor-inf-th*: *R2-xor A B* = *xor-funct A B* ⤜ (λ (*out1*, *out2*). *S2-xor B out2*)
by(*simp add*: *R2-xor-def xor-funct-def S2-xor-def split-def*)

sublocale *xor-sim-det*: *sim-det-def R1-xor S1-xor R2-xor S2-xor xor-funct xor-protocol*

unfolding *sim-det-def-def*
by(*simp add*: *lossless-R1-xor lossless-S1-xor lossless-R2-xor lossless-S2-xor lossless-xor-funct*)

lemma *xor-sim-det.perfect-sec-P1 m1 m2*
by(*simp add*: *xor-sim-det.perfect-sec-P1-def P1-xor-inf-th*)

lemma *xor-sim-det.perfect-sec-P2 m1 m2*
by(*simp add*: *xor-sim-det.perfect-sec-P2-def P2-xor-inf-th*)

definition *and-funct* :: (*share-1* × *share-2*) ⇒ (*share-1* × *share-2*) ⇒ *share-wire spmf*
where *and-funct A B* = *do* {
let (*a1*, *a2*) = *A*;
let (*b1*,*b2*) = *B*;
σ ← *coin-spmf*;
return-spmf (*σ*, *σ* ⊕ ((*a1* ⊕ *b1*) ∧ (*a2* ⊕ *b2*)))}

lemma *lossless-and-funct*: *lossless-spmf* (*and-funct A B*)
by(*simp add*: *and-funct-def split-def*)

definition *and-evaluate* :: *bool* ⇒ *bool* ⇒ *bool spmf*
where *and-evaluate A B* = *return-spmf* (*A* ∧ *B*)

definition *and-protocol* :: *share-wire* ⇒ *share-wire* ⇒ *share-wire spmf*
where *and-protocol A B* = *do* {
let (*a1*, *b1*) = *A*;
let (*a2*,*b2*) = *B*;
σ ← *coin-spmf*;
let s0 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*b1* ⊕ *False*));
let s1 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*b1* ⊕ *True*));
let s2 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*b1* ⊕ *False*));
let s3 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*b1* ⊕ *True*));
(-, *s*) ← *protocol-14-OT* (*s0*,*s1*,*s2*,*s3*) (*a2*,*b2*);
return-spmf (*σ*, *s*)}

lemma *lossless-and-protocol*: *lossless-spmf* (*and-protocol A B*)
by(*simp add*: *and-protocol-def split-def lossless-protocol-14-OT*)

lemma *and-correct*: *and-protocol* (*a1*, *b1*) (*a2*,*b2*) = *and-funct* (*a1*, *b1*) (*a2*,*b2*)
apply(*simp add*: *and-protocol-def and-funct-def correct-14*[*symmetric*] *funct-14*)
by(*cases b2* ; *cases b1*; *cases a1*; *cases a2*; *auto*)

lemma *share-and-reconstruct*:
shows *share x* ≫= (λ (*a1*,*a2*). *share y* ≫= (λ (*b1*,*b2*).
and-protocol (*a1*,*b1*) (*a2*,*b2*) ≫= (λ (*a*, *b*). *reconstruct* (*a*, *b*)))) = *and-evaluate x y*
proof−
have (*yc* = (¬ (*if x* = (¬ *ya*) *then if snd* (*snd* (*ya*, *x* = (¬ *ya*)), *snd* (*yb*, *y* = (¬ *yb*))) *then yc*
= (*fst* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*))) ∨ *snd* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*))))
else yc = (*fst* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*))) ∨ ¬ *snd*

(*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*))))
else if snd (*snd* (*ya*, *x* = (¬ *ya*)), *snd* (*yb*, *y* = (¬ *yb*))) *then yc* = (*fst* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*)))
⟶ *snd* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*))))
else yc = (*fst* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*)))
⟶ ¬ *snd* (*fst* (*ya*, *x* = (¬ *ya*)), *fst* (*yb*, *y* = (¬ *yb*))))))) = (*x* ∧ *y*)
for *yc yb ya* **by** *auto*
then show *?thesis*
by(*auto simp add*: *share-def reconstruct-def and-protocol-def and-evaluate-def split-def correct-14*[*symmetric*] *funct-14 bind-spmf-const Let-def*)
qed

definition *and-R1* :: (*share-1* × *share-1*) ⇒ (*share-2* × *share-2*) ⇒ (((*share-1* × *share-1*) × *bool* × *'v-14-OT1*) × (*share-1* × *share-2*)) *spmf*
where *and-R1 A B* = *do* {
let (*a1*, *a2*) = *A*;
let (*b1*,*b2*) = *B*;
σ ← *coin-spmf*;
let s0 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*));
let s1 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*));
let s2 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*));
let s3 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*));
V ← *R1-14-OT* (*s0*,*s1*,*s2*,*s3*) (*b1*,*b2*);
(-, *s*) ← *protocol-14-OT* (*s0*,*s1*,*s2*,*s3*) (*b1*,*b2*);
return-spmf (((*a1*,*a2*), *σ*, *V*), (*σ*, *s*))}

lemma *lossless-and-R1*: *lossless-spmf* (*and-R1 A B*)
apply(*simp add*: *and-R1-def split-def lossless-R1-14-OT lossless-protocol-14-OT Let-def*)
by (*metis prod.collapse lossless-R1-14-OT*)

definition *S1-and* :: (*share-1* × *share-1*) ⇒ *bool* ⇒ (((*bool* × *bool*) × *bool* × *'v-14-OT1*)) *spmf*
where *S1-and A σ* = *do* {
let (*a1*,*a2*) = *A*;
let s0 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*));
let s1 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*));
let s2 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*));
let s3 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*));
V ← *S1-14-OT* (*s0*,*s1*,*s2*,*s3*) ();
return-spmf ((*a1*,*a2*), *σ*, *V*)}

definition *out1* :: (*share-1* × *share-1*) ⇒ (*share-2* × *share-2*) ⇒ *bool* ⇒ (*share-1* × *share-2*) *spmf*
where *out1 A B σ* = *do* {
let (*a1*,*a2*) = *A*;
let (*b1*,*b2*) = *B*;
return-spmf (*σ*, *σ* ⊕ ((*a1* ⊕ *b1*) ∧ (*a2* ⊕ *b2*)))}

definition *S1-and′* :: (*share-1* × *share-1*) ⇒ (*share-2* × *share-2*) ⇒ *bool* ⇒ (((*bool* × *bool*) × *bool* × *′v-14-OT1*) × (*share-1* × *share-2*)) *spmf*
where *S1-and′ A B σ* = *do* {
let (*a1*,*a2*) = *A*;
let (*b1*,*b2*) = *B*;
let s0 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*));
let s1 = *σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*));
let s2 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*));
let s3 = *σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*));
V ← *S1-14-OT* (*s0*,*s1*,*s2*,*s3*) ();
return-spmf (((*a1*,*a2*), *σ*, *V*), (*σ*, *σ* ⊕ ((*a1* ⊕ *b1*) ∧ (*a2* ⊕ *b2*))))}

lemma *sec-ex-P1-and*:
shows ∃ (*A* :: *′v-14-OT1* ⇒ *bool* ⇒ *bool spmf*).
|*spmf* ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ≫= (λ (*s1*, *s2*). (*S1-and′* (*a1*,*a2*) (*b1*,*b2*) *s1*)
≫= (*D* :: (((*bool* × *bool*) × *bool* × *′v-14-OT1*) × (*share-1* × *share-2*)) ⇒ *bool spmf*))) *True* − *spmf* ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ≫= *D*) *True*| =
|*spmf* (*coin-spmf* ≫= (λ *σ*. *S1-14-OT* ((*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*)))) ()
≫= (λ *view*. *A view σ*))) *True*
− *spmf* (*coin-spmf* ≫= (λ *σ*. *R1-14-OT* ((*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*)))) (*b1*, *b2*)
≫= (λ *view*. *A view σ*))) *True*|
including *monad-normalisation*
proof−
define *A′* **where** *A′* == λ *view σ*. (*D* (((*a1*,*a2*), *σ*, *view*), (*σ*, *σ* ⊕ ((*a1* ⊕ *b1*) ∧ (*a2* ⊕ *b2*)))))
have |*spmf* ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ≫= (λ (*s1*, *s2*). (*S1-and′* (*a1*,*a2*) (*b1*,*b2*) *s1*)
≫= (*D* :: (((*bool* × *bool*) × *bool* × *′v-14-OT1*) × (*share-1* × *share-2*)) ⇒ *bool spmf*))) *True* −
spmf ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ≫= (*D* :: (((*bool* × *bool*) × *bool* × *′v-14-OT1*) × (*bool* × *bool*)) ⇒ *bool spmf*)) *True*| =
|*spmf* (*coin-spmf* ≫= (λ *σ* :: *bool*. *S1-14-OT* ((*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*)))) ()
≫= (λ *view*. *A′ view σ*))) *True* − *spmf* (*coin-spmf* ≫= (λ *σ*. *R1-14-OT* ((*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*))), (*σ* ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*)))) (*b1*, *b2*)
≫= (λ *view*. *A′ view σ*))) *True*|
by(*auto simp add*: *S1-and′-def A′-def and-funct-def and-R1-def Let-def split-def correct-14*[*symmetric*] *funct-14*; *cases a1*; *cases a2*; *cases b1*; *cases b2*; *auto*)
then show *?thesis* **by** *auto*
qed

lemma bound-14-OT:
|spmf (coin-spmf ⤜ (λ σ. S1-14-OT ((σ ⊕ ((a1 ⊕ False) ∧ (a2 ⊕ False))), (σ ⊕ ((a1 ⊕ False) ∧ (a2 ⊕ True))), (σ ⊕ ((a1 ⊕ True) ∧ (a2 ⊕ False))), (σ ⊕ ((a1 ⊕ True) ∧ (a2 ⊕ True)))) ()
⤜ (λ view. (A :: 'v-14-OT1 ⇒ bool ⇒ bool spmf) view σ))) True − spmf (coin-spmf ⤜ (λ σ. R1-14-OT ((σ ⊕ ((a1 ⊕ False) ∧ (a2 ⊕ False))), (σ ⊕ ((a1 ⊕ False) ∧ (a2 ⊕ True))), (σ ⊕ ((a1 ⊕ True) ∧ (a2 ⊕ False))), (σ ⊕ ((a1 ⊕ True) ∧ (a2 ⊕ True)))) (b1, b2)
⤜ (λ view. A view σ))) True| ≤ adv-14-OT
(**is** ?lhs ≤ adv-14-OT)
proof−
have int1: integrable (measure-spmf coin-spmf) (λx. spmf (S1-14-OT (x ⊕ (a1 ⊕ False ∧ a2 ⊕ False), x ⊕ (a1 ⊕ False ∧ a2 ⊕ True), x ⊕ (a1 ⊕ True ∧ a2 ⊕ False), x ⊕ (a1 ⊕ True ∧ a2 ⊕ True)) () ⤜ (λview. A view x)) True)
and int2: integrable (measure-spmf coin-spmf) (λx. spmf (R1-14-OT (x ⊕ (a1 ⊕ False ∧ a2 ⊕ False), x ⊕ (a1 ⊕ False ∧ a2 ⊕ True), x ⊕ (a1 ⊕ True ∧ a2 ⊕ False), x ⊕ (a1 ⊕ True ∧ a2 ⊕ True)) (b1, b2) ⤜ (λview. A view x)) True)
by(rule measure-spmf.integrable-const-bound[**where** B=1]; simp add: pmf-le-1)+
have ?lhs = |LINT x|measure-spmf coin-spmf.
spmf (S1-14-OT (x ⊕ (a1 ⊕ False ∧ a2 ⊕ False), x ⊕ (a1 ⊕ False ∧ a2 ⊕ True), x ⊕ (a1 ⊕ True ∧ a2 ⊕ False), x ⊕ (a1 ⊕ True ∧ a2 ⊕ True)) () ⤜ (λview. A view x)) True −
spmf (R1-14-OT (x ⊕ (a1 ⊕ False ∧ a2 ⊕ False), x ⊕ (a1 ⊕ False ∧ a2 ⊕ True), x ⊕ (a1 ⊕ True ∧ a2 ⊕ False), x ⊕ (a1 ⊕ True ∧ a2 ⊕ True)) (b1, b2) ⤜ (λview. A view x)) True|
apply(subst (1 2) spmf-bind) **using** int1 int2 **by** simp
also have ... ≤ LINT x|measure-spmf coin-spmf. |spmf (S1-14-OT (x = (a1 ⟶ ¬ a2), x = (a1 ⟶ a2), x = (a1 ∨ ¬ a2), x = (a1 ∨ a2)) () ⤜ (λview. A view x)) True
− spmf (R1-14-OT (x = (a1 ⟶ ¬ a2), x = (a1 ⟶ a2), x = (a1 ∨ ¬ a2), x = (a1 ∨ a2)) (b1, b2) ⤜ (λview. A view x)) True|
by(rule integral-abs-bound[THEN order-trans]; simp add: split-beta)
ultimately have ?lhs ≤ LINT x|measure-spmf coin-spmf. |spmf (S1-14-OT (x = (a1 ⟶ ¬ a2), x = (a1 ⟶ a2), x = (a1 ∨ ¬ a2), x = (a1 ∨ a2)) () ⤜ (λview. A view x)) True
− spmf (R1-14-OT (x = (a1 ⟶ ¬ a2), x = (a1 ⟶ a2), x = (a1 ∨ ¬ a2), x = (a1 ∨ a2)) (b1, b2) ⤜ (λview. A view x)) True|
by simp
also have LINT x|measure-spmf coin-spmf. |spmf (S1-14-OT (x = (a1 ⟶ ¬ a2), x = (a1 ⟶ a2), x = (a1 ∨ ¬ a2), x = (a1 ∨ a2)) () ⤜ (λview. A view x)) True
− spmf (R1-14-OT (x = (a1 ⟶ ¬ a2), x = (a1 ⟶ a2), x = (a1 ∨ ¬ a2), x = (a1 ∨ a2)) (b1, b2) ⤜ (λview. A view x)) True| ≤ adv-14-OT
apply(rule integral-mono[THEN order-trans])
apply(rule measure-spmf.integrable-const-bound[**where** B=2])
apply clarsimp
apply(rule abs-triangle-ineq4[THEN order-trans])
apply(cases a1) **apply**(cases a2)
subgoal for M

using *pmf-le-1*[*of R1-14-OT* (¬ *M*, *M*, *M*, *M*) (*b1*,*b2*) ≫= (λ *view*. *A view M*) *Some True*]
pmf-le-1[*of S1-14-OT* (¬ *M*, *M*, *M*, *M*) () ≫= (λ *view*. *A view M*) *Some True*]
by *simp*
subgoal for *M*
using *pmf-le-1*[*of R1-14-OT* (*M*, ¬ *M*, *M*, *M*) (*b1*,*b2*) ≫= (λ *view*. *A view M*) *Some True*]
pmf-le-1[*of S1-14-OT* (*M*, ¬ *M*, *M*, *M*) () ≫= (λ *view*. *A view M*) *Some True*]
by *simp*
apply(*cases a2*) **apply**(*auto*)
subgoal for *M*
using *pmf-le-1*[*of R1-14-OT* (*M*, *M*, ¬ *M*, *M*) (*b1*,*b2*) ≫= (λ *view*. *A view M*) *Some True*]
pmf-le-1[*of S1-14-OT* (*M*, *M*, ¬ *M*, *M*) () ≫= (λ *view*. *A view M*) *Some True*]
by(*simp*)
subgoal for *M*
using *pmf-le-1*[*of R1-14-OT* (*M*, *M*, *M*, ¬ *M*) (*b1*,*b2*) ≫= (λ *view*. *A view M*) *Some True*]
pmf-le-1[*of S1-14-OT* (*M*, *M*, *M*, ¬ *M*) () ≫= (λ *view*. *A view M*) *Some True*]
by(*simp*)
using *ass-adv-14-OT* **by** *fast*
ultimately show *?thesis* **by** *simp*
qed

lemma *security-and-P1*:
shows |*spmf* ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ≫= (λ (*s1*, *s2*). (*S1-and′* (*a1*,*a2*) (*b1*,*b2*) *s1*)
≫= (*D* :: (((*bool* × *bool*) × *bool* × ′*v-14-OT1*) × (*share-1* × *share-2*)) ⇒ *bool spmf*))) *True* −
spmf ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ≫= *D*) *True*| ≤ *adv-14-OT*
proof−
obtain *A* :: ′*v-14-OT1* ⇒ *bool* ⇒ *bool spmf* **where** *A*:
|*spmf* ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ≫= (λ (*s1*, *s2*). (*S1-and′* (*a1*,*a2*) (*b1*,*b2*) *s1*) ≫= *D*)) *True* − *spmf* ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ≫= *D*) *True*| =
|*spmf* (*coin-spmf* ≫= (λ σ. *S1-14-OT* ((σ ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*))), (σ ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*))), (σ ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*))), (σ ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*)))) ()
≫= (λ *view*. *A view* σ))) *True* − *spmf* (*coin-spmf*
≫= (λ σ. *R1-14-OT* ((σ ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*))), (σ ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*))), (σ ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*))), (σ ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*)))) (*b1*, *b2*)
≫= (λ *view*. *A view* σ))) *True*|
using *sec-ex-P1-and* **by** *blast*
then show *?thesis* **using** *bound-14-OT*[*of a1 a2 A b1 b2*] **by** *metis*
qed

lemma *security-and-P1'*:
  shows |*spmf* ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ⤜ *D*) *True* −
    spmf ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ⤜ (λ (*s1*, *s2*). (*S1-and'* (*a1*,*a2*) (*b1*,*b2*) *s1*)
    ⤜ (*D* :: (((*bool* × *bool*) × *bool* × *'v-14-OT1*) × (*share-1* × *share-2*)) ⇒ *bool spmf*))) *True*| ≤ *adv-14-OT*
proof−
  have |*spmf* ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ⤜ *D*) *True* −
    spmf ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ⤜ (λ (*s1*, *s2*). (*S1-and'* (*a1*,*a2*) (*b1*,*b2*) *s1*)
    ⤜ (*D* :: (((*bool* × *bool*) × *bool* × *'v-14-OT1*) × (*share-1* × *share-2*)) ⇒ *bool spmf*))) *True*| =
    |*spmf* ((*and-funct* (*a1*, *a2*) (*b1*,*b2*)) ⤜ (λ (*s1*, *s2*). (*S1-and'* (*a1*,*a2*) (*b1*,*b2*) *s1*)
    ⤜ (*D* :: (((*bool* × *bool*) × *bool* × *'v-14-OT1*) × (*share-1* × *share-2*)) ⇒ *bool spmf*))) *True* −
    spmf ((*and-R1* (*a1*, *a2*) (*b1*,*b2*)) ⤜ *D*) *True*| **using** *abs-minus-commute* **by** *blast*
  thus *?thesis* **using** *security-and-P1* **by** *simp*
qed

definition *and-R2* :: (*share-1* × *share-2*) ⇒ (*share-2* × *share-1*) ⇒ (((*bool* × *bool*) × *'v-14-OT2*) × (*share-1* × *share-2*)) *spmf*
  where *and-R2 A B* = *do* {
    let (*a1*, *a2*) = *A*;
    let (*b1*,*b2*) = *B*;
    σ ← *coin-spmf*;
    let s0 = σ ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *False*));
    let s1 = σ ⊕ ((*a1* ⊕ *False*) ∧ (*a2* ⊕ *True*));
    let s2 = σ ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *False*));
    let s3 = σ ⊕ ((*a1* ⊕ *True*) ∧ (*a2* ⊕ *True*));
    (-, *s*) ← *protocol-14-OT* (*s0*,*s1*,*s2*,*s3*) *B*;
    V ← *R2-14-OT* (*s0*,*s1*,*s2*,*s3*) *B*;
    return-spmf ((*B*, *V*), (σ, *s*))}

lemma *lossless-and-R2*: *lossless-spmf* (*and-R2 A B*)
  apply(*simp add*: *and-R2-def split-def lossless-R2-14-OT lossless-protocol-14-OT Let-def*)
  by (*metis lossless-R2-14-OT prod.collapse*)

definition *S2-and* :: (*share-1* × *share-2*) ⇒ *bool* ⇒ (((*bool* × *bool*) × *'v-14-OT2*)) *spmf*
  where *S2-and B s2* = *do* {
    let (*a2*,*b2*) = *B*;
    V :: *'v-14-OT2* ← *S2-14-OT* (*a2*,*b2*) *s2*;
    return-spmf ((*B*, *V*))}

definition *out2* :: (*share-1* × *share-2*) ⇒ (*share-1* × *share-2*) ⇒ *bool* ⇒ (*share-1*

× *share-2*) *spmf*
  where *out2 B A s2* = *do* {
    let (*a1*, *b1*) = *A*;
    let (*a2*,*b2*) = *B*;
    let s1 = *s2* ⊕ ((*a1* ⊕ *a2*) ∧ (*b1* ⊕ *b2*));
    return-spmf (*s1*, *s2*)}

definition *S2-and′* :: (*share-1* × *share-2*) ⇒ (*share-1* × *share-2*) ⇒ *bool* ⇒ (((*bool* × *bool*) × *′v-14-OT2*) × (*share-1* × *share-2*)) *spmf*
  where *S2-and′ B A s2* = *do* {
    let (*a1*, *a2*) = *A*;
    let (*b1*,*b2*) = *B*;
    V :: *′v-14-OT2* ← *S2-14-OT B s2*;
    let s1 = *s2* ⊕ ((*a1* ⊕ *b1*) ∧ (*a2* ⊕ *b2*));
    return-spmf ((*B*, *V*), *s1*, *s2*)}

lemma *lossless-S2-and*: *lossless-spmf* (*S2-and B s2*)
  apply(*simp add*: *S2-and-def split-def*)
  by(*metis prod.collapse lossless-S2-14-OT*)

sublocale *and-secret-sharing*: *sim-non-det-def and-R1 S1-and out1 and-R2 S2-and out2 and-funct* **.**

lemma *ideal-S1-and*: *and-secret-sharing.Ideal1* (*a1*, *b1*) (*a2*, *b2*) *s2* = *S1-and′* (*a1*, *b1*) (*a2*, *b2*) *s2*
  by(*simp add*: *Let-def and-secret-sharing.Ideal1-def S1-and′-def split-def out1-def S1-and-def*)

lemma *and-P2-security*: *and-secret-sharing.perfect-sec-P2 m1 m2*
proof−
  have *and-R2* (*a1*, *b1*) (*a2*, *b2*) = *and-funct* (*a1*, *b1*) (*a2*, *b2*) ⤜ (λ(*s1*, *s2*). *and-secret-sharing.Ideal2* (*a2*, *b2*) (*a1*, *b1*) *s2*)
    for *a1 a2 b1 b2*
    apply(*auto simp add*: *split-def inf-th-14-OT-P4 S2-and′-def and-R2-def and-funct-def Let-def correct-14*[*symmetric*] *and-secret-sharing.Ideal2-def S2-and-def out2-def*)
    apply(*simp only*: *funct-14*)
    apply *auto*
    by(*cases b1*;*cases b2*; *cases a1*; *cases a2*; *auto*)
  thus *?thesis*
    by(*simp add*: *and-secret-sharing.perfect-sec-P2-def*; *metis prod.collapse*)
qed

lemma *and-P1-security*: *and-secret-sharing.adv-P1 m1 m2 D* ≤ *adv-14-OT*
proof−
  have |*spmf* (*and-R1* (*a1*, *a2*) (*b1*, *b2*) ⤜ *D*) *True* −
        spmf (*and-funct* (*a1*, *a2*) (*b1*, *b2*) ⤜ (λ(*s1*, *s2*).
          and-secret-sharing.Ideal1 (*a1*, *a2*) (*b1*, *b2*) *s1* ⤜ *D*)) *True*|
            ≤ *adv-14-OT* **for** *a1 a2 b1 b2*
    using *security-and-P1′ ideal-S1-and prod.collapse* **by** *simp*

thus *?thesis*
by(*simp add*: *and-secret-sharing.adv-P1-def*; *metis prod.collapse*)
qed

definition $F = \{$*and-evaluate*, *xor-evaluate*$\}$

lemma *share-reconstruct-xor*: *share x* $\gg=$ $(\lambda(a1, a2)$. *share y* $\gg=$ $(\lambda(b1, b2)$.
xor-protocol $(a1, b1)$ $(a2, b2)$ $\gg=$ $(\lambda(a, b)$.
reconstruct $(a, b)))) =$ *xor-evaluate x y*
proof−
have $(((ya = (x = ya)) = (yb = (y = (\neg\ yb))))) = (x = (\neg\ y))$ **for** *ya yb* **by** *auto*
thus *?thesis*
by(*simp add*: *xor-protocol-def share-def reconstruct-def xor-evaluate-def bind-spmf-const*)
qed

sublocale *share-correct*: *secret-sharing-scheme share reconstruct F* **.**

lemma *share-correct.sharing-correct input*
by(*simp add*: *share-correct.sharing-correct-def reconstruct-share*)

lemma *share-correct.correct-share-eval input1 input2*
unfolding *share-correct.correct-share-eval-def*
apply(*auto simp add*: *F-def*)
using *share-and-reconstruct* **apply** *auto*
using *share-reconstruct-xor* **by** *force*

end

locale *gmw-asym* =
fixes *S1-14-OT* :: *nat* ⇒ *msgs-14-OT* ⇒ *unit* ⇒ *'v-14-OT1 spmf*
and *R1-14-OT* :: *nat* ⇒ *msgs-14-OT* ⇒ *choice-14-OT* ⇒ *'v-14-OT1 spmf*
and *S2-14-OT* :: *nat* ⇒ *choice-14-OT* ⇒ *bool* ⇒ *'v-14-OT2 spmf*
and *R2-14-OT* :: *nat* ⇒ *msgs-14-OT* ⇒ *choice-14-OT* ⇒ *'v-14-OT2 spmf*
and *protocol-14-OT* :: *nat* ⇒ *msgs-14-OT* ⇒ *choice-14-OT* ⇒ (*unit* × *bool*) *spmf*
and *adv-14-OT* :: *nat* ⇒ *real*
assumes *gmw-base*: ⋀ (*n*::*nat*). *gmw-base* (*S1-14-OT n*) (*R1-14-OT n*) (*S2-14-OT n*) (*R2-14-OT n*) (*protocol-14-OT n*) (*adv-14-OT n*)
begin

sublocale *gmw-base* (*S1-14-OT n*) (*R1-14-OT n*) (*S2-14-OT n*) (*R2-14-OT n*) (*protocol-14-OT n*) (*adv-14-OT n*)
by (*simp add*: *gmw-base*)

lemma *xor-sim-det.perfect-sec-P1 m1 m2*
by (*simp add*: *P1-xor-inf-th xor-sim-det.perfect-sec-P1-def*)

lemma *xor-sim-det.perfect-sec-P2 m1 m2*

by (*simp add*: *P2-xor-inf-th xor-sim-det.perfect-sec-P2-def*)

lemma *and-P1-adv-negligible*:
assumes *negligible* (λ *n*. *adv-14-OT n*)
shows *negligible* (λ *n*. *and-secret-sharing.adv-P1 n m1 m2 D*)
proof−
have *and-secret-sharing.adv-P1 n m1 m2 D* $\leq$ *adv-14-OT n* **for** *n*
by (*simp add*: *and-P1-security*)
thus *?thesis*
using *and-secret-sharing.adv-P1-def assms negligible-le* **by** *auto*
qed

lemma *and-P2-security*: *and-secret-sharing.perfect-sec-P2 n m1 m2*
by (*simp add*: *and-P2-security*)

end

end

2.8 Secure multiplication protocol

theory *Secure-Multiplication* **imports**
CryptHOL.Cyclic-Group-SPMF
Uniform-Sampling
Semi-Honest-Def
begin

locale *secure-mult* =
fixes *q* :: *nat*
assumes *q-gt-0*: *q* > *0*
and *prime q*
begin

type-synonym *real-view* = *nat* ⇒ *nat* ⇒ ((*nat* × *nat* × *nat* × *nat*) × *nat* × *nat*) *spmf*
type-synonym *sim* = *nat* ⇒ *nat* ⇒ ((*nat* × *nat* × *nat* × *nat*) × *nat* × *nat*) *spmf*

lemma *samp-uni-set-spmf* [*simp*]: *set-spmf* (*sample-uniform q*) = {..< *q*}
by(*simp add*: *sample-uniform-def*)

definition *funct* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat*) *spmf*
where *funct x y* = *do* {
s ← *sample-uniform q*;
return-spmf (*s*, (*x*∗*y* + (*q* − *s*)) *mod q*)}

definition *TI* :: ((*nat* × *nat*) × (*nat* × *nat*)) *spmf*
where *TI* = *do* {
a ← *sample-uniform q*;

```
    b ← sample-uniform q;
    r ← sample-uniform q;
    return-spmf ((a, r), (b, ((a*b + (q − r)) mod q)))}

definition out :: nat ⇒ nat ⇒ (nat × nat) spmf
  where out x y = do {
    ((c1,d1),(c2,d2)) ← TI;
    let e2 = (x + c1) mod q;
    let e1 = (y + (q − c2)) mod q;
    return-spmf (((x*e1 + (q − d1)) mod q), ((e2 * c2 + (q − d2)) mod q))}

definition R1 :: real-view
  where R1 x y = do {
    ((c1, d1), (c2, d2)) ← TI;
    let e2 = (x + c1) mod q;
    let e1 = (y + (q − c2)) mod q;
    let s1 = (x*e1 + (q − d1)) mod q;
    let s2 = (e2 * c2 + (q − d2)) mod q;
    return-spmf ((x, c1, d1, e1), s1, s2)}

definition S1 :: nat ⇒ nat ⇒ (nat × nat × nat × nat) spmf
  where S1 x s1 = do {
    a :: nat ← sample-uniform q;
    e1 ← sample-uniform q;
    let d1 = (x*e1 + (q − s1)) mod q;
    return-spmf (x, a, d1, e1)}

definition Out1 :: nat ⇒ nat ⇒ nat ⇒ (nat × nat) spmf
  where Out1 x y s1 = do {
    let s2 = (x*y + (q − s1)) mod q;
    return-spmf (s1,s2)}

definition R2 :: real-view
  where R2 x y = do {
    ((c1, d1), (c2, d2)) ← TI;
    let e2 = (x + c1) mod q;
    let e1 = (y + (q − c2)) mod q;
    let s1 = (x*e1 + (q − d1)) mod q;
    let s2 = (e2 * c2 + (q − d2)) mod q;
    return-spmf ((y, c2, d2, e2), s1, s2)}

definition S2 :: nat ⇒ nat ⇒ (nat × nat × nat × nat) spmf
  where S2 y s2 = do {
    b ← sample-uniform q;
    e2 ← sample-uniform q;
    let d2 = (e2*b + (q − s2)) mod q;
    return-spmf (y, b, d2, e2)}

definition Out2 :: nat ⇒ nat ⇒ nat ⇒ (nat × nat) spmf
```

where *Out2 y x s2* = *do* {
let s1 = (*x*∗*y* + (*q* − *s2*)) *mod q*;
return-spmf (*s1*,*s2*)}

definition *Ideal2* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ ((*nat* × *nat* × *nat* × *nat*) × (*nat* × *nat*)) *spmf*
where *Ideal2 y x out2* = *do* {
view2 :: (*nat* × *nat* × *nat* × *nat*) ← *S2 y out2*;
out2 ← *Out2 y x out2*;
return-spmf (*view2*, *out2*)}

sublocale *sim-non-det-def*: *sim-non-det-def R1 S1 Out1 R2 S2 Out2 funct* **.**

lemma *minus-mod*:
assumes *a* > *b*
shows [*a* − *b mod q* = *a* − *b*] (*mod q*)
by(*metis cong-diff-nat cong-def le-trans less-or-eq-imp-le assms mod-less-eq-dividend mod-mod-trivial*)

lemma *q-cong*:[*a* = *q* + *a*] (*mod q*)
by (*simp add*: *cong-def*)

lemma *q-cong-reverse*: [*q* + *a* = *a*] (*mod q*)
by (*simp add*: *cong-def*)

lemma *qq-cong*: [*a* = *q*∗*q* + *a*] (*mod q*)
by (*simp add*: *cong-def*)

lemma *minus-q-mult-cancel*:
assumes [*a* = *e* + *b* − *q* ∗ *c* − *d*] (*mod q*)
and *e* + *b* − *d* > *0*
and *e* + *b* − *q* ∗ *c* − *d* > *0*
shows [*a* = *e* + *b* − *d*] (*mod q*)
proof−
have *a mod q* = (*e* + *b* − *q* ∗ *c* − *d*) *mod q*
using *assms*(*1*) *cong-def* **by** *blast*
then have *a mod q* = (*e* + *b* − *d*) *mod q*
by (*metis* (*no-types*) *add-cancel-left-left assms*(*3*) *diff-commute diff-is-0-eq′ linordered-semidom-class.add-diff-inverse mod-add-left-eq mod-mult-self1-is-0 nat-less-le*)
then show *?thesis*
using *cong-def* **by** *blast*
qed

lemma *s1-s2*:
assumes *x* < *q a* < *q b* < *q* **and** *r*:*r* < *q y* < *q*
shows ((*x* + *a*) *mod q* ∗ *b* + *q* − (*a* ∗ *b* + *q* − *r*) *mod q*) *mod q* =
(*x* ∗ *y* + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*
proof−
have *s*: (*x* ∗ *y* + (*q* − (*x* ∗ ((*y* + (*q* − *b*)) *mod q*) + (*q* − *r*)) *mod q*)) *mod q*

$= ((x + a)\ mod\ q * b + (q - (a * b + (q - r))\ mod\ q))\ mod\ q$
proof−
have *lhs*: $(x * y + (q - (x * ((y + (q - b))\ mod\ q) + (q - r))\ mod\ q))\ mod\ q = (x*b + r)\ mod\ q$
proof−
let $?h = (x * y + (q - (x * ((y + (q - b))\ mod\ q) + (q - r))\ mod\ q))\ mod\ q$
have $[?h = x * y + q - (x * ((y + (q - b))\ mod\ q) + (q - r))\ mod\ q]\ (mod\ q)$
by(*simp add*: *assms*(*1*) *cong-def q-gt-0*)
then have $[?h = x * y + q - (x * (y + (q - b)) + (q - r))\ mod\ q]\ (mod\ q)$
by (*metis mod-add-left-eq mod-mult-right-eq*)
then have *no-qq*: $[?h = x * y + q - (x * y + x * (q - b) + (q - r))\ mod\ q]\ (mod\ q)$
by(*metis distrib-left*)
then have $[?h = q*q + x * y + q - (x * y + x * (q - b) + (q - r))\ mod\ q]\ (mod\ q)$
proof−
have $[x * y + q - (x * y + x * (q - b) + (q - r))\ mod\ q = q*q + x * y + q - (x * y + x * (q - b) + (q - r))\ mod\ q]\ (mod\ q)$
by (*smt qq-cong add.assoc cong-diff-nat cong-def le-add2 le-trans mod-le-divisor q-gt-0*)
then show *?thesis* **using** *cong-trans no-qq* **by** *blast*
qed
then have *mod*: $[?h = q + q*q + x * y + q - (x * y + x * (q - b) + (q - r))\ mod\ q]\ (mod\ q)$
by (*smt Nat.add-diff-assoc cong-def add.assoc add.commute le-add2 le-trans mod-add-self2 mod-le-divisor q-gt-0*)
then have $[?h = q + q*q + x * y + q - (x * y + x * (q - b) + (q - r))]\ (mod\ q)$
proof−
have *1*: $q \geq q - b$ **using** *assms* **by** *simp*
then have $q*q \geq x*(q-b)\ q \geq q - r$ **using** *1 assms*
apply (*auto simp add*: *mult-strict-mono*)
by (*simp add*: *mult-le-mono*)
then have $q + q*q + x * y + q > x * y + x * (q - b) + (q - r)$
using *assms*(*5*) **by** *linarith*
then have $[q + q*q + x * y + q - (x * y + x * (q - b) + (q - r))\ mod\ q = q + q*q + x * y + q - (x * y + x * (q - b) + (q - r))]\ (mod\ q)$
using *minus-mod* **by** *blast*
then show *?thesis* **using** *mod* **using** *cong-trans* **by** *blast*
qed
then have $[?h = q + q*q + x * y + q - (x * y + (x * q - x*b) + (q - r))]\ (mod\ q)$
by (*simp add*: *right-diff-distrib'*)
then have $[?h = q + q*q + x * y + q - x * y - (x * q - x*b) - (q - r)]\ (mod\ q)$
by *simp*
then have *mod'*: $[?h = q + q*q + q - (x * q - x*b) - (q - r)]\ (mod\ q)$
by(*simp add*: *add.commute*)

then have *neg*: [*?h* = *q* + *q*∗*q* + *q* − *x* ∗ *q* + *x*∗*b* − (*q* − *r*)] (*mod q*)
proof−
have [*q* + *q*∗*q* + *q* − (*x* ∗ *q* − *x*∗*b*) − (*q* − *r*) = *q* + *q*∗*q* + *q* − *x* ∗ *q* + *x*∗*b* − (*q* − *r*)] (*mod q*)
proof(*cases x* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *x* ∗ *q* − *x*∗*b* > *0* **using** *False assms* **by** *simp*
also have *q* + *q*∗*q* + *q* − *x* ∗ *q* > *0*
by (*metis assms*(*1*) *add.commute diff-mult-distrib2 less-Suc-eq mult.commute mult-Suc-right nat-0-less-mult-iff q-gt-0 zero-less-diff*)
ultimately show *?thesis* **by** *simp*
qed
then show *?thesis* **using** *mod′ cong-trans* **by** *blast*
qed
then have [*?h* = *q* + *q*∗*q* + *q* + *x*∗*b* − (*q* − *r*)] (*mod q*)
proof−
have [*q* + *q*∗*q* + *q* − *x* ∗ *q* + *x*∗*b* − (*q* − *r*) = *q* + *q*∗*q* + *q* + *x*∗*b* − (*q* − *r*)] (*mod q*)
proof(*cases x* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *q*∗*q* > *x*∗*q*
using *False assms*
by (*simp add*: *mult-strict-mono*)
then have *1*: *q* + *q*∗*q* + *q* − *x* ∗ *q* + *x*∗*b* − (*q* − *r*) > *0*
by *linarith*
then have *2*: *q* + *q*∗*q* + *q* + *x*∗*b* − (*q* − *r*) > *0* **by** *simp*
then show *?thesis*
by (*smt 1 2 Nat.add-diff-assoc2* ‹*x* ∗ *q* < *q* ∗ *q*› *add-cancel-left-left add-diff-inverse-nat*
le-add1 le-add2 le-trans less-imp-add-positive less-numeral-extra(*3*) *minus-mod*
minus-q-mult-cancel mod-if mult.commute q-gt-0)
qed
then show *?thesis* **using** *cong-trans neg* **by** *blast*
qed
then have [*?h* = *q* + *q*∗*q* + *q* + *x*∗*b* − *q* + *r*] (*mod q*)
by (*metis r*(*1*) *Nat.add-diff-assoc2 Nat.diff-diff-right le-add2 less-imp-le-nat semiring-normalization-rules*(*23*))
then have [*?h* = *q* + *q*∗*q* + *q* + *x*∗*b* + *r*] (*mod q*)
apply(*simp add*: *cong-def*)
by (*metis* (*no-types, lifting*) *add.assoc add.commute add-diff-cancel-right′ diff-is-0-eq′ mod-if mod-le-divisor q-gt-0*)
then have [*?h* = *x*∗*b* + *r*] (*mod q*)

apply(*simp add*: *cong-def*)
by (*metis mod-add-cong mod-add-self1 mod-mult-self1*)
then show *?thesis* **by** (*simp add*: *cong-def assms*)
qed
also have *rhs*: $((x + a)$ *mod* $q * b + (q - (a * b + (q - r))$ *mod* $q))$ *mod* q $= (x{*}b + r)$ *mod* q
proof−
let $?h = ((x + a)$ *mod* $q * b + (q - (a * b + (q - r))$ *mod* $q))$ *mod* q
have $[?h = (x + a)$ *mod* $q * b + q - (a * b + (q - r))$ *mod* $q]$ (*mod* q)
by (*simp add*: *q-gt-0 assms*(*1*) *cong-def*)
then have $[?h = (x + a) * b + q - (a * b + (q - r))$ *mod* $q]$ (*mod* q)
by (*smt Nat.add-diff-assoc cong-def mod-add-cong mod-le-divisor mod-mult-left-eq q-gt-0 assms*)
then have $[?h = x{*}b + a{*}b + q - (a * b + (q - r))$ *mod* $q]$ (*mod* q)
by(*metis distrib-right*)
then have *mod*: $[?h = q + x{*}b + a{*}b + q - (a * b + (q - r))$ *mod* $q]$ (*mod* q)
by (*smt Nat.add-diff-assoc cong-def add.assoc add.commute le-add2 le-trans mod-add-self2 mod-le-divisor q-gt-0*)
then have $[?h = q + x{*}b + a{*}b + q - (a * b + (q - r))]$ (*mod* q) **using** *q-cong assms*(*1*)
proof−
have *ge*: $q + x{*}b + a{*}b + q > (a * b + (q - r))$ **using** *assms* **by** *simp*
with *minus-mod* [*of* ‹$a * b + (q - r)$› ‹$q + x * b + a * b + q$›]
have $[q + x{*}b + a{*}b + q - (a * b + (q - r))$ *mod* $q = q + x{*}b + a{*}b + q - (a * b + (q - r))]$ (*mod* q)
by *simp*
then show *?thesis* **using** *mod cong-trans* **by** *blast*
qed
then have $[?h = q + x{*}b + q - (q - r)]$ (*mod* q)
by (*simp add*: *add.commute*)
then have $[?h = q + x{*}b + q - q + r]$ (*mod* q)
by (*metis Nat.add-diff-assoc2 Nat.diff-diff-right r*(*1*) *le-add2 less-imp-le-nat*)
then have $[?h = q + x{*}b + r]$ (*mod* q) **by** *simp*
then have $[?h = q + (x{*}b + r)]$ (*mod* q)
using *add.assoc* **by** *metis*
then have $[?h = x{*}b + r]$ (*mod* q)
using *cong-def q-cong-reverse* **by** *metis*
then show *?thesis* **by** (*simp add*: *cong-def assms*(*1*))
qed
ultimately show *?thesis* **by** *simp*
qed
have *lhs*:$((x + a)$ *mod* $q * b + q - (a * b + q - r)$ *mod* $q)$ *mod* $q = ((x + a)$ *mod* $q * b + (q - (a * b + (q - r))$ *mod* $q))$ *mod* q
using *assms* **by** *simp*
have *rhs*: $(x * y + q - (x * ((y + q - b)$ *mod* $q) + q - r)$ *mod* $q)$ *mod* $q = (x * y + (q - (x * ((y + (q - b))$ *mod* $q) + (q - r))$ *mod* $q))$ *mod* q
using *assms* **by** *simp*
have $((x + a)$ *mod* $q * b + (q - (a * b + (q - r))$ *mod* $q))$ *mod* $q = (x * y +$

$(q - (x * ((y + (q - b)) \; mod \; q) + (q - r)) \; mod \; q)) \; mod \; q$
    using *assms s[symmetric]* **by** *blast*
  then show *?thesis* **using** *lhs rhs*
    by *metis*
qed

lemma *s1-s2-P2*:
  assumes $x < q$ $xa < q$ $xb < q$ $xc < q$ $y < q$
  shows $((y, xa, (xb * xa + q - xc) \; mod \; q, (x + xb) \; mod \; q), (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q, ((x + xb) \; mod \; q * xa + q - (xb * xa + q - xc) \; mod \; q) \; mod \; q) =$
          $((y, xa, (xb * xa + q - xc) \; mod \; q, (x + xb) \; mod \; q), (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q, (x * y + q - (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q) \; mod \; q)$
  using *assms s1-s2* **by** *metis*

lemma *c1*:
  assumes $e2 = (x + c1) \; mod \; q$
    and $x < q$ $c1 < q$
  shows $c1 = (e2 + q - x) \; mod \; q$
proof−
  have $[e2 + q = x + c1] \; (mod \; q)$ **by**(*simp add*: *assms cong-def*)
  then have $[e2 + q - x = c1] \; (mod \; q)$
  proof−
    have $e2 + q \geq x$ **using** *assms* **by** *simp*
    then show *?thesis*
     by (*metis* ‹$[e2 + q = x + c1] \; (mod \; q)$› *cong-add-lcancel-nat le-add-diff-inverse*)
  qed
  then show *?thesis* **by**(*simp add*: *cong-def assms*)
qed

lemma *c1-P2*:
  assumes $xb < q$ $xa < q$ $xc < q$ $x < q$
  shows $((y, xa, (xb * xa + q - xc) \; mod \; q, (x + xb) \; mod \; q), (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q, (x * y + q - (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q) \; mod \; q) =$
          $((y, xa, (((x + xb) \; mod \; q + q - x) \; mod \; q * xa + q - xc) \; mod \; q, (x + xb) \; mod \; q), (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q, (x * y + q - (x * ((y + q - xa) \; mod \; q) + q - xc) \; mod \; q) \; mod \; q)$
proof−
  have $(xb * xa + q - xc) \; mod \; q = (((x + xb) \; mod \; q + q - x) \; mod \; q * xa + q - xc) \; mod \; q$
    using *assms c1* **by** *simp*
  then show *?thesis*
    using *assms* **by** *metis*
qed

lemma *minus-mod-cancel*:
  assumes $a - b > 0$ $a - b \; mod \; q > 0$

shows [$a - b + c = a - b \bmod q + c$] ($mod\ q$)
proof−
have ($b - b\ mod\ q + (a - b)$) $mod\ q$ = ($0 + (a - b)$) $mod\ q$
using *cong-def mod-add-cong neq0-conv q-gt-0*
by (*simp add*: *minus-mod-eq-mult-div*)
with ‹$a - b > 0$› **show** *?thesis*
by (*simp add*: *cong-def mod-add-left-eq* [*symmetric*, *of* ‹$a - b\ mod\ q$› $c\ q$])
(*simp add*: *mod-simps*)
qed

lemma *d2*:
assumes *d2*: $d2 = (((e2 + q - x)\ mod\ q)*b + (q - r))\ mod\ q$
and *s1*: $s1 = (x*((y + (q - b))\ mod\ q) + (q - r))\ mod\ q$
and *s2*: $s2 = (x*y + (q - s1))\ mod\ q$
and *x*: $x < q$
and *y*: $y < q$
and *r*: $r < q$
and *b*: $b < q$
and *e2*: $e2 < q$
shows $d2 = (e2*b + (q - s2))\ mod\ q$
proof−
have *s1-le-q*: $s1 < q$
using *s1 q-gt-0* **by** *simp*
have *s2-le-q*: $s2 < q$
using *s2 q-gt-0* **by** *simp*
have *xb*: $(x*b)\ mod\ q = (s2 + (q - r))\ mod\ q$
proof−
have $s1 = (x*(y + (q - b)) + (q - r))\ mod\ q$ **using** *s1 b*
by (*metis mod-add-left-eq mod-mult-right-eq*)
then have *s1-dist*: $s1 = (x*y + x*(q - b) + (q - r))\ mod\ q$
by(*metis distrib-left*)
then have $s1 = (x*y + x*q - x*b + (q - r))\ mod\ q$
using *distrib-left b diff-mult-distrib2* **by** *auto*
then have [$s1 = x*y + x*q - x*b + (q - r)$] ($mod\ q$)
by(*simp add*: *cong-def*)
then have [$s1 + x * b = x*y + x*q - x*b + x*b + (q - r)$] ($mod\ q$)
by (*metis add.commute add.left-commute cong-add-lcancel-nat*)
then have [$s1 + x*b = x*y + x*q + (q - r)$] ($mod\ q$)
using *b* **by** (*simp add*: *algebra-simps*)
(*metis add-diff-inverse-nat diff-diff-left diff-mult-distrib2 less-imp-add-positive mult.commute not-add-less1 zero-less-diff*)
then have *s1-xb*: [$s1 + x*b = q + x*y + x*q + (q - r)$] ($mod\ q$)
by (*smt mod-add-cong mod-add-self1 cong-def*)
then have [$x*b = q + x*y + x*q + (q - r) - s1$] ($mod\ q$)
proof−
have $q + x*y + x*q + (q - r) - s1 > 0$ **using** *s1-le-q* **by** *simp*
then show *?thesis*
by (*metis add-diff-inverse-nat less-numeral-extra*(*3*) *s1-xb cong-add-lcancel-nat nat-diff-split*)

qed
then have $[x*b = x*y + x*q + (q - r) + q - s1]$ $(mod\ q)$
by (*metis add.assoc add.commute*)
then have $[x*b = x*y + (q - r) + q - s1]$ $(mod\ q)$
by (*smt Nat.add-diff-assoc cong-def less-imp-le-nat mod-mult-self1 s1-le-q semiring-normalization-rules*(*23*))
then have $(x*b)$ *mod* $q = (x*y + (q - r) + q - s1)$ *mod* q
by(*simp add*: *cong-def*)
then have $(x*b)$ *mod* $q = (x*y + (q - r) + (q - s1))$ *mod* q
using *add.assoc s1-le-q* **by** *auto*
then have $(x*b)$ *mod* $q = (x*y + (q - s1) + (q - r))$ *mod* q
using *add.commute* **by** *presburger*
then show *?thesis* **using** *s2* **by** *presburger*
qed
have $d2 = (((e2 + q - x)$ *mod* $q)*b + (q - r))$ *mod* q
using *d2* **by** *simp*
then have $d2 = (((e2 + q - x))*b + (q - r))$ *mod* q
using *mod-add-cong mod-mult-left-eq* **by** *blast*
then have $d2 = (e2*b + q*b - x*b + (q - r))$ *mod* q
by (*simp add*: *distrib-right diff-mult-distrib*)
then have *a*: $[d2 = e2*b + q*b - x*b + (q - r)]$ $(mod\ q)$
by(*simp add*: *cong-def*)
then have *b*:$[d2 = q + q + e2*b + q*b - x*b + (q - r)]$ $(mod\ q)$
proof−
have $[e2*b + q*b - x*b + (q - r) = q + q + e2*b + q*b - x*b + (q - r)]$ $(mod\ q)$
by (*smt assms Nat.add-diff-assoc add.commute cong-def less-imp-le-nat mod-add-self2 mult.commute nat-mult-le-cancel-disj semiring-normalization-rules*(*23*))
then show *?thesis* **using** *cong-trans a* **by** *blast*
qed
then have $[d2 = q + q + e2*b + q*b - (x*b)$ *mod* $q + (q - r)]$ $(mod\ q)$
proof−
have $[q + q + e2*b + q*b - (x*b) + (q - r) = q + q + e2*b + q*b - (x*b)$ *mod* $q + (q - r)]$ $(mod\ q)$
proof(*cases* $b = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have $q*b - (x*b) > 0$
using *False x* **by** *simp*
then have *1*: $q + q + e2*b + q*b - (x*b) > 0$ **by** *linarith*
then have *2*:$q + q + e2*b + q*b - (x*b)$ *mod* $q > 0$
by (*simp add*: *q-gt-0 trans-less-add1*)
then show *?thesis* **using** *1 2 minus-mod-cancel* **by** *simp*
qed
then show *?thesis* **using** *cong-trans b* **by** *blast*
qed

then have *c*: [*d2* = *q* + *q* + *e2*∗*b* + *q*∗*b* − (*s2* + (*q* − *r*)) *mod q* + (*q* − *r*)] (*mod q*)
using *xb* **by** *simp*
then have [*d2* = *q* + *q* + *e2*∗*b* + *q*∗*b* − (*s2* + (*q* − *r*)) + (*q* − *r*)] (*mod q*)
proof−
have [*q* + *q* + *e2*∗*b* + *q*∗*b* − (*s2* + (*q* − *r*)) *mod q* + (*q* − *r*) = *q* + *q* + *e2*∗*b* + *q*∗*b* − (*s2* + (*q* − *r*)) + (*q* − *r*)] (*mod q*)
proof−
have *q* + *q* + *e2*∗*b* + *q*∗*b* − (*s2* + (*q* − *r*)) *mod q* > *0*
by (*metis diff-is-0-eq gr0I le-less-trans mod-less-divisor not-add-less1 q-gt-0 semiring-normalization-rules*(*23*) *trans-less-add2*)
moreover have*q* + *q* + *e2*∗*b* + *q*∗*b* − (*s2* + (*q* − *r*)) > *0*
using *s2-le-q* **by** *simp*
ultimately show *?thesis*
using *minus-mod-cancel cong-sym* **by** *blast*
qed
then show *?thesis* **using** *cong-trans c* **by** *blast*
qed
then have *d*: [*d2* = *q* + *q* + *e2*∗*b* + *q*∗*b* − *s2* − (*q* − *r*) + (*q* − *r*)] (*mod q*) **by** *simp*
then have [*d2* = *q* + *q* + *e2*∗*b* + *q*∗*b* − *s2*] (*mod q*)
proof−
have *q* + *q* + *e2*∗*b* + *q*∗*b* − *s2* − (*q* − *r*) > *0*
using *s2-le-q* **by** *simp*
then show *?thesis* **using** *d cong-trans* **by** *simp*
qed
then have [*d2* = *q* + *q* + *e2*∗*b* − *s2*] (*mod q*)
by (*smt Nat.add-diff-assoc2 cong-def less-imp-le-nat mod-mult-self1 mult.commute s2-le-q semiring-normalization-rules*(*23*) *trans-less-add2*)
then have [*d2* = *q* + *e2*∗*b* + *q* − *s2*] (*mod q*)
by(*simp add*: *add.commute add.assoc*)
then have [*d2* = *e2*∗*b* + *q* − *s2*] (*mod q*)
by (*smt Nat.add-diff-assoc2 add.commute cong-def less-imp-le-nat mod-add-self2 s2-le-q trans-less-add2*)
then have [*d2* = *e2*∗*b* + (*q* − *s2*)] (*mod q*)
by (*simp add*: *less-imp-le-nat s2-le-q*)
then show *?thesis* **by**(*simp add*: *cong-def d2*)
qed

lemma *d2-P2*:
assumes *x*: *x* < *q* **and** *y*: *y* < *q* **and** *r*: *b* < *q* **and** *b*: *e2* < *q* **and** *e2*: *r* < *q*
shows ((*y*, *b*, ((*e2* + *q* − *x*) *mod q* ∗ *b* + *q* − *r*) *mod q*, *e2*), (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*, (*x* ∗ *y* + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*) =
((*y*, *b*, (*e2* ∗ *b* + *q* − (*x* ∗ *y* + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*) *mod q*, *e2*), (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*,
(*x* ∗ *y* + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*)
proof−
have ((*e2* + *q* − *x*) *mod q* ∗ *b* + *q* − *r*) *mod q* = (*e2* ∗ *b* + *q* − (*x* ∗ *y* + *q* −

$(x * ((y + q - b) \ mod \ q) + q - r) \ mod \ q) \ mod \ q) \ mod \ q$
  (**is** *?lhs* = *?rhs*)
 proof−
  have *d2*: $(((e2 + q - x) \ mod \ q)*b + (q - r)) \ mod \ q = (e2*b + (q - ((x*y + (q - ((x*((y + (q - b)) \ mod \ q) + (q - r)) \ mod \ q))) \ mod \ q))) \ mod \ q$
   using *assms d2* **by** *blast*
  have *?lhs* = $(((e2 + q - x) \ mod \ q)*b + (q - r)) \ mod \ q$
   using *assms* **by** *simp*
  also have *?rhs* = $(e2*b + (q - ((x*y + (q - ((x*((y + (q - b)) \ mod \ q) + (q - r)) \ mod \ q))) \ mod \ q))) \ mod \ q$
   using *assms* **by** *simp*
  ultimately show *?thesis* **using** *assms d2* **by** *metis*
 qed
 then show *?thesis* **using** *assms* **by** *metis*
qed

lemma *s1*:
 assumes *s2*: $s2 = (x*y + q - s1) \ mod \ q$
  and *x*: $x < q$
  and *y*: $y < q$
  and *s1*: $s1 < q$
 shows $s1 = (x*y + q - s2) \ mod \ q$
proof−
 have *s2-le-q*:$s2 < q$ **using** *s2 q-gt-0* **by** *simp*
 have $[s2 = x*y + q - s1] \ (mod \ q)$ **by**(*simp add*: *cong-def s2*)
 then have $[s2 = x*y + q - s1] \ (mod \ q)$ **using** *add.assoc*
  by (*simp add*: *less-imp-le-nat s1*)
 then have *s1-s2*: $[s2 + s1 = x*y + q] \ (mod \ q)$
   by (*metis* (*mono-tags*, *lifting*) *cong-def le-add2 le-add-diff-inverse2 le-trans mod-add-left-eq order.strict-implies-order s1*)
 then have $[s1 = x*y + q - s2] \ (mod \ q)$
 proof−
  have $x*y + q - s2 > 0$ **using** *s2-le-q* **by** *simp*
  then show *?thesis*
   by (*metis s1-s2 add-diff-cancel-left′ cong-diff-nat cong-def le-add1 less-imp-le-nat zero-less-diff*)
 qed
 then show *?thesis* **by**(*simp add*: *cong-def s1*)
qed

lemma *s1-P2*:
 assumes *x*: $x < q$
  and *y*: $y < q$
  and $b < q$
  and $e2 < q$
  and $r < q$
  and $s1 < q$
 shows $((y, b, (e2 * b + q - (x * y + q - r) \ mod \ q) \ mod \ q, e2), r, (x * y + q - r) \ mod \ q) =$

((y, b, (e2 ∗ b + q − (x ∗ y + q − r) mod q) mod q, e2), (x ∗ y + q − (x ∗ y + q − r) mod q) mod q, (x ∗ y + q − r) mod q)
proof−
  have *s1* = (*x*∗*y* + *q* − ((*x*∗*y* + *q* − *s1*) *mod q*)) *mod q*
    using *assms secure-mult.s1 secure-mult-axioms* **by** *blast*
  then show *?thesis* **using** *assms s1* **by** *blast*
qed

theorem *P2-security*:
  assumes *x* < *q* *y* < *q*
  shows *sim-non-det-def.perfect-sec-P2 x y*
  including *monad-normalisation*
proof−
  have ((*funct x y*) ≫= (λ (*s1′*,*s2′*). (*sim-non-det-def.Ideal2 y x s2′*))) = *R2 x y*
  proof−
    have *R2 x y* = *do* {
      a :: *nat* ← *sample-uniform q*;
      b :: *nat* ← *sample-uniform q*;
      r :: *nat* ← *sample-uniform q*;
      let c1 = *a*;
      let d1 = *r*;
      let c2 = *b*;
      let d2 = ((*a*∗*b* + (*q* − *r*)) *mod q*);
      let e2 = (*x* + *c1*) *mod q*;
      let e1 = (*y* + (*q* − *c2*)) *mod q*;
      let s1 = (*x*∗*e1* + (*q* − *r*)) *mod q*;
      let s2 = (*e2* ∗ *c2* + (*q* − *d2*)) *mod q*;
      return-spmf ((*y*, *c2*, *d2*, *e2*), *s1*, *s2*)}
      by(*simp add*: *R2-def TI-def Let-def*)
    also have ... = *do* {
      a :: *nat* ← *sample-uniform q*;
      b :: *nat* ← *sample-uniform q*;
      r :: *nat* ← *sample-uniform q*;
      let c1 = *a*;
      let d1 = *r*;
      let c2 = *b*;
      let e2 = (*x* + *c1*) *mod q*;
      let d2 = ((((*e2* + *q* − *x*) *mod q*)∗*b* + (*q* − *r*)) *mod q*);
      let s1 = (*x*∗((*y* + (*q* − *c2*)) *mod q*) + (*q* − *r*)) *mod q*;
      return-spmf ((*y*, *c2*, *d2*, *e2*), (*s1*, (*x*∗*y* + (*q* − *s1*)) *mod q*))}
      by(*simp add*: *Let-def s1-s2-P2 assms c1-P2 cong*: *bind-spmf-cong-simp*)
    also have ... = *do* {
      b :: *nat* ← *sample-uniform q*;
      r :: *nat* ← *sample-uniform q*;
      let d1 = *r*;
      let c2 = *b*;
      e2 ← *map-spmf* (λ *c1*. (*x* + *c1*) *mod q*) (*sample-uniform q*);
      let d2 = ((((*e2* + *q* − *x*) *mod q*)∗*b* + (*q* − *r*)) *mod q*);
      let s1 = (*x*∗((*y* + (*q* − *c2*)) *mod q*) + (*q* − *r*)) *mod q*;

return-spmf ((*y*, *c2*, *d2*, *e2*), *s1*, (*x*∗*y* + (*q* − *s1*)) *mod q*)}
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
b :: *nat* ← *sample-uniform q*;
r :: *nat* ← *sample-uniform q*;
let d1 = *r*;
let c2 = *b*;
e2 ← *sample-uniform q*;
let d2 = (((*e2* + *q* − *x*) *mod q*)∗*b* + (*q* − *r*)) *mod q*;
let s1 = (*x*∗((*y* + (*q* − *c2*)) *mod q*) + (*q* − *r*)) *mod q*;
return-spmf ((*y*, *c2*, *d2*, *e2*), *s1*, (*x*∗*y* + (*q* − *s1*)) *mod q*)}
by(*simp add*: *samp-uni-plus-one-time-pad*)
also have ... = *do* {
b :: *nat* ← *sample-uniform q*;
r :: *nat* ← *sample-uniform q*;
e2 ← *sample-uniform q*;
let s1 = (*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − *r*)) *mod q*;
let s2 = (*x*∗*y* + (*q* − *s1*)) *mod q*;
let d2 = (((*e2* + *q* − *x*) *mod q*)∗*b* + (*q* − *r*)) *mod q*;
return-spmf ((*y*, *b*, *d2*, *e2*), *s1*, *s2*)}
by(*simp*)
also have ... = *do* {
b :: *nat* ← *sample-uniform q*;
r :: *nat* ← *sample-uniform q*;
e2 ← *sample-uniform q*;
let s1 = (*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − *r*)) *mod q*;
let s2 = (*x*∗*y* + (*q* − *s1*)) *mod q*;
let d2 = (*e2*∗*b* + (*q* − *s2*)) *mod q*;
return-spmf ((*y*, *b*, *d2*, *e2*), *s1*, *s2*)}
by(*simp add*: *d2-P2 assms Let-def cong*: *bind-spmf-cong-simp*)
also have ... = *do* {
b :: *nat* ← *sample-uniform q*;
e2 ← *sample-uniform q*;
s1 ← *map-spmf* (λ *r*. (*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − *r*)) *mod q*) (*sample-uniform q*);
let s2 = (*x*∗*y* + (*q* − *s1*)) *mod q*;
let d2 = (*e2*∗*b* + (*q* − *s2*)) *mod q*;
return-spmf ((*y*, *b*, *d2*, *e2*), *s1*, *s2*)}
by(*simp add*: *bind-map-spmf o-def Let-def*)
also have ... = *do* {
b :: *nat* ← *sample-uniform q*;
e2 ← *sample-uniform q*;
s1 ← *sample-uniform q*;
let s2 = (*x*∗*y* + (*q* − *s1*)) *mod q*;
let d2 = (*e2*∗*b* + (*q* − *s2*)) *mod q*;
return-spmf ((*y*, *b*, *d2*, *e2*), *s1*, *s2*)}
by(*simp add*: *samp-uni-minus-one-time-pad*)
also have ... = *do* {
b :: *nat* ← *sample-uniform q*;

e2 ← *sample-uniform q*;
s1 ← *sample-uniform q*;
let s2 = $(x*y + (q - s1))$ *mod q*;
let d2 = $(e2*b + (q - s2))$ *mod q*;
return-spmf $((y, b, d2, e2), (x*y + (q - s2))$ *mod q*, $s2)\}$
by(*simp add*: *s1-P2 assms Let-def cong*: *bind-spmf-cong-simp*)
ultimately show *?thesis* **by**(*simp add*: *funct-def Let-def sim-non-det-def.Ideal2-def Out2-def S2-def R2-def*)
qed
then show *?thesis* **by**(*simp add*: *sim-non-det-def.perfect-sec-P2-def*)
qed

lemma *s1-s2-P1*: **assumes** $x < q$ $xa < q$ $xb < q$ $xc < q$ $y < q$
shows $((x, xa, xb, (y + q - xc)$ *mod q*$), (x * ((y + q - xc)$ *mod q*$) + q - xb)$ *mod q*, $((x + xa)$ *mod q* $* xc + q - (xa * xc + q - xb)$ *mod q*$)$ *mod q*$)$ =
$((x, xa, xb, (y + q - xc)$ *mod q*$), (x * ((y + q - xc)$ *mod q*$) + q - xb)$ *mod q*, $(x * y + q - (x * ((y + q - xc)$ *mod q*$) + q - xb)$ *mod q*$)$ *mod q*$)$
using *assms s1-s2* **by** *metis*

lemma *mod-minus*: **assumes** $a - b > 0$ **and** $c - d > 0$
shows $(a - b + (c - d$ *mod q*$))$ *mod q* = $(a - b + (c - d))$ *mod q*
using *assms*
by (*metis cong-def minus-mod mod-add-right-eq zero-less-diff*)

lemma *r*:
assumes *e1*: *e1* = $(y + (q - b))$ *mod q*
and *s1*: *s1* = $(x*((y + (q - b))$ *mod q*$) + (q - r))$ *mod q*
and *b*: $b < q$
and *x*: $x < q$
and *y*: $y < q$
and *r*: $r < q$
shows $r = (x*e1 + (q - s1))$ *mod q*
(**is** *?lhs* = *?rhs*)
proof−
have *s1* = $(x*((y + (q - b))) + (q - r))$ *mod q* **using** *s1 b*
by (*metis mod-add-left-eq mod-mult-right-eq*)
then have *s1-dist*: *s1* = $(x*y + x*(q - b) + (q - r))$ *mod q*
by(*metis distrib-left*)
then have *?rhs* = $(x*((y + (q - b))$ *mod q*$) + (q - (x*y + x*(q - b) + (q - r))$ *mod q*$))$ *mod q*
using *e1* **by** *simp*
then have *?rhs* = $(x*((y + (q - b))) + (q - (x*y + x*(q - b) + (q - r))$ *mod q*$))$ *mod q*
by (*metis mod-add-left-eq mod-mult-right-eq*)
then have *?rhs* = $(x*y + x*(q - b) + (q - (x*y + x*(q - b) + (q - r))$ *mod q*$))$ *mod q*
by(*metis distrib-left*)
then have *a*: *?rhs* = $(x*y + x*q - x*b + (q - (x*y + x*(q - b) + (q - r))$ *mod q*$))$ *mod q*

using *distrib-left b diff-mult-distrib2* **by** *auto*
then have *b*: *?rhs* = (*x*∗*y* + *x*∗*q* − *x*∗*b* + (*q*∗*q* + *q*∗*q* + *q* − (*x*∗*y* + *x*∗(*q* − *b*) + (*q* − *r*)) *mod q*)) *mod q*
proof −
have (*x*∗*y* + *x*∗*q* − *x*∗*b* + (*q* − (*x*∗*y* + *x*∗(*q* − *b*) + (*q* − *r*)) *mod q*)) *mod q* = (*x*∗*y* + *x*∗*q* − *x*∗*b* + (*q*∗*q* + *q*∗*q* + *q* − (*x*∗*y* + *x*∗(*q* − *b*) + (*q* − *r*)) *mod q*)) *mod q*
proof −
have *f1*: ∀ *n na nb nc nd*. (*n*::*nat*) *mod na* ≠ *nb mod na* ∨ *nc mod na* ≠ *nd mod na* ∨ (*n* + *nc*) *mod na* = (*nb* + *nd*) *mod na*
by (*meson mod-add-cong*)
then have (*q* − (*x* ∗ *y* + *x*∗(*q* − *b*) + (*q* − *r*)) *mod q*) *mod q* = (*q* ∗ *q* + *q* ∗ *q* + *q* − (*x* ∗ *y* + *x*∗(*q* − *b*) + (*q* − *r*)) *mod q*) *mod q*
by (*metis Nat.diff-add-assoc mod-le-divisor q-gt-0 mod-mult-self4*)
then show *?thesis*
using *f1* **by** *blast*
qed
then show *?thesis* **using** *a* **by** *simp*
qed
then have *?rhs* = (*x*∗*y* + *x*∗*q* − *x*∗*b* + (*q*∗*q* + *q*∗*q* + *q* − (*x*∗*y* + *x*∗(*q* − *b*) + (*q* − *r*)))) *mod q*
proof−
have (*x*∗*y* + *x*∗*q* − *x*∗*b* + (*q*∗*q* + *q*∗*q* + *q* − (*x*∗*y* + *x*∗(*q* − *b*) + (*q* − *r*)) *mod q*)) *mod q* =
(*x*∗*y* + *x*∗*q* − *x*∗*b* + (*q*∗*q* + *q*∗*q* + *q* − (*x*∗*y* + *x*∗(*q* − *b*)+ (*q* − *r*)))) *mod q*
proof(*cases x* = *0*)
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *assms*(*2*) *assms*(*4*) *True Nat.add-diff-assoc add.left-neutral*
cong-def diff-le-self minus-mod mult-is-0 not-gr-zero zero-eq-add-iff-both-eq-0 zero-less-diff)
next
case *False*
have *qb*: *q* − *b* ≤ *q*
using *b* **by** *simp*
then have *qb*′:*x*∗(*q* − *b*) < *q*∗*q*
using *x* **by** (*metis mult-less-le-imp-less nat-0-less-mult-iff nat-less-le neq0-conv*)

have *a*: *x*∗*y* + *x*∗(*q* − *b*) > *0*
using *False assms* **by** *simp*
have *1*: *q*∗*q* > *x*∗*y*
using *False* **by** (*simp add*: *mult-strict-mono q-gt-0 x y*)
have *2*: *q*∗*q* > *x*∗*q* **using** *False*
by (*simp add*: *mult-strict-mono q-gt-0 x y*)
have *b*: (*q*∗*q* + *q*∗*q* + *q* − (*x*∗*y* + *x*∗(*q* − *b*) + (*q* − *r*))) > *0*
using *1 qb*′ **by** *simp*
then show *?thesis* **using** *a b mod-minus*[*of x*∗*y* + *x*∗*q x*∗*b q*∗*q* + *q*∗*q* + *q*

```
x∗y + x∗(q − b) + (q − r)]
      by (smt add.left-neutral cong-def gr0I minus-mod zero-less-diff)
    qed
    then show ?thesis using b by simp
  qed
  then have d: ?rhs = (x∗y + x∗q − x∗b + (q∗q + q∗q + q − x∗y − x∗(q − b)
− (q − r))) mod q
    by simp
  then have e: ?rhs = (x∗y + x∗q − x∗b + q∗q + q∗q + q − x∗y − x∗(q − b)
− (q − r)) mod q
  proof(cases x = 0)
    case True
    then show ?thesis using True d by simp
  next
    case False
    have qb: q − b ≤ q using b by simp
    then have qb':x∗(q − b) < q∗q
     using x by (metis mult-less-le-imp-less nat-0-less-mult-iff nat-less-le neq0-conv)

    have a: x∗y + x∗(q − b) > 0 using False assms by simp
    have 1: q∗q > x∗y using False
      by (simp add: mult-strict-mono q-gt-0 x y)
    have 2: q∗q > x∗q using False
      by (simp add: mult-strict-mono q-gt-0 x y)
    have b: q∗q + q∗q + q  − x∗y − x∗(q − b) − (q − r) > 0 using 1 qb' by
simp
    then show ?thesis using b d
     by (smt Nat.add-diff-assoc add.assoc diff-diff-left less-imp-le-nat zero-less-diff)
  qed
  then have ?rhs = (x∗q − x∗b + q∗q + q∗q + q − x∗(q − b) − (q − r)) mod q
  proof−
   have (x∗y + x∗q − x∗b + q∗q + q∗q + q − x∗y − x∗(q − b) − (q − r)) mod
q = (x∗q − x∗b + q∗q + q∗q + q − x∗(q − b) − (q − r)) mod q
    proof −
      have 1: q + n − b = q − b + n for n
        by (simp add: assms(3) less-imp-le)
      have 2: (c::nat) ∗ b + (c ∗ a + n) = c ∗ (b + a) + n
        for n a b c by (simp add: distrib-left)
      have (c::nat) + (b + a) − (n + a) = c + b − n for n a b c
        by auto
      then have (q + (q ∗ q + (q ∗ q + x ∗ (q + y − b))) − (q − r + x ∗ (y +
(q − b)))) mod q = (q + (q ∗ q + (q ∗ q + x ∗ (q − b))) − (q − r + x ∗ (q −
b))) mod q
        by (metis (no-types) add.commute 1 2)
      then show ?thesis
        by (simp add: add.commute diff-mult-distrib2 distrib-left)
    qed
    then show ?thesis using e by simp
  qed
```

then have *?rhs* = (*x*∗(*q* − *b*) + *q*∗*q* + *q*∗*q* + *q* − *x*∗(*q* − *b*) − (*q* − *r*)) *mod q*
by(*metis diff-mult-distrib2*)
then have *?rhs* = (*q*∗*q* + *q*∗*q* + *q* − (*q* − *r*)) *mod q*
using *assms*(*6*) **by** *simp*
then have *?rhs* = (*q*∗*q* + *q*∗*q* + *q* − *q* + *r*) *mod q*
using *assms*(*6*) **by**(*simp add*: *Nat.diff-add-assoc2 less-imp-le*)
then have *?rhs* = (*q*∗*q* + *q*∗*q* + *r*) *mod q*
by *simp*
then have *?rhs* = *r mod q*
by (*metis add.commute distrib-right mod-mult-self1*)
then show *?thesis* **using** *assms*(*6*) **by** *simp*
qed

lemma *r-P2*:
assumes *b*: *b* < *q* **and** *x*: *x* < *q* **and** *y*: *y* < *q* **and** *r*: *r* < *q*
shows
((*x*, *a*, *r*, (*y* + *q* − *b*) *mod q*), (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*, (*x* ∗ *y* + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*) =
((*x*, *a*, (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*, (*y* + *q* − *b*) *mod q*), (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*,
(*x* ∗ *y* + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q*)
proof−
have (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − (*x* ∗ ((*y* + *q* − *b*) *mod q*) + *q* − *r*) *mod q*) *mod q* = *r*
(**is** *?lhs* = *?rhs*)
proof−
have *1*:*r* = (*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − ((*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − *r*)) *mod q*))) *mod q*
using *assms secure-mult.r secure-mult-axioms* **by** *blast*
also have *?rhs* = (*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − ((*x*∗((*y* + (*q* − *b*)) *mod q*) + (*q* − *r*)) *mod q*))) *mod q* **using** *assms 1* **by** *blast*
ultimately show *?thesis* **using** *assms 1* **by** *simp*
qed
then show *?thesis* **using** *assms* **by** *simp*
qed

theorem *P1-security*:
assumes *x* < *q* *y* < *q*
shows *sim-non-det-def.perfect-sec-P1 x y*
including *monad-normalisation*
proof−
have (*funct x y*) ≫= (λ (*s1′*,*s2′*). (*sim-non-det-def.Ideal1 x y s1′*)) = *R1 x y*
proof−
have *R1 x y* = *do* {
a :: *nat* ← *sample-uniform q*;
b :: *nat* ← *sample-uniform q*;
r :: *nat* ← *sample-uniform q*;
let c1 = *a*;

let d1 = r;
let c2 = b;
let d2 = ((a∗b + (q − r)) mod q);
let e2 = (x + c1) mod q;
let e1 = (y + (q − c2)) mod q;
let s1 = (x∗e1 + (q − d1)) mod q;
let s2 = (e2 ∗ c2 + (q − d2)) mod q;
return-spmf ((x, c1, d1, e1), s1, s2)}
by(*simp add*: *R1-def TI-def Let-def*)
also have ... = *do* {
a :: *nat* ← *sample-uniform q*;
b :: *nat* ← *sample-uniform q*;
r :: *nat* ← *sample-uniform q*;
let c1 = a;
let c2 = b;
let e1 = (y + (q − b)) mod q;
let s1 = (x∗((y + (q − b)) mod q) + (q − r)) mod q;
let d1 = (x∗e1 + (q − s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x∗y + (q − s1)) mod q)}
by(*simp add*: *Let-def assms s1-s2-P1 r-P2 cong*: *bind-spmf-cong-simp*)
also have ... = *do* {
a :: *nat* ← *sample-uniform q*;
b :: *nat* ← *sample-uniform q*;
let c1 = a;
let c2 = b;
let e1 = (y + (q − b)) mod q;
s1 ← map-spmf (λ r. (x∗((y + (q − b)) mod q) + (q − r)) mod q) (sample-uniform q);
let d1 = (x∗e1 + (q − s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x∗y + (q − s1)) mod q)}
by(*simp add*: *bind-map-spmf Let-def o-def*)
also have ... = *do* {
a :: *nat* ← *sample-uniform q*;
b :: *nat* ← *sample-uniform q*;
let c1 = a;
let c2 = b;
let e1 = (y + (q − b)) mod q;
s1 ← sample-uniform q;
let d1 = (x∗e1 + (q − s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x∗y + (q − s1)) mod q)}
by(*simp add*: *samp-uni-minus-one-time-pad*)
also have ... = *do* {
a :: *nat* ← *sample-uniform q*;
let c1 = a;
e1 ← map-spmf (λb. (y + (q − b)) mod q) (sample-uniform q);
s1 ← sample-uniform q;
let d1 = (x∗e1 + (q − s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x∗y + (q − s1)) mod q)}
by(*simp add*: *bind-map-spmf Let-def o-def*)

```
    also have ... = do {
    a :: nat ← sample-uniform q;
    let c1 = a;
    e1 ← sample-uniform q;
    s1 ← sample-uniform q;
    let d1 = (x*e1 + (q − s1)) mod q;
    return-spmf ((x, c1, d1, e1), s1, (x*y + (q − s1)) mod q)}
      by(simp add: samp-uni-minus-one-time-pad)
    ultimately show ?thesis by(simp add: funct-def sim-non-det-def.Ideal1-def
Let-def R1-def TI-def Out1-def S1-def)
  qed
  thus ?thesis by(simp add: sim-non-det-def.perfect-sec-P1-def)
qed

end

locale secure-mult-asymp =
  fixes q :: nat ⇒ nat
  assumes ⋀ n. secure-mult (q n)
begin

sublocale secure-mult q n for n
  using secure-mult-asymp-axioms secure-mult-asymp-def by blast

theorem P1-secure:
  assumes x < q n y < q n
  shows sim-non-det-def.perfect-sec-P1 n x y
  by (metis P1-security assms)

theorem P2-secure:
  assumes x < q n y < q n
  shows sim-non-det-def.perfect-sec-P2 n x y
  by (metis P2-security assms)

end

end
```

2.9 DHH Extension

We define a variant of the DDH assumption and show it is as hard as the original DDH assumption.

```
theory DH-Ext imports
  Game-Based-Crypto.Diffie-Hellman
  Cyclic-Group-Ext
begin

context ddh begin
```

definition *DDH0* :: *'grp adversary* ⇒ *bool spmf*
  where *DDH0* $\mathcal{A}$ = *do* {
    s ← *sample-uniform* (*order* $\mathcal{G}$);
    r ← *sample-uniform* (*order* $\mathcal{G}$);
    let h = **g** [^] *s*;
    $\mathcal{A}$ *h* (**g** [^] *r*) (*h* [^] *r*)}

definition *DDH1* :: *'grp adversary* ⇒ *bool spmf*
  where *DDH1* $\mathcal{A}$ = *do* {
    s ← *sample-uniform* (*order* $\mathcal{G}$);
    r ← *sample-uniform* (*order* $\mathcal{G}$);
    let h = **g** [^] *s*;
    $\mathcal{A}$ *h* (**g** [^] *r*) ((*h* [^] *r*) ⊗ **g**)}

definition *DDH-advantage* :: *'grp adversary* ⇒ *real*
  where *DDH-advantage* $\mathcal{A}$ = |*spmf* (*DDH0* $\mathcal{A}$) *True* − *spmf* (*DDH1* $\mathcal{A}$) *True*|

definition *DDH-*$\mathcal{A}'$ :: *'grp adversary* ⇒ *'grp* ⇒ *'grp* ⇒ *'grp* ⇒ *bool spmf*
  where *DDH-*$\mathcal{A}'$ *D-ddh a b c* = *D-ddh a b* (*c* ⊗ **g**)

end

locale *ddh-ext* = *ddh* + *cyclic-group* $\mathcal{G}$
begin

lemma *DDH0-eq-ddh-0*: *ddh.DDH0* $\mathcal{G}$ $\mathcal{A}$ = *ddh.ddh-0* $\mathcal{G}$ $\mathcal{A}$
  by(*simp add*: *ddh.DDH0-def Let-def monoid.nat-pow-pow ddh.ddh-0-def*)

lemma *DDH-bound1*: |*spmf* (*ddh.DDH0* $\mathcal{G}$ $\mathcal{A}$) *True* − *spmf* (*ddh.DDH1* $\mathcal{G}$ $\mathcal{A}$) *True*|
    ≤ |*spmf* (*ddh.ddh-0* $\mathcal{G}$ $\mathcal{A}$) *True* − *spmf* (*ddh.ddh-1* $\mathcal{G}$ $\mathcal{A}$) *True*|
      + |*spmf* (*ddh.ddh-1* $\mathcal{G}$ $\mathcal{A}$) *True* − *spmf* (*ddh.DDH1* $\mathcal{G}$ $\mathcal{A}$) *True*|
  by (*simp add*: *abs-diff-triangle-ineq2 DDH0-eq-ddh-0*)

lemma *DDH-bound2*:
  shows |*spmf* (*ddh.DDH0* $\mathcal{G}$ $\mathcal{A}$) *True* − *spmf* (*ddh.DDH1* $\mathcal{G}$ $\mathcal{A}$) *True*|
    ≤ *ddh.advantage* $\mathcal{G}$ $\mathcal{A}$ + |*spmf* (*ddh.ddh-1* $\mathcal{G}$ $\mathcal{A}$) *True* − *spmf* (*ddh.DDH1* $\mathcal{G}$ $\mathcal{A}$) *True*|
  using *advantage-def DDH-bound1* **by** *simp*

lemma *rewrite*:
  shows (*sample-uniform* (*order* $\mathcal{G}$) ⤜ (λ*x*. *sample-uniform* (*order* $\mathcal{G}$)
    ⤜ (λ*y*. *sample-uniform* (*order* $\mathcal{G}$) ⤜ (λ*z*. $\mathcal{A}$ (**g** [^] *x*) (**g** [^] *y*) (**g** [^] *z* ⊗ **g**)))))
      = (*sample-uniform* (*order* $\mathcal{G}$) ⤜ (λ*x*. *sample-uniform* (*order* $\mathcal{G}$)
        ⤜ (λ*y*. *sample-uniform* (*order* $\mathcal{G}$) ⤜ (λ*z*. $\mathcal{A}$ (**g** [^] *x*) (**g** [^] *y*) (**g** [^] *z*)))))
(**is** *?lhs* = *?rhs*)

proof−
  have *?lhs* = *do* {
   $x \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
   $y \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
   $c \leftarrow$ *map-spmf* (λ z. **g** [^] $z \otimes$ **g**) (*sample-uniform* (*order* $\mathcal{G}$));
   $\mathcal{A}$ (**g** [^] x) (**g** [^] y) c}
    by(*simp add*: *o-def bind-map-spmf Let-def*)
  also have ... = *do* {
   $x \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
   $y \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
   $c \leftarrow$ *map-spmf* (λx. **g** [^] x) (*sample-uniform* (*order* $\mathcal{G}$));
   $\mathcal{A}$ (**g** [^] x) (**g** [^] y) c}
    by(*simp add*: *sample-uniform-one-time-pad*)
  ultimately show *?thesis*
    by(*simp add*: *Let-def bind-map-spmf o-def*)
qed

lemma *DDH-$\mathcal{A}'$-bound*: *ddh.advantage* $\mathcal{G}$ (*ddh.DDH-$\mathcal{A}'$* $\mathcal{G}$ $\mathcal{A}$) = |*spmf* (*ddh.ddh-1* $\mathcal{G}$ $\mathcal{A}$) *True* − *spmf* (*ddh.DDH1* $\mathcal{G}$ $\mathcal{A}$) *True*|
 unfolding *ddh.advantage-def ddh.ddh-1-def ddh.DDH1-def Let-def ddh.DDH-$\mathcal{A}'$-def ddh.ddh-0-def*
  by (*simp add*: *rewrite abs-minus-commute nat-pow-pow*)

lemma *DDH-advantage-bound*: *ddh.DDH-advantage* $\mathcal{G}$ $\mathcal{A}$ $\leq$ *ddh.advantage* $\mathcal{G}$ $\mathcal{A}$ + *ddh.advantage* $\mathcal{G}$ (*ddh.DDH-$\mathcal{A}'$* $\mathcal{G}$ $\mathcal{A}$)
  using *DDH-bound2 DDH-$\mathcal{A}'$-bound DDH-advantage-def* **by** *simp*

end

end

3 Malicious Security

Here we define security in the malicious security setting. We follow the definitions given in [4] and [2]. The definition of malicious security follows the real/ideal world paradigm.

3.1 Malicious Security Definitions

theory *Malicious-Defs* **imports**
  CryptHOL.CryptHOL
begin

type-synonym (*'in1'*,*'aux'*, *'P1-S1-aux'*) *P1-ideal-adv1* = *'in1'* ⇒ *'aux'* ⇒ (*'in1'* × *'P1-S1-aux'*) *spmf*

type-synonym (*'in1'*, *'aux'*, *'out1'*, *'P1-S1-aux'*, *'adv-out1'*) *P1-ideal-adv2* = *'in1'* ⇒ *'aux'* ⇒ *'out1'* ⇒ *'P1-S1-aux'* ⇒ *'adv-out1'* *spmf*

type-synonym (*'in1'*, *'aux'*, *'out1'*, *'P1-S1-aux'*, *'adv-out1'*) *P1-ideal-adv* = (*'in1'*,*'aux'*, *'P1-S1-aux'*) *P1-ideal-adv1* × (*'in1'*, *'aux'*, *'out1'*, *'P1-S1-aux'*, *'adv-out1'*) *P1-ideal-adv2*

type-synonym (*'P1-real-adv'*, *'in1'*, *'aux'*, *'P1-S1-aux'*) *P1-sim1* = *'P1-real-adv'* ⇒ *'in1'* ⇒ *'aux'* ⇒ (*'in1'* × *'P1-S1-aux'*) *spmf*

type-synonym (*'P1-real-adv'*, *'in1'*, *'aux'*, *'out1'*, *'P1-S1-aux'*, *'adv-out1'*) *P1-sim2*

= *'P1-real-adv'* ⇒ *'in1'* ⇒ *'aux'* ⇒ *'out1'*
⇒ *'P1-S1-aux'* ⇒ *'adv-out1'* *spmf*

type-synonym (*'P1-real-adv'*, *'in1'*, *'aux'*, *'out1'*, *'P1-S1-aux'*, *'adv-out1'*) *P1-sim*

= ((*'P1-real-adv'*, *'in1'*, *'aux'*, *'P1-S1-aux'*) *P1-sim1*
× (*'P1-real-adv'*, *'in1'*, *'aux'*, *'out1'*, *'P1-S1-aux'*, *'adv-out1'*) *P1-sim2*)

type-synonym (*'in2'*,*'aux'*, *'P2-S2-aux'*) *P2-ideal-adv1* = *'in2'* ⇒ *'aux'* ⇒ (*'in2'* × *'P2-S2-aux'*) *spmf*

type-synonym (*'in2'*, *'aux'*, *'out2'*, *'P2-S2-aux'*, *'adv-out2'*) *P2-ideal-adv2*
= *'in2'* ⇒ *'aux'* ⇒ *'out2'* ⇒ *'P2-S2-aux'* ⇒ *'adv-out2'* *spmf*

type-synonym (*'in2'*, *'aux'*, *'out2'*, *'P2-S2-aux'*, *'adv-out2'*) *P2-ideal-adv*
= (*'in2'*,*'aux'*, *'P2-S2-aux'*) *P2-ideal-adv1*
× (*'in2'*, *'aux'*, *'out2'*, *'P2-S2-aux'*, *'adv-out2'*) *P2-ideal-adv2*

type-synonym (*'P2-real-adv'*, *'in2'*, *'aux'*, *'P2-S2-aux'*) *P2-sim1* = *'P2-real-adv'* ⇒ *'in2'* ⇒ *'aux'* ⇒ (*'in2'* × *'P2-S2-aux'*) *spmf*

type-synonym (*'P2-real-adv'*, *'in2'*, *'aux'*, *'out2'*, *'P2-S2-aux'*, *'adv-out2'*) *P2-sim2*

= *'P2-real-adv'* ⇒ *'in2'* ⇒ *'aux'* ⇒ *'out2'*
⇒ *'P2-S2-aux'* ⇒ *'adv-out2'* *spmf*

type-synonym (*'P2-real-adv'*, *'in2'*, *'aux'*, *'out2'*, *'P2-S2-aux'*, *'adv-out2'*) *P2-sim*

= ((*'P2-real-adv'*, *'in2'*, *'aux'*, *'P2-S2-aux'*) *P2-sim1*
× (*'P2-real-adv'*, *'in2'*, *'aux'*, *'out2'*, *'P2-S2-aux'*, *'adv-out2'*) *P2-sim2*)

locale *malicious-base* =
fixes *funct* :: *'in1* ⇒ *'in2* ⇒ (*'out1* × *'out2*) *spmf* — the functionality
and *protocol* :: *'in1* ⇒ *'in2* ⇒ (*'out1* × *'out2*) *spmf* — outputs the output of each party in the protocol
and *S1-1* :: (*'P1-real-adv*, *'in1*, *'aux*, *'P1-S1-aux*) *P1-sim1* — first part of the simulator for party 1
and *S1-2* :: (*'P1-real-adv*, *'in1*, *'aux*, *'out1*, *'P1-S1-aux*, *'adv-out1*) *P1-sim2* —

second part of the simulator for party 1

and *P1-real-view* :: *'in1* ⇒ *'in2* ⇒ *'aux* ⇒ *'P1-real-adv* ⇒ (*'adv-out1* × *'out2*) *spmf* — real view for party 1, the adversary totally controls party 1

and *S2-1* :: (*'P2-real-adv*, *'in2*, *'aux*, *'P2-S2-aux*) *P2-sim1* — first part of the simulator for party 2

and *S2-2* :: (*'P2-real-adv*, *'in2*, *'aux*, *'out2*, *'P2-S2-aux*, *'adv-out2*) *P2-sim2* — second part of the simulator for party 1

and *P2-real-view* :: *'in1* ⇒ *'in2* ⇒ *'aux* ⇒ *'P2-real-adv* ⇒ (*'out1* × *'adv-out2*) *spmf* — real view for party 2, the adversary totally controls party 2

begin

definition *correct m1 m2* ⟷ (*protocol m1 m2* = *funct m1 m2*)

abbreviation *trusted-party x y* ≡ *funct x y*

The ideal game defines the ideal world. First we consider the case where party 1 is corrupt, and thus controlled by the adversary. The adversary is split into two parts, the first part takes the original input and auxillary information and outputs its input to the protocol. The trusted party then computes the functionality on the input given by the adversary and the correct input for party 2. Party 2 outputs the its correct output as given by the trusted party, the adversary provides the output for party 1.

definition *ideal-game-1* :: *'in1* ⇒ *'in2* ⇒ *'aux* ⇒ (*'in1*, *'aux*, *'out1*, *'P1-S1-aux*, *'adv-out1*) *P1-ideal-adv* ⇒ (*'adv-out1* × *'out2*) *spmf*
  where *ideal-game-1 x y z A* = *do* {
    let (*A1*,*A2*) = *A*;
    (*x'*, *aux-out*) ← *A1 x z*;
    (*out1*, *out2*) ← *trusted-party x' y*;
    out1' :: *'adv-out1* ← *A2 x' z out1 aux-out*;
    return-spmf (*out1'*, *out2*)}

definition *ideal-view-1* :: *'in1* ⇒ *'in2* ⇒ *'aux* ⇒ (*'P1-real-adv*, *'in1*, *'aux*, *'out1*, *'P1-S1-aux*, *'adv-out1*) *P1-sim* ⇒ *'P1-real-adv* ⇒ (*'adv-out1* × *'out2*) *spmf*
  where *ideal-view-1 x y z S* $\mathcal{A}$ = (*let* (*S1*, *S2*) = *S in* (*ideal-game-1 x y z* (*S1* $\mathcal{A}$, *S2* $\mathcal{A}$)))

We have information theoretic security when the real and ideal views are equal.

definition *perfect-sec-P1 x y z S* $\mathcal{A}$ ⟷ (*ideal-view-1 x y z S* $\mathcal{A}$ = *P1-real-view x y z* $\mathcal{A}$)

The advantage of party 1 denotes the probability of a distinguisher distinguishing the real and ideal views.

definition *adv-P1 x y z S* $\mathcal{A}$ (*D* :: (*'adv-out1* × *'out2*) ⇒ *bool spmf*) =
    |*spmf* (*P1-real-view x y z* $\mathcal{A}$ ⤜ (λ *view*. *D view*)) *True*
      − *spmf* (*ideal-view-1 x y z S* $\mathcal{A}$ ⤜ (λ *view*. *D view*)) *True* |

definition *ideal-game-2* :: *'in1* ⇒ *'in2* ⇒ *'aux* ⇒ (*'in2*, *'aux*, *'out2*, *'P2-S2-aux*, *'adv-out2*) *P2-ideal-adv* ⇒ (*'out1* × *'adv-out2*) *spmf*
  where *ideal-game-2 x y z A* = *do* {
    let (*A1*,*A2*) = *A*;
    (*y'*, *aux-out*) ← *A1 y z*;
    (*out1*, *out2*) ← *trusted-party x y'*;
    out2' :: *'adv-out2* ← *A2 y' z out2 aux-out*;
    return-spmf (*out1*, *out2'*)}

definition *ideal-view-2* :: *'in1* ⇒ *'in2* ⇒ *'aux* ⇒ (*'P2-real-adv*, *'in2*, *'aux*, *'out2*, *'P2-S2-aux*, *'adv-out2*) *P2-sim* ⇒ *'P2-real-adv* ⇒ (*'out1* × *'adv-out2*) *spmf*
  where *ideal-view-2 x y z S* $\mathcal{A}$ = (*let* (*S1*, *S2*) = *S in* (*ideal-game-2 x y z* (*S1* $\mathcal{A}$, *S2* $\mathcal{A}$)))

definition *perfect-sec-P2 x y z S* $\mathcal{A}$ ⟷ (*ideal-view-2 x y z S* $\mathcal{A}$ = *P2-real-view x y z* $\mathcal{A}$)

definition *adv-P2 x y z S* $\mathcal{A}$ (*D* :: (*'out1* × *'adv-out2*) ⇒ *bool spmf*) =
  |*spmf* (*P2-real-view x y z* $\mathcal{A}$ ⤜ (λ *view*. *D view*)) *True*
    − *spmf* (*ideal-view-2 x y z S* $\mathcal{A}$ ⤜ (λ *view*. *D view*)) *True* |

end

end

3.2 Malicious OT

Here we prove secure the 1-out-of-2 OT protocol given in [4] (p190). For party 1 reduce security to the DDH assumption and for party 2 we show information theoretic security.

theory *Malicious-OT* **imports**
  HOL−Number-Theory.Cong
  Cyclic-Group-Ext
  DH-Ext
  Malicious-Defs
  Number-Theory-Aux
  OT-Functionalities
  Uniform-Sampling
begin

type-synonym (*'aux*, *'grp'*, *'state*) *adv-1-P1* = (*'grp'* × *'grp'*) ⇒ *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *'aux* ⇒ ((*'grp'* × *'grp'* × *'grp'*) × *'state*) *spmf*

type-synonym (*'grp'*, *'state*) *adv-2-P1* = *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ (*'grp'* × *'grp'*) ⇒ *'state* ⇒ (((*'grp'* × *'grp'*) × (*'grp'* × *'grp'*)) × *'state*) *spmf*

type-synonym (*'adv-out1*,*'state*) *adv-3-P1* = *'state* ⇒ *'adv-out1 spmf*

type-synonym ('*aux*, '*grp*', '*adv-out1*, '*state*) *adv-mal-P1* = (('*aux*, '*grp*', '*state*) *adv-1-P1* × ('*grp*', '*state*) *adv-2-P1* × ('*adv-out1*,'*state*) *adv-3-P1*)

type-synonym ('*aux*, '*grp*','*state*) *adv-1-P2* = *bool* ⇒ '*aux* ⇒ (('*grp*' × '*grp*' × '*grp*' × '*grp*' × '*grp*') × '*state*) *spmf*

type-synonym ('*grp*','*state*) *adv-2-P2* = ('*grp*' × '*grp*' × '*grp*' × '*grp*' × '*grp*') ⇒ '*state* ⇒ ((('*grp*' × '*grp*' × '*grp*') × *nat*) × '*state*) *spmf*

type-synonym ('*grp*', '*adv-out2*, '*state*) *adv-3-P2* = ('*grp*' × '*grp*') ⇒ ('*grp*' × '*grp*') ⇒ '*state* ⇒ '*adv-out2* *spmf*

type-synonym ('*aux*, '*grp*', '*adv-out2*, '*state*) *adv-mal-P2* = (('*aux*, '*grp*','*state*) *adv-1-P2* × ('*grp*','*state*) *adv-2-P2* × ('*grp*', '*adv-out2*,'*state*) *adv-3-P2*)

locale *ot-base* =
  fixes $\mathcal{G}$ :: '*grp cyclic-group* (**structure**)
  assumes *finite-group*: *finite* (*carrier* $\mathcal{G}$)
    and *order-gt-0*: *order* $\mathcal{G}$ > *0*
    and *prime-order*: *prime* (*order* $\mathcal{G}$)
begin

lemma *prime-field*: *a* < (*order* $\mathcal{G}$) ⟹ *a* ≠ *0* ⟹ *coprime a* (*order* $\mathcal{G}$)
 by (*metis dvd-imp-le neq0-conv not-le prime-imp-coprime prime-order coprime-commute*)

The protocol uses a call to an idealised functionality of a zero knowledge protocol for the DDH relation, this is described by the functionality given below.

fun *funct-DH-ZK* :: ('*grp* × '*grp* × '*grp*) ⇒ (('*grp* × '*grp* × '*grp*) × *nat*) ⇒ (*bool* × *unit*) *spmf*
  where *funct-DH-ZK* (*h*,*a*,*b*) ((*h*',*a*',*b*'),*r*) = *return-spmf* (*a* = **g** [^] *r* ∧ *b* = *h* [^] *r* ∧ (*h*,*a*,*b*) = (*h*',*a*',*b*'), ())

The probabilistic program that defines the output for both parties in the protocol.

definition *protocol-ot* :: ('*grp* × '*grp*) ⇒ *bool* ⇒ (*unit* × '*grp*) *spmf*
  where *protocol-ot M σ* = *do* {
  let (*x0*,*x1*) = *M*;
  r ← *sample-uniform* (*order* $\mathcal{G}$);
  α0 ← *sample-uniform* (*order* $\mathcal{G}$);
  α1 ← *sample-uniform* (*order* $\mathcal{G}$);
  let h0 = **g** [^] *α0*;
  let h1 = **g** [^] *α1*;
  let a = **g** [^] *r*;
  let b0 = *h0* [^] *r* ⊗ **g** [^] (*if σ then* (*1*::*nat*) *else 0*);
  let b1 = *h1* [^] *r* ⊗ **g** [^] (*if σ then* (*1*::*nat*) *else 0*);
  let h = *h0* ⊗ *inv h1*;
  let b = *b0* ⊗ *inv b1*;
  - :: *unit* ← *assert-spmf* (*a* = **g** [^] *r* ∧ *b* = *h* [^] *r*);

$u0 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$u1 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$v0 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$v1 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
let $z0 = b0$ [^] $u0 \otimes h0$ [^] $v0 \otimes x0$;
let $w0 = a$ [^] $u0 \otimes \mathbf{g}$ [^] $v0$;
let $e0 = (w0, z0)$;
let $z1 = (b1 \otimes inv\ \mathbf{g})$ [^] $u1 \otimes h1$ [^] $v1 \otimes x1$;
let $w1 = a$ [^] $u1 \otimes \mathbf{g}$ [^] $v1$;
let $e1 = (w1, z1)$;
return-spmf ((), (*if* σ *then* ($z1 \otimes inv$ ($w1$ [^] $\alpha 1$)) *else* ($z0 \otimes inv$ ($w0$ [^] $\alpha 0$))))}

Party 1 sends three messages (including the output) in the protocol so we split the adversary into three parts, one part to output each message. The real view of the protocol for party 1 outputs the correct output for party 2 and the adversary outputs the output for party 1.

definition *P1-real-model* :: ('*grp* × '*grp*) ⇒ *bool* ⇒ '*aux* ⇒ ('*aux*, '*grp*, '*adv-out1*, '*state*) *adv-mal-P1* ⇒ ('*adv-out1* × '*grp*) *spmf*
where *P1-real-model M* σ *z* $\mathcal{A}$ = *do* {
let ($\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$) = $\mathcal{A}$;
$r \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$\alpha 0 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$\alpha 1 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
let $h0 = \mathbf{g}$ [^] $\alpha 0$;
let $h1 = \mathbf{g}$ [^] $\alpha 1$;
let $a = \mathbf{g}$ [^] r;
let $b0 = h0$ [^] $r \otimes$ (*if* σ *then* $\mathbf{g}$ *else* $\mathbf{1}$);
let $b1 = h1$ [^] $r \otimes$ (*if* σ *then* $\mathbf{g}$ *else* $\mathbf{1}$);
((*in1* :: '*grp*, *in2* ::'*grp*, *in3* :: '*grp*), *s*) ← $\mathcal{A}1$ *M h0 h1 a b0 b1 z*;
let (*h*,*a*,*b*) = ($h0 \otimes inv\ h1$, *a*, $b0 \otimes inv\ b1$);
(*b* :: *bool*, - :: *unit*) ← *funct-DH-ZK* (*in1*, *in2*, *in3*) ((*h*,*a*,*b*), *r*);
- :: *unit* ← *assert-spmf* (*b*);
(((*w0*,*z0*),(*w1*,*z1*)), *s'*) ← $\mathcal{A}2$ *h0 h1 a b0 b1 M s*;
adv-out :: '*adv-out1* ← $\mathcal{A}3$ *s'*;
return-spmf (*adv-out*, (*if* σ *then* ($z1 \otimes$ (*inv w1* [^] $\alpha 1$)) *else* ($z0 \otimes$ (*inv w0* [^] $\alpha 0$))))}

The first and second part of the simulator for party 1 are defined below.

definition *P1-S1* :: ('*aux*, '*grp*, '*adv-out1*, '*state*) *adv-mal-P1* ⇒ ('*grp* × '*grp*) ⇒ '*aux* ⇒ (('*grp* × '*grp*) × '*state*) *spmf*
where *P1-S1* $\mathcal{A}$ *M z* = *do* {
let ($\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$) = $\mathcal{A}$;
$r \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$\alpha 0 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
$\alpha 1 \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
let $h0 = \mathbf{g}$ [^] $\alpha 0$;
let $h1 = \mathbf{g}$ [^] $\alpha 1$;
let $a = \mathbf{g}$ [^] r;
let $b0 = h0$ [^] r;

let b1 = h1 [^] r ⊗ **g**;
((in1 :: 'grp, in2 ::'grp, in3 :: 'grp), s) ← 𝒜1 M h0 h1 a b0 b1 z;
let (h,a,b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
- :: unit ← assert-spmf ((in1, in2, in3) = (h,a,b));
(((w0,z0),(w1,z1)),s') ← 𝒜2 h0 h1 a b0 b1 M s;
let x0 = (z0 ⊗ (inv w0 [^] α0));
let x1 = (z1 ⊗ (inv w1 [^] α1));
return-spmf ((x0,x1), s')}

definition *P1-S2 :: ('aux, 'grp, 'adv-out1,'state) adv-mal-P1 ⇒ ('grp × 'grp) ⇒ 'aux ⇒ unit ⇒ 'state ⇒ 'adv-out1 spmf*
where *P1-S2 𝒜 M z out1 s' = do {*
let (𝒜1, 𝒜2, 𝒜3) = 𝒜;
𝒜3 s'}

We explicitly provide the unfolded definition of the ideal model for convieience in the proof.

definition *P1-ideal-model :: ('grp × 'grp) ⇒ bool ⇒ 'aux ⇒ ('aux, 'grp, 'adv-out1,'state) adv-mal-P1 ⇒ ('adv-out1 × 'grp) spmf*
where *P1-ideal-model M σ z 𝒜 = do {*
let (𝒜1, 𝒜2, 𝒜3) = 𝒜;
r ← sample-uniform (order 𝒢);
α0 ← sample-uniform (order 𝒢);
α1 ← sample-uniform (order 𝒢);
let h0 = **g** *[^] α0;*
let h1 = **g** *[^] α1;*
let a = **g** *[^] r;*
let b0 = h0 [^] r;
let b1 = h1 [^] r ⊗ **g**;
((in1 :: 'grp, in2 ::'grp, in3 :: 'grp), s) ← 𝒜1 M h0 h1 a b0 b1 z;
let (h,a,b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
- :: unit ← assert-spmf ((in1, in2, in3) = (h,a,b));
(((w0,z0),(w1,z1)),s') ← 𝒜2 h0 h1 a b0 b1 M s;
let x0' = z0 ⊗ inv w0 [^] α0;
let x1' = z1 ⊗ inv w1 [^] α1;
(-, f-out2) ← funct-OT-12 (x0', x1') σ;
adv-out :: 'adv-out1 ← 𝒜3 s';
return-spmf (adv-out, f-out2)}

The advantage associated with the unfolded definition of the ideal view.

definition
P1-adv-real-ideal-model (D :: ('adv-out1 × 'grp) ⇒ bool spmf) M σ 𝒜 z
= |spmf ((P1-real-model M σ z 𝒜) ≫= (λ view. D view)) True
− spmf ((P1-ideal-model M σ z 𝒜) ≫= (λ view. D view)) True|

We now define the real view and simulators for party 2 in an analogous way.

definition *P2-real-model :: ('grp × 'grp) ⇒ bool ⇒ 'aux ⇒ ('aux, 'grp, 'adv-out2,'state) adv-mal-P2 ⇒ (unit × 'adv-out2) spmf*

where *P2-real-model M σ z 𝒜 = do* {
let (*x0*,*x1*) = *M*;
let (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
((*h0*,*h1*,*a*,*b0*,*b1*),*s*) ← *𝒜1 σ z*;
- :: *unit* ← *assert-spmf* (*h0* ∈ *carrier* 𝒢 ∧ *h1* ∈ *carrier* 𝒢 ∧ *a* ∈ *carrier* 𝒢 ∧ *b0* ∈ *carrier* 𝒢 ∧ *b1* ∈ *carrier* 𝒢);
(((*in1*, *in2*, *in3* :: ′*grp*), *r*),*s*′) ← *𝒜2* (*h0*,*h1*,*a*,*b0*,*b1*) *s*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
(*out-zk-funct*, *-*) ← *funct-DH-ZK* (*h*,*a*,*b*) ((*in1*, *in2*, *in3*), *r*);
- :: *unit* ← *assert-spmf out-zk-funct*;
u0 ← *sample-uniform* (*order* 𝒢);
u1 ← *sample-uniform* (*order* 𝒢);
v0 ← *sample-uniform* (*order* 𝒢);
v1 ← *sample-uniform* (*order* 𝒢);
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*;
let w0 = *a* [^] *u0* ⊗ **g** [^] *v0*;
let e0 = (*w0*, *z0*);
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let w1 = *a* [^] *u1* ⊗ **g** [^] *v1*;
let e1 = (*w1*, *z1*);
out ← *𝒜3 e0 e1 s*′;
return-spmf ((), *out*)}

definition *P2-S1* :: (′*aux*, ′*grp*, ′*adv-out2*,′*state*) *adv-mal-P2* ⇒ *bool* ⇒ ′*aux* ⇒ (*bool* × (′*grp* × ′*grp* × ′*grp* × ′*grp* × ′*grp*) × ′*state*) *spmf*
where *P2-S1 𝒜 σ z = do* {
let (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
((*h0*,*h1*,*a*,*b0*,*b1*),*s*) ← *𝒜1 σ z*;
- :: *unit* ← *assert-spmf* (*h0* ∈ *carrier* 𝒢 ∧ *h1* ∈ *carrier* 𝒢 ∧ *a* ∈ *carrier* 𝒢 ∧ *b0* ∈ *carrier* 𝒢 ∧ *b1* ∈ *carrier* 𝒢);
(((*in1*, *in2*, *in3* :: ′*grp*), *r*),*s*′) ← *𝒜2* (*h0*,*h1*,*a*,*b0*,*b1*) *s*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
(*out-zk-funct*, *-*) ← *funct-DH-ZK* (*h*,*a*,*b*) ((*in1*, *in2*, *in3*), *r*);
- :: *unit* ← *assert-spmf out-zk-funct*;
let l = *b0* ⊗ (*inv* (*h0* [^] *r*));
return-spmf ((*if l* = **1** *then False else True*), (*h0*,*h1*,*a*,*b0*,*b1*), *s*′)}

definition *P2-S2* :: (′*aux*, ′*grp*, ′*adv-out2*,′*state*) *adv-mal-P2* ⇒ *bool* ⇒ ′*aux* ⇒ ′*grp* ⇒ ((′*grp* × ′*grp* × ′*grp* × ′*grp* × ′*grp*) × ′*state*) ⇒ ′*adv-out2 spmf*
where *P2-S2 𝒜 σ*′ *z xσ aux-out = do* {
let (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
let ((*h0*,*h1*,*a*,*b0*,*b1*),*s*) = *aux-out*;
u0 ← *sample-uniform* (*order* 𝒢);
v0 ← *sample-uniform* (*order* 𝒢);
u1 ← *sample-uniform* (*order* 𝒢);
v1 ← *sample-uniform* (*order* 𝒢);
let w0 = *a* [^] *u0* ⊗ **g** [^] *v0*;
let w1 = *a* [^] *u1* ⊗ **g** [^] *v1*;
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ (*if σ*′ *then* **1** *else xσ*);

let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ (*if* σ′ *then* *x*σ *else* **1**);
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
𝒜*3 e0 e1 s*}

sublocale *mal-def* : *malicious-base funct-OT-12 protocol-ot P1-S1 P1-S2 P1-real-model P2-S1 P2-S2 P2-real-model* **.**

We prove the unfolded definition of the ideal views are equal to the definition we provide in the abstract locale that defines security.

lemma *P1-ideal-ideal-eq*:
shows *mal-def.ideal-view-1 x y z* (*P1-S1*, *P1-S2*) 𝒜 = *P1-ideal-model x y z* 𝒜
including *monad-normalisation*
unfolding *mal-def.ideal-view-1-def mal-def.ideal-game-1-def P1-ideal-model-def P1-S1-def P1-S2-def Let-def split-def*
by(*simp add*: *split-def*)

lemma *P1-advantages-eq*:
shows *mal-def.adv-P1 x y z* (*P1-S1*, *P1-S2*) 𝒜 *D* = *P1-adv-real-ideal-model D x y* 𝒜 *z*
by(*simp add*: *mal-def.adv-P1-def P1-adv-real-ideal-model-def P1-ideal-ideal-eq*)

fun *P1-DDH-mal-adv-σ-false* :: (′*grp* × ′*grp*) ⇒ ′*aux* ⇒ (′*aux*, ′*grp*, ′*adv-out1*,′*state*) *adv-mal-P1* ⇒ ((′*adv-out1* × ′*grp*) ⇒ *bool spmf*) ⇒ ′*grp ddh.adversary*
where *P1-DDH-mal-adv-σ-false M z* 𝒜 *D h a t* = *do* {
let (𝒜*1*, 𝒜*2*, 𝒜*3*) = 𝒜;
α*0* ← *sample-uniform* (*order* 𝒢);
let h0 = **g** [^] α*0*;
let h1 = *h*;
let b0 = *a* [^] α*0*;
let b1 = *t*;
((*in1* :: ′*grp*, *in2* ::′*grp*, *in3* :: ′*grp*),*s*) ← 𝒜*1 M h0 h1 a b0 b1 z*;
- :: *unit* ← *assert-spmf* (*in1* = *h0* ⊗ *inv h1* ∧ *in2* = *a* ∧ *in3* = *b0* ⊗ *inv b1*);
(((*w0*,*z0*),(*w1*,*z1*)),*s*′) ← 𝒜*2 h0 h1 a b0 b1 M s*;
let x0 = (*z0* ⊗ (*inv w0* [^] α*0*));
adv-out :: ′*adv-out1* ← 𝒜*3 s*′;
D (*adv-out*, *x0*)}

fun *P1-DDH-mal-adv-σ-true* :: (′*grp* × ′*grp*) ⇒ ′*aux* ⇒ (′*aux*, ′*grp*, ′*adv-out1*,′*state*) *adv-mal-P1* ⇒ ((′*adv-out1* × ′*grp*) ⇒ *bool spmf*) ⇒ ′*grp ddh.adversary*
where *P1-DDH-mal-adv-σ-true M z* 𝒜 *D h a t* = *do* {
let (𝒜*1*, 𝒜*2*, 𝒜*3*) = 𝒜;
α*1* :: *nat* ← *sample-uniform* (*order* 𝒢);
let h1 = **g** [^] α*1*;
let h0 = *h*;
let b0 = *t*;
let b1 = *a* [^] α*1* ⊗ **g**;
((*in1* :: ′*grp*, *in2* ::′*grp*, *in3* :: ′*grp*),*s*) ← 𝒜*1 M h0 h1 a b0 b1 z*;
- :: *unit* ← *assert-spmf* (*in1* = *h0* ⊗ *inv h1* ∧ *in2* = *a* ∧ *in3* = *b0* ⊗ *inv b1*);

$(((w0,z0),(w1,z1)),s') \leftarrow \mathcal{A}2\ h0\ h1\ a\ b0\ b1\ M\ s;$
$let\ x1 = (z1 \otimes (inv\ w1\ [\widehat{}]\ \alpha 1));$
$adv\text{-}out :: 'adv\text{-}out1 \leftarrow \mathcal{A}3\ s';$
$D\ (adv\text{-}out,\ x1)\}$

definition *P2-ideal-model* :: $('grp \times 'grp) \Rightarrow bool \Rightarrow 'aux \Rightarrow ('aux, 'grp, 'adv\text{-}out2, 'state)\ adv\text{-}mal\text{-}P2 \Rightarrow (unit \times 'adv\text{-}out2)\ spmf$
where *P2-ideal-model* $M\ \sigma\ z\ \mathcal{A} = do\ \{$
$let\ (x0,x1) = M;$
$let\ (\mathcal{A}1,\ \mathcal{A}2,\ \mathcal{A}3) = \mathcal{A};$
$((h0,h1,a,b0,b1),\ s) \leftarrow \mathcal{A}1\ \sigma\ z;$
$- :: unit \leftarrow assert\text{-}spmf\ (h0 \in carrier\ \mathcal{G} \wedge h1 \in carrier\ \mathcal{G} \wedge a \in carrier\ \mathcal{G} \wedge b0 \in carrier\ \mathcal{G} \wedge b1 \in carrier\ \mathcal{G});$
$(((in1,\ in2,\ in3),\ r),s') \leftarrow \mathcal{A}2\ (h0,h1,a,b0,b1)\ s;$
$let\ (h,a,b) = (h0 \otimes inv\ h1,\ a,\ b0 \otimes inv\ b1);$
$(out\text{-}zk\text{-}funct,\ -) \leftarrow funct\text{-}DH\text{-}ZK\ (h,a,b)\ ((in1,\ in2,\ in3),\ r);$
$- :: unit \leftarrow assert\text{-}spmf\ out\text{-}zk\text{-}funct;$
$let\ l = b0 \otimes (inv\ (h0\ [\widehat{}]\ r));$
$let\ \sigma' = (if\ l = \mathbf{1}\ then\ False\ else\ True);$
$(- :: unit,\ x\sigma) \leftarrow funct\text{-}OT\text{-}12\ (x0,\ x1)\ \sigma';$
$u0 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$v0 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$u1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$v1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$let\ w0 = a\ [\widehat{}]\ u0 \otimes \mathbf{g}\ [\widehat{}]\ v0;$
$let\ w1 = a\ [\widehat{}]\ u1 \otimes \mathbf{g}\ [\widehat{}]\ v1;$
$let\ z0 = b0\ [\widehat{}]\ u0 \otimes h0\ [\widehat{}]\ v0 \otimes (if\ \sigma'\ then\ \mathbf{1}\ else\ x\sigma);$
$let\ z1 = (b1 \otimes inv\ \mathbf{g})\ [\widehat{}]\ u1 \otimes h1\ [\widehat{}]\ v1 \otimes (if\ \sigma'\ then\ x\sigma\ else\ \mathbf{1});$
$let\ e0 = (w0,z0);$
$let\ e1 = (w1,z1);$
$out \leftarrow \mathcal{A}3\ e0\ e1\ s';$
$return\text{-}spmf\ ((),\ out)\}$

definition *P2-ideal-model-end* :: $('grp \times 'grp) \Rightarrow 'grp \Rightarrow (('grp \times 'grp \times 'grp \times 'grp \times 'grp) \times 'state)$
$\Rightarrow ('grp,\ 'adv\text{-}out2,\ 'state)\ adv\text{-}3\text{-}P2 \Rightarrow (unit \times 'adv\text{-}out2)\ spmf$
where *P2-ideal-model-end* $M\ l\ bs\ \mathcal{A}3 = do\ \{$
$let\ (x0,x1) = M;$
$let\ ((h0,h1,a,b0,b1),s) = bs;$
$let\ \sigma' = (if\ l = \mathbf{1}\ then\ False\ else\ True);$
$(-:: unit,\ x\sigma) \leftarrow funct\text{-}OT\text{-}12\ (x0,\ x1)\ \sigma';$
$u0 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$v0 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$u1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$v1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G});$
$let\ w0 = a\ [\widehat{}]\ u0 \otimes \mathbf{g}\ [\widehat{}]\ v0;$
$let\ w1 = a\ [\widehat{}]\ u1 \otimes \mathbf{g}\ [\widehat{}]\ v1;$
$let\ z0 = b0\ [\widehat{}]\ u0 \otimes h0\ [\widehat{}]\ v0 \otimes (if\ \sigma'\ then\ \mathbf{1}\ else\ x\sigma);$

let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ (*if σ′ then xσ else* **1**);
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← *𝒜3 e0 e1 s*;
return-spmf ((), *out*)}

definition *P2-ideal-model′* :: (*′grp* × *′grp*) ⇒ *bool* ⇒ *′aux* ⇒ (*′aux*, *′grp*, *′adv-out2*, *′state*) *adv-mal-P2* ⇒ (*unit* × *′adv-out2*) *spmf*
where *P2-ideal-model′ M σ z 𝒜* = *do* {
let (*x0*,*x1*) = *M*;
let (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
((*h0*,*h1*,*a*,*b0*,*b1*),*s*) ← *𝒜1 σ z*;
- :: *unit* ← *assert-spmf* (*h0* ∈ *carrier 𝒢* ∧ *h1* ∈ *carrier 𝒢* ∧ *a* ∈ *carrier 𝒢* ∧ *b0* ∈ *carrier 𝒢* ∧ *b1* ∈ *carrier 𝒢*);
(((*in1*, *in2*, *in3* :: *′grp*), *r*),*s′*) ← *𝒜2* (*h0*,*h1*,*a*,*b0*,*b1*) *s*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
(*out-zk-funct*, -) ← *funct-DH-ZK* (*h*,*a*,*b*) ((*in1*, *in2*, *in3*), *r*);
- :: *unit* ← *assert-spmf out-zk-funct*;
let l = *b0* ⊗ (*inv* (*h0* [^] *r*));
P2-ideal-model-end (*x0*,*x1*) *l* ((*h0*,*h1*,*a*,*b0*,*b1*),*s′*) *𝒜3*}

lemma *P2-ideal-model-rewrite*: *P2-ideal-model M σ z 𝒜* = *P2-ideal-model′ M σ z 𝒜*
by(*simp add*: *P2-ideal-model′-def P2-ideal-model-def P2-ideal-model-end-def Let-def split-def*)

definition *P2-real-model-end* :: (*′grp* × *′grp*) ⇒ ((*′grp* × *′grp* × *′grp* × *′grp* × *′grp*) × *′state*)
⇒ (*′grp*, *′adv-out2*,*′state*) *adv-3-P2* ⇒ (*unit* × *′adv-out2*) *spmf*
where *P2-real-model-end M bs 𝒜3* = *do* {
let (*x0*,*x1*) = *M*;
let ((*h0*,*h1*,*a*,*b0*,*b1*),*s*) = *bs*;
u0 ← *sample-uniform* (*order 𝒢*);
u1 ← *sample-uniform* (*order 𝒢*);
v0 ← *sample-uniform* (*order 𝒢*);
v1 ← *sample-uniform* (*order 𝒢*);
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*;
let w0 = *a* [^] *u0* ⊗ **g** [^] *v0*;
let e0 = (*w0*, *z0*);
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let w1 = *a* [^] *u1* ⊗ **g** [^] *v1*;
let e1 = (*w1*, *z1*);
out ← *𝒜3 e0 e1 s*;
return-spmf ((), *out*)}

definition *P2-real-model′* ::(*′grp* × *′grp*) ⇒ *bool* ⇒ *′aux* ⇒ (*′aux*, *′grp*, *′adv-out2*, *′state*) *adv-mal-P2* ⇒ (*unit* × *′adv-out2*) *spmf*
where *P2-real-model′ M σ z 𝒜* = *do* {

let $(x0,x1) = M$;
let $(\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A}$;
$((h0,h1,a,b0,b1),s) \leftarrow \mathcal{A}1\ \sigma\ z$;
- :: *unit* $\leftarrow$ *assert-spmf* ($h0 \in$ *carrier* $\mathcal{G} \wedge h1 \in$ *carrier* $\mathcal{G} \wedge a \in$ *carrier* $\mathcal{G} \wedge b0 \in$ *carrier* $\mathcal{G} \wedge b1 \in$ *carrier* $\mathcal{G}$);
$(((in1, in2, in3 :: 'grp), r),s') \leftarrow \mathcal{A}2\ (h0,h1,a,b0,b1)\ s$;
let $(h,a,b) = (h0 \otimes inv\ h1,\ a,\ b0 \otimes inv\ b1)$;
(*out-zk-funct*, *-*) $\leftarrow$ *funct-DH-ZK* (h,a,b) $((in1, in2, in3), r)$;
- :: *unit* $\leftarrow$ *assert-spmf out-zk-funct*;
P2-real-model-end M $((h0,h1,a,b0,b1),s')$ $\mathcal{A}3$}

lemma *P2-real-model-rewrite*: *P2-real-model M* σ *z* $\mathcal{A}$ = *P2-real-model′ M* σ *z* $\mathcal{A}$
by(*simp add*: *P2-real-model′-def P2-real-model-def P2-real-model-end-def split-def*)

lemma *P2-ideal-view-unfold*: *mal-def.ideal-view-2* $(x0,x1)$ σ *z* (*P2-S1*, *P2-S2*) $\mathcal{A}$ = *P2-ideal-model* $(x0,x1)$ σ *z* $\mathcal{A}$
unfolding *local.mal-def.ideal-view-2-def P2-ideal-model-def local.mal-def.ideal-game-2-def P2-S1-def P2-S2-def*
by(*auto simp add*: *Let-def split-def*)

end

locale *ot* = *ot-base* + *cyclic-group* $\mathcal{G}$
begin

lemma *P1-assert-correct1*:
shows $((\mathbf{g} [\widehat{\ }] (\alpha 0::nat)) [\widehat{\ }] (r::nat) \otimes \mathbf{g} \otimes inv\ ((\mathbf{g} [\widehat{\ }] (\alpha 1::nat)) [\widehat{\ }] r \otimes \mathbf{g})$
$= (\mathbf{g} [\widehat{\ }] \alpha 0 \otimes inv\ (\mathbf{g} [\widehat{\ }] \alpha 1)) [\widehat{\ }] r)$
(**is** *?lhs* = *?rhs*)
proof−
have *in-carrier1*: $(\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r \in$ *carrier* $\mathcal{G}$ **by** *simp*
have *in-carrier2*: *inv* $((\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r) \in$ *carrier* $\mathcal{G}$ **by** *simp*
have *1*:*?lhs* = $(\mathbf{g} [\widehat{\ }] \alpha 0) [\widehat{\ }] r \otimes \mathbf{g} \otimes inv\ ((\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r) \otimes inv\ \mathbf{g}$
using *cyclic-group-assoc nat-pow-pow inverse-split in-carrier1* **by** *simp*
also have *2*:... = $(\mathbf{g} [\widehat{\ }] \alpha 0) [\widehat{\ }] r \otimes (\mathbf{g} \otimes inv\ ((\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r)) \otimes inv\ \mathbf{g}$
using *cyclic-group-assoc in-carrier2* **by** *simp*
also have ... = $(\mathbf{g} [\widehat{\ }] \alpha 0) [\widehat{\ }] r \otimes (inv\ ((\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r) \otimes \mathbf{g}) \otimes inv\ \mathbf{g}$
using *in-carrier2 cyclic-group-commute* **by** *simp*
also have *3*: ... = $(\mathbf{g} [\widehat{\ }] \alpha 0) [\widehat{\ }] r \otimes inv\ ((\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r) \otimes (\mathbf{g} \otimes inv\ \mathbf{g})$
using *cyclic-group-assoc in-carrier2* **by** *simp*
also have ... = $(\mathbf{g} [\widehat{\ }] \alpha 0) [\widehat{\ }] r \otimes inv\ ((\mathbf{g} [\widehat{\ }] \alpha 1) [\widehat{\ }] r)$ **by** *simp*
also have ... = $(\mathbf{g} [\widehat{\ }] \alpha 0) [\widehat{\ }] r \otimes inv\ ((\mathbf{g} [\widehat{\ }] \alpha 1)) [\widehat{\ }] r$
using *inverse-pow-pow* **by** *simp*
ultimately show *?thesis*
by (*simp add*: *cyclic-group-commute pow-mult-distrib*)
qed

lemma *P1-assert-correct2*:
shows $(\mathbf{g} [\widehat{\ }] (\alpha 0::nat)) [\widehat{\ }] (r::nat) \otimes inv\ ((\mathbf{g} [\widehat{\ }] (\alpha 1::nat)) [\widehat{\ }] r) = (\mathbf{g} [\widehat{\ }] \alpha 0$

⊗ *inv* (**g** [^] *α1*)) [^] *r*
  (**is** *?lhs* = *?rhs*)
proof−
 have *in-carrier2*:**g** [^] *α1* ∈ *carrier* $\mathcal{G}$ **by** *simp*
 hence *?lhs* = (**g** [^] *α0*) [^] *r* ⊗ *inv* ((**g** [^] *α1*)) [^] *r*
  using *inverse-pow-pow* **by** *simp*
 thus *?thesis*
  by (*simp add*: *cyclic-group-commute monoid-comm-monoidI pow-mult-distrib*)
qed

sublocale *ddh*: *ddh-ext*
 by (*simp add*: *cyclic-group-axioms ddh-ext.intro*)

lemma *P1-real-ddh0-σ-false*:
 assumes *σ* = *False*
 shows ((*P1-real-model M σ z* $\mathcal{A}$) ⤜ (*λ view. D view*)) = (*ddh.DDH0* (*P1-DDH-mal-adv-σ-false M z* $\mathcal{A}$ *D*))
 including *monad-normalisation*
proof−
 have
  (*in2* = **g** [^] (*r*::*nat*) ∧ *in3* = *in1* [^] *r* ∧ *in1* = **g** [^] (*α0*::*nat*) ⊗ *inv* (**g** [^] (*α1*::*nat*))
   ∧ *in2* = **g** [^] *r* ∧ *in3* = (**g** [^] *r*) [^] *α0* ⊗ *inv* ((**g** [^] *α1*) [^] *r*))
    ⟷ (*in1* = **g** [^] *α0* ⊗ *inv* (**g** [^] *α1*) ∧ *in2* = **g** [^] *r* ∧ *in3*
     = (**g** [^] *r*) [^] *α0* ⊗ *inv* ((**g** [^] *α1*) [^] *r*))
  for *in1 in2 in3 r α0 α1*
  by (*auto simp add*: *P1-assert-correct2 power-swap*)
 moreover have ((*P1-real-model M σ z* $\mathcal{A}$) ⤜ (*λ view. D view*)) = *do* {
  let ($\mathcal{A}$*1*, $\mathcal{A}$*2*, $\mathcal{A}$*3*) = $\mathcal{A}$;
  r ← *sample-uniform* (*order* $\mathcal{G}$);
  α0 ← *sample-uniform* (*order* $\mathcal{G}$);
  α1 ← *sample-uniform* (*order* $\mathcal{G}$);
  let h0 = **g** [^] *α0*;
  let h1 = **g** [^] *α1*;
  let a = **g** [^] *r*;
  let b0 = (**g** [^] *r*) [^] *α0*;
  let b1 = *h1* [^] *r*;
  ((*in1*, *in2*, *in3*),*s*) ← $\mathcal{A}$*1 M h0 h1 a b0 b1 z*;
  let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
  (*b* :: *bool*, - :: *unit*) ← *funct-DH-ZK* (*in1*, *in2*, *in3*) ((*h*,*a*,*b*), *r*);
  - :: *unit* ← *assert-spmf* (*b*);
  (((*w0*,*z0*),(*w1*,*z1*)),*s'*) ← $\mathcal{A}$*2 h0 h1 a b0 b1 M s*;
  adv-out ← $\mathcal{A}$*3 s'*;
  D (*adv-out*, ((*z0* ⊗ (*inv w0* [^] *α0*))))}
  by(*simp add*: *P1-real-model-def assms split-def Let-def power-swap*)
 ultimately show *?thesis*
  by(*simp add*: *P1-real-model-def ddh.DDH0-def Let-def*)
qed

lemma *P1-ideal-ddh1-σ-false*:
assumes σ = *False*
shows ((*P1-ideal-model M σ z 𝒜*) ⤜ (λ *view*. *D view*)) = (*ddh.DDH1* (*P1-DDH-mal-adv-σ-false M z 𝒜 D*))
including *monad-normalisation*
proof−
have ((*P1-ideal-model M σ z 𝒜*) ⤜ (λ *view*. *D view*)) = *do* {
let (𝒜*1*, 𝒜*2*, 𝒜*3*) = 𝒜;
r ← *sample-uniform* (*order* 𝒢);
α0 ← *sample-uniform* (*order* 𝒢);
α1 ← *sample-uniform* (*order* 𝒢);
let h0 = **g** [^] *α0*;
let h1 = **g** [^] *α1*;
let a = **g** [^] *r*;
let b0 = (**g** [^] *r*) [^] *α0*;
let b1 = *h1* [^] *r* ⊗ **g**;
((*in1*, *in2*, *in3*),*s*) ← 𝒜*1 M h0 h1 a b0 b1 z*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
- :: *unit* ← *assert-spmf* ((*in1*, *in2*, *in3*) = (*h*,*a*,*b*));
(((*w0*,*z0*),(*w1*,*z1*)),*s'*) ← 𝒜*2 h0 h1 a b0 b1 M s*;
let x0 = (*z0* ⊗ (*inv w0* [^] *α0*));
let x1 = (*z1* ⊗ (*inv w1* [^] *α1*));
(-, *f-out2*) ← *funct-OT-12* (*x0*, *x1*) σ;
adv-out ← 𝒜*3 s'*;
D (*adv-out*, *f-out2*)}
by(*simp add*: *P1-ideal-model-def assms split-def Let-def power-swap*)
thus *?thesis*
by(*auto simp add*: *P1-ideal-model-def ddh.DDH1-def funct-OT-12-def Let-def assms*)
qed

lemma *P1-real-ddh1-σ-true*:
assumes σ = *True*
shows ((*P1-real-model M σ z 𝒜*) ⤜ (λ *view*. *D view*)) = (*ddh.DDH1* (*P1-DDH-mal-adv-σ-true M z 𝒜 D*))
including *monad-normalisation*
proof−
have (*in2* = **g** [^] (*r*::*nat*) ∧ *in3* = *in1* [^] *r* ∧ *in1* = **g** [^] (*α0*::*nat*) ⊗ *inv* (**g** [^] (*α1*::*nat*))
∧ *in2* = **g** [^] *r* ∧ *in3* = (**g** [^] *r*) [^] *α0* ⊗ **g** ⊗ *inv* ((**g** [^] *α1*) [^] *r* ⊗ **g**))
⟷ (*in1* = **g** [^] *α0* ⊗ *inv* (**g** [^] *α1*) ∧ *in2* = **g** [^] *r*
∧ *in3* = (**g** [^] *α0*) [^] *r* ⊗ **g** ⊗ *inv* ((**g** [^] *r*) [^] *α1* ⊗ **g**))
for *in1 in2 in3 r α0 α1*
by (*auto simp add*: *P1-assert-correct1 power-swap*)
moreover have ((*P1-real-model M σ z 𝒜*) ⤜ (λ *view*. *D view*)) = *do* {
let (𝒜*1*, 𝒜*2*, 𝒜*3*) = 𝒜;
r ← *sample-uniform* (*order* 𝒢);
α0 ← *sample-uniform* (*order* 𝒢);
α1 ← *sample-uniform* (*order* 𝒢);

let h0 = **g** [^] *α0*;
let h1 = **g** [^] *α1*;
let a = **g** [^] *r*;
let b0 = ((**g** [^] *r*) [^] *α0*) ⊗ **g**;
let b1 = (*h1* [^] *r*) ⊗ **g**;
((*in1*, *in2*, *in3*),*s*) ← *𝒜1 M h0 h1 a b0 b1 z*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
(*b* :: *bool*, - :: *unit*) ← *funct-DH-ZK* (*in1*, *in2*, *in3*) ((*h*,*a*,*b*), *r*);
- :: *unit* ← *assert-spmf* (*b*);
(((*w0*,*z0*),(*w1*,*z1*)),*s'*) ← *𝒜2 h0 h1 a b0 b1 M s*;
adv-out ← *𝒜3 s'*;
D (*adv-out*, ((*z1* ⊗ (*inv w1* [^] *α1*))))}
by(*simp add*: *P1-real-model-def assms split-def Let-def power-swap*)
ultimately show *?thesis*
by(*simp add*: *Let-def P1-real-model-def ddh.DDH1-def assms power-swap*)
qed

lemma *P1-ideal-ddh0-σ-true*:
assumes *σ* = *True*
shows ((*P1-ideal-model M σ z 𝒜*) ⤜ (*λ view. D view*)) = (*ddh.DDH0* (*P1-DDH-mal-adv-σ-true M z 𝒜 D*))
including *monad-normalisation*
proof−
have ((*P1-ideal-model M σ z 𝒜*) ⤜ (*λ view. D view*)) = *do* {
let (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
r ← *sample-uniform* (*order 𝒢*);
α0 ← *sample-uniform* (*order 𝒢*);
α1 ← *sample-uniform* (*order 𝒢*);
let h0 = **g** [^] *α0*;
let h1 = **g** [^] *α1*;
let a = **g** [^] *r*;
let b0 = (**g** [^] *r*) [^] *α0*;
let b1 = *h1* [^] *r* ⊗ **g**;
((*in1*, *in2*, *in3*),*s*) ← *𝒜1 M h0 h1 a b0 b1 z*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
- :: *unit* ← *assert-spmf* ((*in1*, *in2*, *in3*) = (*h*,*a*,*b*));
(((*w0*,*z0*),(*w1*,*z1*)),*s'*) ← *𝒜2 h0 h1 a b0 b1 M s*;
let x0 = (*z0* ⊗ (*inv w0* [^] *α0*));
let x1 = (*z1* ⊗ (*inv w1* [^] *α1*));
(-, *f-out2*) ← *funct-OT-12* (*x0*, *x1*) *σ*;
adv-out ← *𝒜3 s'*;
D (*adv-out*, *f-out2*)}
by(*simp add*: *P1-ideal-model-def assms Let-def split-def power-swap*)
thus *?thesis*
by(*simp add*: *split-def Let-def P1-ideal-model-def ddh.DDH0-def assms funct-OT-12-def power-swap*)
qed

lemma *P1-real-ideal-DDH-advantage-false*:

assumes σ = *False*
shows *mal-def.adv-P1 M* σ *z* (*P1-S1*, *P1-S2*) $\mathcal{A}$ *D* = *ddh.DDH-advantage* (*P1-DDH-mal-adv-*σ*-false M z* $\mathcal{A}$ *D*)
by(*simp add*: *P1-adv-real-ideal-model-def ddh.DDH-advantage-def P1-ideal-ddh1-*σ*-false P1-real-ddh0-*σ*-false assms P1-advantages-eq*)

lemma *P1-real-ideal-DDH-advantage-false-bound*:
assumes σ = *False*
shows *mal-def.adv-P1 M* σ *z* (*P1-S1*, *P1-S2*) $\mathcal{A}$ *D*
$\leq$ *ddh.advantage* (*P1-DDH-mal-adv-*σ*-false M z* $\mathcal{A}$ *D*)
\+ *ddh.advantage* (*ddh.DDH-*$\mathcal{A}'$ (*P1-DDH-mal-adv-*σ*-false M z* $\mathcal{A}$ *D*))
using *P1-real-ideal-DDH-advantage-false ddh.DDH-advantage-bound assms* **by** *metis*

lemma *P1-real-ideal-DDH-advantage-true*:
assumes σ = *True*
shows *mal-def.adv-P1 M* σ *z* (*P1-S1*, *P1-S2*) $\mathcal{A}$ *D* = *ddh.DDH-advantage* (*P1-DDH-mal-adv-*σ*-true M z* $\mathcal{A}$ *D*)
by(*simp add*: *P1-adv-real-ideal-model-def ddh.DDH-advantage-def P1-real-ddh1-*σ*-true P1-ideal-ddh0-*σ*-true assms P1-advantages-eq*)

lemma *P1-real-ideal-DDH-advantage-true-bound*:
assumes σ = *True*
shows *mal-def.adv-P1 M* σ *z* (*P1-S1*, *P1-S2*) $\mathcal{A}$ *D*
$\leq$ *ddh.advantage* (*P1-DDH-mal-adv-*σ*-true M z* $\mathcal{A}$ *D*)
\+ *ddh.advantage* (*ddh.DDH-*$\mathcal{A}'$ (*P1-DDH-mal-adv-*σ*-true M z* $\mathcal{A}$ *D*))
using *P1-real-ideal-DDH-advantage-true ddh.DDH-advantage-bound assms* **by** *metis*

lemma *P2-output-rewrite*:
assumes *s* < *order* $\mathcal{G}$
shows (**g** [^] (*r* ∗ *u1* + *v1*), **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ *inv* **g** [^] *u1*)
= (**g** [^] (*r* ∗ ((*s* + *u1*) *mod order* $\mathcal{G}$) + (*r* ∗ *order* $\mathcal{G}$ − *r* ∗ *s* + *v1*) *mod order* $\mathcal{G}$),
g [^] (*r* ∗ α ∗ ((*s* + *u1*) *mod order* $\mathcal{G}$) + (*r* ∗ *order* $\mathcal{G}$ − *r* ∗ *s* + *v1*) *mod order* $\mathcal{G}$ ∗ α)
⊗ *inv* **g** [^] ((*s* + *u1*) *mod order* $\mathcal{G}$ + (*order* $\mathcal{G}$ − *s*)))
proof−
have g [^] (*r* ∗ *u1* + *v1*) = **g** [^] (*r* ∗ ((*s* + *u1*) *mod order* $\mathcal{G}$) + (*r* ∗ *order* $\mathcal{G}$ − *r* ∗ *s* + *v1*) *mod order* $\mathcal{G}$)
proof−
have [(*r* ∗ *u1* + *v1*) = (*r* ∗ ((*s* + *u1*) *mod order* $\mathcal{G}$) + (*r* ∗ *order* $\mathcal{G}$ − *r* ∗ *s* + *v1*) *mod order* $\mathcal{G}$)] (*mod* (*order* $\mathcal{G}$))
proof−
have [(*r* ∗ ((*s* + *u1*) *mod order* $\mathcal{G}$) + (*r* ∗ *order* $\mathcal{G}$ − *r* ∗ *s* + *v1*) *mod order* $\mathcal{G}$) = *r* ∗ (*s* + *u1*) + (*r* ∗ *order* $\mathcal{G}$ − *r* ∗ *s* + *v1*)] (*mod* (*order* $\mathcal{G}$))

by (*metis* (*no-types*, *opaque-lifting*) *cong-def mod-add-left-eq mod-add-right-eq mod-mult-right-eq*)
hence [(*r* ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢) = *r* ∗ *s* + *r* ∗ *u1* + *r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*] (*mod* (*order* 𝒢))
by (*metis* (*no-types*, *lifting*) *Nat.add-diff-assoc add.assoc assms distrib-left less-or-eq-imp-le mult-le-mono*)
hence [(*r* ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢) = *r* ∗ *u1* + *r* ∗ *order* 𝒢 + *v1*] (*mod* (*order* 𝒢)) **by** *simp*
thus *?thesis*
by (*simp add*: *cong-def semiring-normalization-rules*(*23*))
qed
then show *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
moreover have **g** [^] (*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α)
⊗ *inv* **g** [^] ((*s* + *u1*) *mod order* 𝒢 + (*order* 𝒢 − *s*))
= **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ *inv* **g** [^] *u1*
proof−
have **g** [^] (*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α) = **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α)
proof−
have [(*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α) = *r* ∗ α ∗ *u1* + *v1* ∗ α] (*mod* (*order* 𝒢))
proof−
have [(*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α)
= *r* ∗ α ∗ (*s* + *u1*) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) ∗ α] (*mod* (*order* 𝒢))
using *cong-def mod-add-cong mod-mult-left-eq mod-mult-right-eq* **by** *blast*
hence [(*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α)
= *r* ∗ α ∗ *s* + *r* ∗ α ∗ *u1* + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) ∗ α] (*mod* (*order* 𝒢))
by (*simp add*: *distrib-left*)
hence [(*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α)
= *r* ∗ α ∗ *s* + *r* ∗ α ∗ *u1* + *r* ∗ *order* 𝒢 ∗ α − *r* ∗ *s* ∗ α + *v1* ∗ α] (*mod* (*order* 𝒢)) **using** *distrib-right assms*
by (*smt Groups.mult-ac*(*3*) *order-gt-0 Nat.add-diff-assoc2 add.commute diff-mult-distrib2 mult.commute mult-strict-mono order.strict-implies-order semiring-normalization-rules*(*25*) *zero-order*(*1*))
hence [(*r* ∗ α ∗ ((*s* + *u1*) *mod order* 𝒢) + (*r* ∗ *order* 𝒢 − *r* ∗ *s* + *v1*) *mod order* 𝒢 ∗ α)
= *r* ∗ α ∗ *u1* + *r* ∗ *order* 𝒢 ∗ α + *v1* ∗ α] (*mod* (*order* 𝒢)) **by** *simp*
thus *?thesis*
by (*simp add*: *cong-def semiring-normalization-rules*(*16*) *semiring-normalization-rules*(*23*))
qed
thus *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed

also have *inv* **g** [^] ((*s* + *u1*) *mod order* $\mathcal{G}$ + (*order* $\mathcal{G}$ − *s*)) = *inv* **g** [^] *u1*
proof−
have [((*s* + *u1*) *mod order* $\mathcal{G}$ + (*order* $\mathcal{G}$ − *s*)) = *u1*] (*mod* (*order* $\mathcal{G}$))
proof−
have [((*s* + *u1*) *mod order* $\mathcal{G}$ + (*order* $\mathcal{G}$ − *s*)) = *s* + *u1* + (*order* $\mathcal{G}$ − *s*)] (*mod* (*order* $\mathcal{G}$))
by (*simp add*: *add.commute mod-add-right-eq cong-def*)
hence [((*s* + *u1*) *mod order* $\mathcal{G}$ + (*order* $\mathcal{G}$ − *s*)) = *u1* + *order* $\mathcal{G}$] (*mod* (*order* $\mathcal{G}$))
using *assms* **by** *simp*
thus *?thesis* **by** (*simp add*: *cong-def*)
qed
hence **g** [^] ((*s* + *u1*) *mod order* $\mathcal{G}$ + (*order* $\mathcal{G}$ − *s*)) = **g** [^] *u1*
using *finite-group pow-generator-eq-iff-cong* **by** *blast*
thus *?thesis*
by (*metis generator-closed inverse-pow-pow*)
qed
ultimately show *?thesis* **by** *argo*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *P2-inv-g-rewrite*:
assumes *s* < *order* $\mathcal{G}$
shows (*inv* **g**) [^] (*u1'* + (*order* $\mathcal{G}$ − *s*)) = **g** [^] *s* ⊗ *inv* (**g** [^] *u1'*)
proof−
have *power-commute-rewrite*: **g** [^] (((*order* $\mathcal{G}$ − *s*) + *u1'*) *mod order* $\mathcal{G}$) = **g** [^] (*u1'* + (*order* $\mathcal{G}$ − *s*))
using *add.commute pow-generator-mod* **by** *metis*
have (*order* $\mathcal{G}$ − *s* + *u1'*) *mod order* $\mathcal{G}$ = (*int* (*order* $\mathcal{G}$) − *int s* + *int u1'*) *mod order* $\mathcal{G}$
by (*metis of-nat-add of-nat-diff order.strict-implies-order zmod-int assms*(*1*))
also have ... = (− *int s* + *int u1'*) *mod order* $\mathcal{G}$
by (*metis* (*full-types*) *add.commute minus-mod-self1 mod-add-right-eq*)
ultimately have (*order* $\mathcal{G}$ − *s* + *u1'*) *mod order* $\mathcal{G}$ = (− *int s mod* (*order* $\mathcal{G}$) + *int u1' mod* (*order* $\mathcal{G}$)) *mod order* $\mathcal{G}$
by *presburger*
hence **g** [^] (((*order* $\mathcal{G}$ − *s*) + *u1'*) *mod order* $\mathcal{G}$)
= **g** [^] ((− *int s mod* (*order* $\mathcal{G}$) + *int u1' mod* (*order* $\mathcal{G}$)) *mod order* $\mathcal{G}$)
by (*metis int-pow-int*)
hence **g** [^] (*u1'* + (*order* $\mathcal{G}$ − *s*))
= **g** [^] ((− *int s mod* (*order* $\mathcal{G}$) + *int u1' mod* (*order* $\mathcal{G}$)) *mod order* $\mathcal{G}$)
using *power-commute-rewrite* **by** *argo*
also have ...
= **g** [^] (− *int s mod* (*order* $\mathcal{G}$) + *int u1' mod* (*order* $\mathcal{G}$))
using *pow-generator-mod-int* **by** *blast*
also have ... = **g** [^] (− *int s mod* (*order* $\mathcal{G}$)) ⊗ **g** [^] (*int u1' mod* (*order* $\mathcal{G}$))

by (*simp add*: *int-pow-mult*)
also have ... = **g** [^] (− *int s*) ⊗ **g** [^] (*int u1'*)
using *pow-generator-mod-int* **by** *simp*
ultimately have *inv* (**g** [^] (*u1'* + (*order* $\mathcal{G}$ − *s*))) = *inv* (**g** [^] (− *int s*) ⊗ **g** [^] (*int u1'*)) **by** *simp*
hence *inv* (**g** [^] ((*u1'* + (*order* $\mathcal{G}$ − *s*)) *mod* (*order* $\mathcal{G}$))) = *inv* (**g** [^] (− *int s*)) ⊗ *inv* (**g** [^] (*int u1'*))
using *pow-generator-mod*
by (*simp add*: *inverse-split*)
also have ... = **g** [^] (*int s*) ⊗ *inv* (**g** [^] (*int u1'*))
by (*simp add*: *int-pow-neg*)
also have ... = **g** [^] *s* ⊗ *inv* (**g** [^] *u1'*)
by (*simp add*: *int-pow-int*)
ultimately show *?thesis*
by (*simp add*: *inverse-pow-pow pow-generator-mod*)
qed

lemma *P2-inv-g-s-rewrite*:
assumes *s* < *order* $\mathcal{G}$
shows **g** [^] ((*r*::*nat*) ∗ α ∗ *u1* + *v1* ∗ α) ⊗ *inv* **g** [^] (*u1* + (*order* $\mathcal{G}$ − *s*)) = **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ **g** [^] *s* ⊗ *inv* **g** [^] *u1*
proof−
have *in-carrier1*: *inv* **g** [^] (*u1* + (*order* $\mathcal{G}$ − *s*)) ∈ *carrier* $\mathcal{G}$ **by** *blast*
have *in-carrier2*: *inv* **g** [^] *u1* ∈ *carrier* $\mathcal{G}$ **by** *simp*
have *in-carrier-3*: **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ∈ *carrier* $\mathcal{G}$ **by** *simp*
have **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ (*inv* **g** [^] (*u1* + (*order* $\mathcal{G}$ − *s*))) = **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ (**g** [^] *s* ⊗ *inv* **g** [^] *u1*)
using *P2-inv-g-rewrite assms*
by (*simp add*: *inverse-pow-pow*)
thus *?thesis* **using** *cyclic-group-assoc in-carrier1 in-carrier2* **by** *auto*
qed

lemma *P2-e0-rewrite*:
assumes *s* < *order* $\mathcal{G}$
shows (**g** [^] (*r* ∗ *x* + *xa*), **g** [^] (*r* ∗ α ∗ *x* + *xa* ∗ α) ⊗ **g** [^] *x*) =
(**g** [^] (*r* ∗ ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$) + (*r* ∗ *s* + *xa*) *mod order* $\mathcal{G}$),
g [^] (*r* ∗ α ∗ ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$) + (*r* ∗ *s* + *xa*) *mod order* $\mathcal{G}$ ∗ α)
⊗ **g** [^] ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$ + *s*))
proof−
have **g** [^] (*r* ∗ *x* + *xa*) = **g** [^] (*r* ∗ ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$) + (*r* ∗ *s* + *xa*) *mod order* $\mathcal{G}$)
proof−
have [(*r* ∗ *x* + *xa*) = (*r* ∗ ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$) + (*r* ∗ *s* + *xa*) *mod order* $\mathcal{G}$)] (*mod order* $\mathcal{G}$)
proof−
have [(*r* ∗ ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$) + (*r* ∗ *s* + *xa*) *mod order* $\mathcal{G}$)
= (*r* ∗ ((*order* $\mathcal{G}$ − *s* + *x*)) + (*r* ∗ *s* + *xa*))] (*mod order* $\mathcal{G}$)

by (*metis* (*no-types*, *lifting*) *mod-mod-trivial cong-add cong-def mod-mult-right-eq*)
hence [($r * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G}$)
$= r * (order\ \mathcal{G} - s) + r * x + r * s + xa$] (*mod order* $\mathcal{G}$)
by (*simp add*: *add.assoc distrib-left*)
hence [($r * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G}$)
$= r * x + r * s + r * (order\ \mathcal{G} - s) + xa$] (*mod order* $\mathcal{G}$)
by (*metis add.assoc add.commute*)
hence [($r * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G}$)
$= r * x + r * s + r * order\ \mathcal{G} - r * s + xa$] (*mod order* $\mathcal{G}$)
proof −
have [($xa + r * s$) *mod order* $\mathcal{G} + r * ((x + (order\ \mathcal{G} - s))\ mod\ order\ \mathcal{G})$
$= xa + r * (s + x + (order\ \mathcal{G} - s))$] (*mod order* $\mathcal{G}$)
by (*metis* (*no-types*) ‹[$r * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} = r * x + r * s + r * (order\ \mathcal{G} - s) + xa$] (*mod order* $\mathcal{G}$)› *add.commute distrib-left*)
then show *?thesis*
by (*simp add*: *assms add.commute distrib-left order.strict-implies-order*)
qed
hence [($r * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G}$)
$= r * x + xa$] (*mod order* $\mathcal{G}$)
proof −
have [($xa + r * s$) *mod order* $\mathcal{G} + r * ((x + (order\ \mathcal{G} - s))\ mod\ order\ \mathcal{G})$
$= xa + (r * x + r * order\ \mathcal{G})$] (*mod order* $\mathcal{G}$)
by (*metis* (*no-types*) ‹[$r * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} = r * x + r * s + r * order\ \mathcal{G} - r * s + xa$] (*mod order* $\mathcal{G}$)› *add.commute add.left-commute add-diff-cancel-left'*)
then show *?thesis*
by (*metis* (*no-types*) *add.commute cong-add-lcancel-nat cong-def distrib-left mod-add-self2 mod-mult-right-eq*)
qed
then show *?thesis* **using** *cong-def* **by** *metis*
qed
then show *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
moreover have g [^] ($r * \alpha * x + xa * \alpha$) ⊗ **g** [^] x =
g [^] ($r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha$)
⊗ **g** [^] (($order\ \mathcal{G} - s + x$) *mod order* $\mathcal{G} + s$)
proof−
have g [^] ($r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha$)
= **g** [^] ($r * \alpha * x + xa * \alpha$)
proof−
have [$r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha) = r * \alpha * x + xa * \alpha$] (*mod order* $\mathcal{G}$)
proof−
have [$r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha$)
$= (r * \alpha * ((order\ \mathcal{G} - s) + x) + (r * s + xa) * \alpha)$] (*mod order* $\mathcal{G}$)

by (*metis* (*no-types*, *lifting*) *cong-add cong-def mod-mult-left-eq mod-mult-right-eq*)

hence $[(r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha)$
$= r * \alpha * (order\ \mathcal{G} - s) + r * \alpha * x + r * s * \alpha + xa * \alpha]\ (mod\ order\ \mathcal{G})$
by (*simp add*: *add.assoc distrib-left distrib-right*)
hence $[(r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha)$
$= r * \alpha * x + r * s * \alpha + r * \alpha * (order\ \mathcal{G} - s) + xa * \alpha]\ (mod\ order\ \mathcal{G})$
by (*simp add*: *add.commute add.left-commute*)
hence $[(r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha)$
$= r * \alpha * x + r * s * \alpha + r * \alpha * order\ \mathcal{G} - r * \alpha * s + xa * \alpha]\ (mod\ order\ \mathcal{G})$
proof −
have $\forall n\ na.\ \neg\ (n{::}nat) \leq na \vee n + (na - n) = na$
by (*meson ordered-cancel-comm-monoid-diff-class.add-diff-inverse*)
then have $r * \alpha * s + r * \alpha * (order\ \mathcal{G} - s) = r * \alpha * order\ \mathcal{G}$
by (*metis add-mult-distrib2 assms less-or-eq-imp-le*)
then have $r * \alpha * x + r * s * \alpha + r * \alpha * order\ \mathcal{G} = r * \alpha * s + r * \alpha * (order\ \mathcal{G} - s) + (r * \alpha * x + r * s * \alpha)$
by *presburger*
then have *f1*: $r * \alpha * x + r * s * \alpha + r * \alpha * order\ \mathcal{G} - r * \alpha * s = r * \alpha * s + r * \alpha * (order\ \mathcal{G} - s) - r * \alpha * s + (r * \alpha * x + r * s * \alpha)$
by *simp*
have $r * \alpha * s + r * \alpha * (order\ \mathcal{G} - s) = r * \alpha * (order\ \mathcal{G} - s) + r * \alpha * s$
by *presburger*
then have $r * \alpha * x + r * s * \alpha + r * \alpha * order\ \mathcal{G} - r * \alpha * s = r * \alpha * x + r * s * \alpha + r * \alpha * (order\ \mathcal{G} - s)$
using *f1 diff-add-inverse2* **by** *presburger*
then show *?thesis*
using ‹$[r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha = r * \alpha * x + r * s * \alpha + r * \alpha * (order\ \mathcal{G} - s) + xa * \alpha]\ (mod\ order\ \mathcal{G})$› **by** *presburger*
qed
hence $[(r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha)$
$= r * \alpha * x + r * \alpha * s + r * \alpha * order\ \mathcal{G} - r * \alpha * s + xa * \alpha]\ (mod\ order\ \mathcal{G})$
using *add.commute add.assoc* **by** *force*
hence $[(r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha)$
$= r * \alpha * x + r * \alpha * order\ \mathcal{G} + xa * \alpha]\ (mod\ order\ \mathcal{G})$ **by** *simp*
thus *?thesis* **using** *cong-def semiring-normalization-rules*(*23*)
by (*simp add*: ‹$\bigwedge c\ b\ a.\ [b = c]\ (mod\ a) = (b\ mod\ a = c\ mod\ a)$› ‹$\bigwedge c\ b\ a.\ a + b + c = a + c + b$›)

qed
thus *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
also have g [^] ((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$ + *s*) = **g** [^] *x*
proof−
have [((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$ + *s*) = *x*] (*mod order* $\mathcal{G}$)
proof−
have [((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$ + *s*) = (*order* $\mathcal{G}$ − *s* + *x*+ *s*)] (*mod order* $\mathcal{G}$)
by (*simp add*: *add.commute cong-def mod-add-right-eq*)
hence [((*order* $\mathcal{G}$ − *s* + *x*) *mod order* $\mathcal{G}$ + *s*) = (*order* $\mathcal{G}$ + *x*)] (*mod order* $\mathcal{G}$)
using *assms* **by** *auto*
thus *?thesis*
by (*simp add*: *cong-def*)
qed
thus *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
ultimately show *?thesis* **by** *argo*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *P2-case-l-new-1-gt-e0-rewrite*:
assumes *s* < *order* $\mathcal{G}$
shows (**g** [^] (*r* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *x*) *mod order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *xa*) *mod order* $\mathcal{G}$),
g [^] (*r* ∗ α ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *x*) *mod order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *xa*) *mod order* $\mathcal{G}$ ∗ α) ⊗
g [^] (*t* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *x*) *mod order* $\mathcal{G}$
+ *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$))))) = (**g** [^] (*r* ∗ *x* + *xa*), **g** [^] (*r* ∗ α ∗ *x* + *xa* ∗ α) ⊗ **g** [^] (*t* ∗ *x*))
proof−
have g [^] ((*r*::*nat*) ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *x*) *mod order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) + *xa*) *mod order* $\mathcal{G}$)
= **g** [^] (*r* ∗ *x* + *xa*)
proof(*cases r* = *0*)
case *True*
then show *?thesis*
by (*simp add*: *pow-generator-mod*)
next
case *False*

have [(*r*::*nat*) ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *x*) *mod* *order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *xa*) *mod* *order* $\mathcal{G}$ = *r* ∗ *x* + *xa*] (*mod* *order* $\mathcal{G}$)
proof−
have [*r* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* (*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$) + *x*) *mod* *order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ *nat* (*fst* (*bezw* *t* (*order* $\mathcal{G}$))) + *xa*) *mod* *order* $\mathcal{G}$
= (*r* ∗ (((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* (*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *x*))
+ (*r* ∗ *s* ∗ *nat* (*fst* (*bezw* *t* (*order* $\mathcal{G}$))) + *xa*))] (*mod* *order* $\mathcal{G}$)
proof −
have *order* $\mathcal{G} \neq 0$
using *order-gt-0* **by** *simp*
then show *?thesis*
using *cong-add cong-def mod-mult-right-eq*
by (*smt mod-mod-trivial*)
qed
hence [*r* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *x*) *mod* *order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *xa*) *mod* *order* $\mathcal{G}$
= *r* ∗ (*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$))) + *r* ∗ *x*
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *xa*)] (*mod* *order* $\mathcal{G}$)
proof −
have [*r* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *x*) *mod* *order* $\mathcal{G}$)
= *r* ∗ (*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$))) + *r* ∗ *x*] (*mod* *order* $\mathcal{G}$)
by (*simp add*: *cong-def distrib-left mod-mult-right-eq*)
then show *?thesis*
using *assms cong-add gr-implies-not0* **by** *fastforce*
qed
hence [*r* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *x*) *mod* *order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *xa*) *mod* *order* $\mathcal{G}$
= *r* ∗ *order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *r* ∗ *x*
+ *r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *xa*] (*mod* *order* $\mathcal{G}$)
by (*simp add*: *ab-semigroup-mult-class.mult-ac*(*1*) *right-diff-distrib′ add.assoc*)
hence [*r* ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *x*) *mod* *order* $\mathcal{G}$)
+ (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *xa*) *mod* *order* $\mathcal{G}$

= *r* ∗ *order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ + *r* ∗ *x* + *xa*] (*mod order* $\mathcal{G}$)
proof−
have *r* ∗ *order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *r* ∗ *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) > *0*
proof−
have *order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ > *s* ∗ (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$))
proof−
have (*nat* ((*fst* (*bezw t* (*order* $\mathcal{G}$))) *mod order* $\mathcal{G}$)) ≤ *order* $\mathcal{G}$
proof −
have ∀ *x0 x1*. ((*x0*::*int*) *mod x1* < *x1*) = (¬ *x1* + − *1* ∗ (*x0 mod x1*) ≤ *0*)
by *linarith*
then have ¬ *int* (*order* $\mathcal{G}$) + − *1* ∗ (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$)) ≤ *0*
using *of-nat-0-less-iff order-gt-0* **by** *fastforce*
then show *?thesis*
by *linarith*
qed
thus *?thesis* **using** *assms*
proof −
have ∀ *n na*. ¬ *n* ≤ *na* ∨ ¬ *na* ∗ *order* $\mathcal{G}$ < *n* ∗ *nat* (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$))
by (*meson* ‹*nat* (*fst* (*bezw* (*t*::*nat*) (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$)) ≤ *order* $\mathcal{G}$› *mult-le-mono not-le*)
then show *?thesis*
by(*metis* (*no-types*, *opaque-lifting*) ‹(*s*::*nat*) < *order* $\mathcal{G}$› *mult-less-cancel2 nat-less-le not-le not-less-zero*)
qed
qed
thus *?thesis* **using** *False*
by *auto*
qed
thus *?thesis*
proof −
have *r* ∗ *order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ + *r* ∗ *x* + *xa* = *r* ∗ (*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ *nat* (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$))) + (*r* ∗ *s* ∗ *nat* (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$)) + *xa*) + *r* ∗ *x*
using ‹(*0*::*nat*) < (*r*::*nat*) ∗ *order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *r* ∗ (*s*::*nat*) ∗ *nat* (*fst* (*bezw* (*t*::*nat*) (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$))› *diff-mult-distrib2* **by** *force*
then show *?thesis*
by (*metis* (*no-types*) ‹[(*r*::*nat*) ∗ ((*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − (*s*::*nat*) ∗ *nat* (*fst* (*bezw* (*t*::*nat*) (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$)) + (*x*::*nat*)) *mod order* $\mathcal{G}$) + (*r* ∗ *s* ∗ *nat* (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$)) + (*xa*::*nat*)) *mod order* $\mathcal{G}$ = *r* ∗ (*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − *s* ∗ *nat* (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$))) + *r* ∗ *x* + (*r* ∗ *s* ∗ *nat* (*fst* (*bezw t* (*order* $\mathcal{G}$)) *mod int* (*order* $\mathcal{G}$)) + *xa*)] (*mod order* $\mathcal{G}$)› *semiring-normalization-rules*(*23*))
qed
qed

hence [r * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$)
+ (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$
= r * x + xa] (mod order $\mathcal{G}$)
by (*metis* (*no-types*, *lifting*) *mod-mult-self4 add.assoc mult.commute cong-def*)
thus *?thesis* **by** *blast*
qed
then show *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
moreover have g [^] (r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$)
+ (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$ * α) $\otimes$
g [^] (t * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$
+ s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$))))
= **g** [^] (r * α * x + xa * α) $\otimes$ **g** [^] (t * x)
proof−
have g [^] (r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$) + (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$ * α)
= **g** [^] (r * α * x + xa * α)
proof−
have [r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$) + (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$ * α
= r * α * x + xa * α] (mod order $\mathcal{G}$)
proof−
have [r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$) + (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$ * α
= r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$))) + x) + (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) * α] (mod order $\mathcal{G}$)
proof −
show *?thesis*
by (*meson cong-def mod-add-cong mod-mult-left-eq mod-mult-right-eq*)
qed
hence *mod-eq*: [r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$) + (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$ * α
= r * α * (order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$))) + r * α * x + r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) * α + xa * α] (mod order $\mathcal{G}$)
by (*simp add*: *distrib-left distrib-right add.assoc*)
hence *mod-eq'*: [r * α * ((order $\mathcal{G}$ * order $\mathcal{G}$ − s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + x) mod order $\mathcal{G}$) + (r * s * (nat ((fst (bezw t (order $\mathcal{G}$))) mod order $\mathcal{G}$)) + xa) mod order $\mathcal{G}$ * α

$= r * \alpha * order\ \mathcal{G} * order\ \mathcal{G} - r * \alpha * s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) + r * \alpha * x + r * \alpha * s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) + xa * \alpha]\ (mod\ order\ \mathcal{G})$

by (*simp add*: *semiring-normalization-rules*(*16*) *diff-mult-distrib2 semiring-normalization-rules*(*18*))

hence $[r * \alpha * ((order\ \mathcal{G} * order\ \mathcal{G} - s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) + x)\ mod\ order\ \mathcal{G}) + (r * s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) + xa)\ mod\ order\ \mathcal{G} * \alpha$

$= r * \alpha * order\ \mathcal{G} * order\ \mathcal{G} + r * \alpha * x + xa * \alpha]\ (mod\ order\ \mathcal{G})$

proof(*cases* $r * \alpha = 0$)

case *True*

then show *?thesis*

by (*metis mod-eq′ diff-zero mult-0 plus-nat.add-0*)

next

case *False*

hence *bound*: $r * \alpha * order\ \mathcal{G} * order\ \mathcal{G} - r * \alpha * s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) > 0$

proof−

have $s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) < order\ \mathcal{G} * order\ \mathcal{G}$

using *assms*

by (*simp add*: *mult-strict-mono nat-less-iff*)

thus *?thesis*

using *False* **by** *auto*

qed

thus *?thesis*

proof −

have $r * \alpha * order\ \mathcal{G} * order\ \mathcal{G} = r * \alpha * (order\ \mathcal{G} * order\ \mathcal{G} - s * nat\ (fst\ (bezw\ t\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G})))$

$+ r * s * nat\ (fst\ (bezw\ t\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G})) * \alpha$

using *bound diff-mult-distrib2* **by** *force*

then have $r * \alpha * order\ \mathcal{G} * order\ \mathcal{G} + r * \alpha * x = r * \alpha * (order\ \mathcal{G} * order\ \mathcal{G} - s * nat\ (fst\ (bezw\ t\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G})))$

$+ r * \alpha * x + r * s * nat\ (fst\ (bezw\ t\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G})) * \alpha$

by *presburger*

then show *?thesis*

using *mod-eq* **by** *presburger*

qed

qed

thus *?thesis*

by (*metis* (*mono-tags*, *lifting*) *add.assoc cong-def mod-mult-self3*)

qed

then show *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*

qed

also have g $[\widehat{\ }]\ (t * ((order\ \mathcal{G} * order\ \mathcal{G} - s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G})) + x)\ mod\ order\ \mathcal{G} + s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G}))) \ mod\ order\ \mathcal{G}))))$

= **g** [^] $(t * x)$
proof −
have [*t* ∗ ((*order* 𝒢 ∗ *order* 𝒢 − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) + *x*) *mod order* 𝒢 + *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢))) = *t* ∗ *x*] (*mod order* 𝒢)
proof −
have [*t* ∗ ((*order* 𝒢 ∗ *order* 𝒢 − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) + *x*) *mod order* 𝒢 + *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))
= (*t* ∗ (*order* 𝒢 ∗ *order* 𝒢 − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) + *x* + *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢))))] (*mod order* 𝒢)
using *cong-def mod-add-left-eq mod-mult-cong* **by** *blast*
hence [*t* ∗ ((*order* 𝒢 ∗ *order* 𝒢 − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) + *x*) *mod order* 𝒢 + *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))
= *t* ∗ (*order* 𝒢 ∗ *order* 𝒢 + *x*)] (*mod order* 𝒢)
proof −
have *order* 𝒢 ∗ *order* 𝒢 − *s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) > *0*
proof −
have (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) ≤ *order* 𝒢
using *nat-le-iff order.strict-implies-order order-gt-0*
by (*simp add*: *order.strict-implies-order*)
thus *?thesis*
by (*metis assms diff-mult-distrib le0 linorder-neqE-nat mult-strict-mono not-le zero-less-diff*)
qed
thus *?thesis*
using ‹[(*t*::*nat*) ∗ ((*order* 𝒢 ∗ *order* 𝒢 − (*s*::*nat*) ∗ *nat* (*fst* (*bezw t* (*order* 𝒢)) *mod int* (*order* 𝒢)) + (*x*::*nat*)) *mod order* 𝒢 + *s* ∗ *nat* (*fst* (*bezw t* (*order* 𝒢)) *mod int* (*order* 𝒢))) = *t* ∗ (*order* 𝒢 ∗ *order* 𝒢 − *s* ∗ *nat* (*fst* (*bezw t* (*order* 𝒢)) *mod int* (*order* 𝒢)) + *x* + *s* ∗ *nat* (*fst* (*bezw t* (*order* 𝒢)) *mod int* (*order* 𝒢)))] (*mod order* 𝒢)› **by** *auto*
qed
thus *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *add.commute cong-def mod-mult-right-eq mod-mult-self1*)
qed
thus *?thesis* **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
ultimately show *?thesis* **by** *argo*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *P2-case-l-neq-1-gt-x0-rewrite*:
assumes *t* < *order* 𝒢
and *t* ≠ *0*
shows g [^] (*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢))))) = **g** [^] (*t* ∗ *u0*) ⊗ **g** [^] *s*
proof −

from *assms* **have** *gcd*: *gcd t* (*order* 𝒢) = *1*
using *prime-field coprime-imp-gcd-eq-1* **by** *blast*
hence *inverse-t*: [*s* ∗ (*t* ∗ (*fst* (*bezw t* (*order* 𝒢)))) = *s* ∗ *1*] (*mod order* 𝒢)
by (*metis Num.of-nat-simps*(*2*) *Num.of-nat-simps*(*5*) *cong-scalar-left order-gt-0 inverse*)
hence *inverse-t'*: [*t* ∗ *u0* + *s* ∗ (*t* ∗ (*fst* (*bezw t* (*order* 𝒢)))) =*t* ∗ *u0* + *s* ∗ *1*] (*mod order* 𝒢)
using *cong-add-lcancel* **by** *fastforce*
have *eq*: *int* (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) = (*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢
proof−
have (*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢 ≥ *0* **using** *order-gt-0* **by** *simp*
hence (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)) = (*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢 **by** *linarith*
thus *?thesis* **by** *blast*
qed
have [(*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢))))) = *t* ∗ *u0* + *s*] (*mod order* 𝒢)
proof−
have [*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))) = *t* ∗ *u0* + *t* ∗ (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))] (*mod order* 𝒢)
by (*simp add*: *distrib-left*)
hence [*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))) = *t* ∗ *u0* + *s* ∗ (*t* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))] (*mod order* 𝒢)
by (*simp add*: *ab-semigroup-mult-class.mult-ac*(*1*) *mult.left-commute*)
hence [*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))) = *t* ∗ *u0* + *s* ∗ (*t* ∗ (((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))] (*mod order* 𝒢)
using *eq*
by (*simp add*: *distrib-left mult.commute semiring-normalization-rules*(*18*))
hence [*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))) = *t* ∗ *u0* + *s* ∗ (*t* ∗ (*fst* (*bezw t* (*order* 𝒢))))] (*mod order* 𝒢)
by (*metis* (*no-types*, *opaque-lifting*) *cong-def mod-add-right-eq mod-mult-right-eq*)
hence [*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢)))) = *t* ∗ *u0* + *s* ∗ *1*] (*mod order* 𝒢) **using** *inverse-t'*
using *cong-trans cong-int-iff* **by** *blast*
thus *?thesis* **by** *simp*
qed
hence **g** [^] (*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw t* (*order* 𝒢))) *mod order* 𝒢))))) = **g** [^] (*t* ∗ *u0* + *s*) **using** *finite-group pow-generator-eq-iff-cong* **by** *blast*
thus *?thesis*
by (*simp add*: *nat-pow-mult*)
qed

Now we show the two end definitions are equal when the input for l (in the ideal model, the second input) is the one constructed by the simulator

lemma *P2-ideal-real-end-eq*:
assumes *b0-inv-b1*: *b0* ⊗ *inv b1* = (*h0* ⊗ *inv h1*) [^] *r*
and *assert-in-carrier*: *h0* ∈ *carrier* 𝒢 ∧ *h1* ∈ *carrier* 𝒢 ∧ *b0* ∈ *carrier* 𝒢 ∧ *b1* ∈ *carrier* 𝒢

and *x1-in-carrier*: *x1* ∈ *carrier* $\mathcal{G}$
and *x0-in-carrier*: *x0* ∈ *carrier* $\mathcal{G}$
shows *P2-ideal-model-end* (*x0*,*x1*) (*b0* ⊗ (*inv* (*h0* [^] *r*))) ((*h0*,*h1*, **g** [^] (*r*::*nat*),*b0*,*b1*),*s'*) $\mathcal{A}3$ = *P2-real-model-end* (*x0*,*x1*) ((*h0*,*h1*, **g** [^] (*r*::*nat*),*b0*,*b1*),*s'*) $\mathcal{A}3$
including *monad-normalisation*
proof(*cases* (*b0* ⊗ (*inv* (*h0* [^] *r*))) = **1**) — The case distinctions follow the 3 cases give on p 193/194*)
case *True*
have *b1-h1*: *b1* = *h1* [^] *r*
proof−
from *b0-inv-b1 assert-in-carrier* **have** *b0* ⊗ *inv b1* = *h0* [^] *r* ⊗ *inv h1* [^] *r*
by (*simp add*: *pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*)
hence *b0* ⊗ *inv h0* [^] *r* = *b1* ⊗ *inv h1* [^] *r*
by (*metis Units-eq Units-l-cancel local.inv-equality True assert-in-carrier cyclic-group.inverse-pow-pow cyclic-group-axioms inv-closed nat-pow-closed r-inv*)
with *True* **have** **1** = *b1* ⊗ *inv h1* [^] *r*
by (*simp add*: *assert-in-carrier inverse-pow-pow*)
hence **1** ⊗ *h1* [^] *r* = *b1*
by (*metis assert-in-carrier cyclic-group.inverse-pow-pow cyclic-group-axioms inv-closed inv-inv l-one local.inv-equality nat-pow-closed*)
thus *?thesis*
using *assert-in-carrier l-one* **by** *blast*
qed
obtain α :: *nat* **where** α: **g** [^] α = *h1* **and** α < *order* $\mathcal{G}$
by (*metis mod-less-divisor assert-in-carrier generatorE order-gt-0 pow-generator-mod*)

obtain *s* :: *nat* **where** *s*: **g** [^] *s* = *x1* **and** *s-lt*: *s* < *order* $\mathcal{G}$
by (*metis assms*(*3*) *mod-less-divisor generatorE order-gt-0 pow-generator-mod*)
have *b1* ⊗ *inv* **g** = **g** [^] (*r* ∗ α) ⊗ *inv* **g**
by (*metis* α *b1-h1 generator-closed mult.commute nat-pow-pow*)
have *a-g-exp-rewrite*: (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0* = **g** [^] (*r* ∗ *u0* + *v0*)
for *u0 v0*
by (*simp add*: *nat-pow-mult nat-pow-pow*)
have *z1-rewrite*: (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ **1** = **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ *inv* **g** [^] *u1*
for *u1 v1* :: *nat*
by (*smt* α *b1-h1 pow-mult-distrib cyclic-group-commute generator-closed inv-closed m-assoc m-closed monoid-comm-monoidI mult.commute nat-pow-closed nat-pow-mult nat-pow-pow r-one*)
have *z1-rewrite'*: **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ **g** [^] *s* ⊗ *inv* **g** [^] *u1* = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*
for *u1 v1*
using *assert-in-carrier cyclic-group-commute m-assoc s z1-rewrite* **by** *auto*
have *P2-ideal-model-end* (*x0*,*x1*) (*b0* ⊗ (*inv* (*h0* [^] *r*))) ((*h0*,*h1*, **g** [^] (*r*::*nat*),*b0*,*b1*),*s'*) $\mathcal{A}3$ = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);

v1 ← sample-uniform (order 𝒢);
let w0 = (**g** [^] (r::nat)) [^] u0 ⊗ **g** [^] v0;
let w1 = (**g** [^] (r::nat)) [^] u1 ⊗ **g** [^] v1;
let z0 = b0 [^] u0 ⊗ h0 [^] v0 ⊗ x0;
let z1 = (b1 ⊗ inv **g**) [^] u1 ⊗ h1 [^] v1 ⊗ **1**;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ← 𝒜3 e0 e1 s′;
return-spmf ((), out)}
by(simp add: P2-ideal-model-end-def True funct-OT-12-def)
also have ... = do {
u0 ← sample-uniform (order 𝒢);
v0 ← sample-uniform (order 𝒢);
u1 ← sample-uniform (order 𝒢);
v1 ← sample-uniform (order 𝒢);
let w0 = (**g** [^] (r::nat)) [^] u0 ⊗ **g** [^] v0;
let w1 = (**g** [^] (r::nat)) [^] u1 ⊗ **g** [^] v1;
let z0 = b0 [^] u0 ⊗ h0 [^] v0 ⊗ x0;
let z1 = **g** [^] (r ∗ α ∗ u1 + v1 ∗ α) ⊗ inv **g** [^] u1;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ← 𝒜3 e0 e1 s′;
return-spmf ((), out)}
by(simp add: z1-rewrite)
also have ... = do {
u0 ← sample-uniform (order 𝒢);
v0 ← sample-uniform (order 𝒢);
u1 ← sample-uniform (order 𝒢);
v1 ← sample-uniform (order 𝒢);
let w0 = (**g** [^] (r::nat)) [^] u0 ⊗ **g** [^] v0;
let w1 = **g** [^] (r ∗ u1 + v1);
let z0 = b0 [^] u0 ⊗ h0 [^] v0 ⊗ x0;
let z1 = **g** [^] (r ∗ α ∗ u1 + v1 ∗ α) ⊗ inv **g** [^] u1;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ← 𝒜3 e0 e1 s′;
return-spmf ((), out)}
by(simp add: a-g-exp-rewrite)
also have ... = do {
u0 ← sample-uniform (order 𝒢);
v0 ← sample-uniform (order 𝒢);
u1 ← map-spmf (λ u1′. (s + u1′) mod (order 𝒢)) (sample-uniform (order 𝒢));
v1 ← map-spmf (λ v1′. ((r ∗ order 𝒢 − r ∗ s) + v1′) mod (order 𝒢)) (sample-uniform (order 𝒢));
let w0 = (**g** [^] (r::nat)) [^] u0 ⊗ **g** [^] v0;
let w1 = **g** [^] (r ∗ u1 + v1);
let z0 = b0 [^] u0 ⊗ h0 [^] v0 ⊗ x0;
let z1 = **g** [^] (r ∗ α ∗ u1 + v1 ∗ α) ⊗ inv **g** [^] (u1 + (order 𝒢 − s));
let e0 = (w0,z0);

let e1 = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0 e1 s'*;
return-spmf ((), *out*)}
apply(*simp add*: *bind-map-spmf o-def Let-def*)
using *P2-output-rewrite assms s-lt assms* **by** *presburger*
also have ... = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0*;
let w1 = **g** [^] (*r* ∗ *u1* + *v1*);
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*;
let z1 = **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ *inv* **g** [^] (*u1* + (*order* $\mathcal{G}$ − *s*));
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0 e1 s'*;
return-spmf ((), *out*)}
by(*simp add*: *samp-uni-plus-one-time-pad*)
also have ... = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0*;
let w1 = **g** [^] (*r* ∗ *u1* + *v1*);
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*;
let z1 = **g** [^] (*r* ∗ α ∗ *u1* + *v1* ∗ α) ⊗ **g** [^] *s* ⊗ *inv* **g** [^] *u1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0 e1 s'*;
return-spmf ((), *out*)}
by(*simp add*: *P2-inv-g-s-rewrite assms s-lt cong*: *bind-spmf-cong-simp*)
also have ... = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0*;
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*;
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0 e1 s'*;
return-spmf ((), *out*)}
by(*simp add*: *a-g-exp-rewrite z1-rewrite'*)
ultimately show *?thesis*
by(*simp add*: *P2-real-model-end-def*)

next
  obtain α :: *nat* **where** α: **g** [^] α = *h0*
    using *generatorE assms*
    using *assert-in-carrier* **by** *auto*
  have *w0-rewrite*: **g** [^] (*r* ∗ *u0* + *v0*) = (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0*
    for *u0 v0*
    by (*simp add*: *nat-pow-mult nat-pow-pow*)
  have *order-gt-0*: *order* $\mathcal{G}$ > *0* **using** *order-gt-0* **by** *simp*
  obtain *s* :: *nat* **where** *s*: **g** [^] *s* = *x0* **and** *s-lt*: *s* < *order* $\mathcal{G}$
   by (*metis mod-less-divisor generatorE order-gt-0 pow-generator-mod x0-in-carrier*)
  case *False* — case 2
  hence *l-neq-1*: (*b0* ⊗ (*inv* (*h0* [^] *r*))) ≠ **1by** *auto*
  then show *?thesis*
  proof(*cases* (*b0* ⊗ (*inv* (*h0* [^] *r*))) = **g**)
    case *True*
    hence *b0* = **g** ⊗ *h0* [^] *r*
      by (*metis assert-in-carrier generator-closed inv-solve-right nat-pow-closed*)
    hence *b0* = **g** ⊗ **g** [^] (*r* ∗ α)
      by (*metis* α *generator-closed mult.commute nat-pow-pow*)
    have *z0-rewrite*: *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ **1** = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] *u0*
      for *u0 v0* :: *nat*
     by (*smt* α ‹*b0* = **g** ⊗ **g** [^] (*r* ∗ α)› *pow-mult-distrib cyclic-group-commute generator-closed m-assoc monoid-comm-monoidI mult.commute nat-pow-closed nat-pow-mult nat-pow-pow r-one*)
    have *z0-rewrite′*: **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*u0* + *s*) = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] *u0* ⊗ **g** [^] *s*
      for *u0 v0*
      by (*simp add*: *add.assoc nat-pow-mult*)
    have *z0-rewrite″*: **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] *u0* ⊗ *x0* = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*
      for *u0 v0* **using** *z0-rewrite*
      using *assert-in-carrier* **by** *auto*
   have *P2-ideal-model-end* (*x0*,*x1*) (*b0* ⊗ (*inv* (*h0* [^] *r*))) ((*h0*,*h1*,**g** [^] (*r*::*nat*),*b0*,*b1*),*s′*) $\mathcal{A}3$ = *do* {
    u0 ← *sample-uniform* (*order* $\mathcal{G}$);
    v0 ← *sample-uniform* (*order* $\mathcal{G}$);
    u1 ← *sample-uniform* (*order* $\mathcal{G}$);
    v1 ← *sample-uniform* (*order* $\mathcal{G}$);
    let w0 = (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0*;
    let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
    let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ **1**;
    let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
    let e0 = (*w0*,*z0*);
    let e1 = (*w1*,*z1*);
    out ← $\mathcal{A}3$ *e0 e1 s′*;
    return-spmf ((), *out*)}
      apply(*simp add*: *P2-ideal-model-end-def True funct-OT-12-def*)
      using *order-gt-0 order-gt-1-gen-not-1 True l-neq-1* **by** *auto*

also have ... = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = **g** [^] (*r* ∗ *u0* + *v0*);
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] *u0*;
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}$*3 e0 e1 s′*;
return-spmf ((), *out*)}
by(*simp add*: *z0-rewrite w0-rewrite*)
also have ... = *do* {
u0 ← *map-spmf* (λ *u0*. ((*order* $\mathcal{G}$ − *s*) + *u0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform* (*order* $\mathcal{G}$));
v0 ← *map-spmf* (λ *v0*. (*r* ∗ *s* + *v0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform* (*order* $\mathcal{G}$));
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = **g** [^] (*r* ∗ *u0* + *v0*);
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*u0* + *s*);
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}$*3 e0 e1 s′*;
return-spmf ((), *out*)}
apply(*simp add*: *bind-map-spmf o-def Let-def cong*: *bind-spmf-cong-simp*)
using *P2-e0-rewrite assms s-lt assms* **by** *presburger*
also have ... = *do* {
u0 ← *map-spmf* (λ *u0*. ((*order* $\mathcal{G}$ − *s*) + *u0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform* (*order* $\mathcal{G}$));
v0 ← *map-spmf* (λ *v0*. (*r* ∗ *s* + *v0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform* (*order* $\mathcal{G}$));
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = **g** [^] (*r* ∗ *u0* + *v0*);
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] *u0* ⊗ *x0*;
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}$*3 e0 e1 s′*;
return-spmf ((), *out*)}
by(*simp add*: *z0-rewrite′ s*)
also have ... = *do* {
u0 ← *map-spmf* (λ *u0*. ((*order* $\mathcal{G}$ − *s*) + *u0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform*

$(order\ \mathcal{G}))$;
  $v0 \leftarrow map\text{-}spmf\ (\lambda\ v0.\ (r * s + v0)\ mod\ (order\ \mathcal{G}))\ (sample\text{-}uniform\ (order\ \mathcal{G}))$;
  $u1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
  $v1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
  $let\ w0 = (\mathbf{g}\ [\urcorner]\ (r{::}nat))\ [\urcorner]\ u0 \otimes \mathbf{g}\ [\urcorner]\ v0$;
  $let\ w1 = (\mathbf{g}\ [\urcorner]\ (r{::}nat))\ [\urcorner]\ u1 \otimes \mathbf{g}\ [\urcorner]\ v1$;
  $let\ z0 = b0\ [\urcorner]\ u0 \otimes h0\ [\urcorner]\ v0 \otimes x0$;
  $let\ z1 = (b1 \otimes inv\ \mathbf{g})\ [\urcorner]\ u1 \otimes h1\ [\urcorner]\ v1 \otimes x1$;
  $let\ e0 = (w0,z0)$;
  $let\ e1 = (w1,z1)$;
  $out \leftarrow \mathcal{A}3\ e0\ e1\ s'$;
  $return\text{-}spmf\ ((),\ out)\}$
   by(*simp add*: *w0-rewrite z0-rewrite''*)
  also have ... = *do* {
  $u0 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
  $v0 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
  $u1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
  $v1 \leftarrow sample\text{-}uniform\ (order\ \mathcal{G})$;
  $let\ w0 = (\mathbf{g}\ [\urcorner]\ (r{::}nat))\ [\urcorner]\ u0 \otimes \mathbf{g}\ [\urcorner]\ v0$;
  $let\ w1 = (\mathbf{g}\ [\urcorner]\ (r{::}nat))\ [\urcorner]\ u1 \otimes \mathbf{g}\ [\urcorner]\ v1$;
  $let\ z0 = b0\ [\urcorner]\ u0 \otimes h0\ [\urcorner]\ v0 \otimes x0$;
  $let\ z1 = (b1 \otimes inv\ \mathbf{g})\ [\urcorner]\ u1 \otimes h1\ [\urcorner]\ v1 \otimes x1$;
  $let\ e0 = (w0,z0)$;
  $let\ e1 = (w1,z1)$;
  $out \leftarrow \mathcal{A}3\ e0\ e1\ s'$;
  $return\text{-}spmf\ ((),\ out)\}$
   by(*simp add*: *samp-uni-plus-one-time-pad*)
  ultimately show *?thesis*
   by(*simp add*: *P2-real-model-end-def*)
 next
  case *False* — case 3
  have *b0-l*: $b0 = (b0 \otimes (inv\ (h0\ [\urcorner]\ r))) \otimes h0\ [\urcorner]\ r$
   by (*simp add*: *assert-in-carrier m-assoc*)
  have *b0-g-r*: $b0 = (b0 \otimes (inv\ (h0\ [\urcorner]\ r))) \otimes \mathbf{g}\ [\urcorner]\ (r * \alpha)$
   by (*metis* α *b0-l generator-closed mult.commute nat-pow-pow*)
  obtain $t :: nat$ **where** t: $\mathbf{g}\ [\urcorner]\ t = (b0 \otimes (inv\ (h0\ [\urcorner]\ r)))$ **and** *t-lt-order-g*: $t < order\ \mathcal{G}$
   by (*metis* (*full-types*) *mod-less-divisor order-gt-0 pow-generator-mod*
     assert-in-carrier cyclic-group.generatorE cyclic-group-axioms
      inv-closed m-closed nat-pow-closed)
  with *l-neq-1* **have** *t-neq-0*: $t \neq 0$ **using** *l-neq-1-exp-neq-0* **by** *simp*
  have *z0-rewrite*: $b0\ [\urcorner]\ u0 \otimes h0\ [\urcorner]\ v0 \otimes \mathbf{1} = \mathbf{g}\ [\urcorner]\ (r * \alpha * u0 + v0 * \alpha) \otimes ((b0 \otimes (inv\ (h0\ [\urcorner]\ r))))\ [\urcorner]\ u0$
   for *u0 v0*
  proof−
   from *b0-l* **have** $b0\ [\urcorner]\ u0 \otimes h0\ [\urcorner]\ v0 = ((b0 \otimes (inv\ (h0\ [\urcorner]\ r))) \otimes h0\ [\urcorner]\ r)\ [\urcorner]\ u0 \otimes h0\ [\urcorner]\ v0$ **by** *simp*
   hence $b0\ [\urcorner]\ u0 \otimes h0\ [\urcorner]\ v0 = ((b0 \otimes (inv\ (h0\ [\urcorner]\ r))))\ [\urcorner]\ u0 \otimes (h0\ [\urcorner]\ r)$

[^] *u0* ⊗ *h0* [^] *v0*
by (*simp add*: *assert-in-carrier pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*)
hence *b0* [^] *u0* ⊗ *h0* [^] *v0* = ((**g** [^] α) [^] *r*) [^] *u0* ⊗ (**g** [^] α) [^] *v0* ⊗ ((*b0* ⊗ (*inv* (*h0* [^] *r*)))) [^] *u0*
using *cyclic-group-assoc cyclic-group-commute assert-in-carrier* α **by** *simp*
hence *b0* [^] *u0* ⊗ *h0* [^] *v0* = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ ((*b0* ⊗ (*inv* (*h0* [^] *r*)))) [^] *u0*
by (*simp add*: *monoid.nat-pow-pow mult.commute nat-pow-mult*)
thus *?thesis*
by (*simp add*: *assert-in-carrier*)
qed
have *z0-rewrite′*: **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ ((*b0* ⊗ (*inv* (*h0* [^] *r*)))) [^] *u0* = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*t* ∗ *u0*)
for *u0 v0*
by (*metis generator-closed nat-pow-pow t*)
have *z0-rewrite″*: **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*t* ∗ *u0*) ⊗ **g** [^] *s* = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*
for *u0 v0*
using *assert-in-carrier s z0-rewrite z0-rewrite′* **by** *auto*
have *P2-ideal-model-end* (*x0*,*x1*) (*b0* ⊗ (*inv* (*h0* [^] *r*))) ((*h0*,*h1*,**g** [^] (*r*::*nat*),*b0*,*b1*),*s′*) *𝒜3* = *do* {
u0 ← *sample-uniform* (*order* 𝒢);
v0 ← *sample-uniform* (*order* 𝒢);
u1 ← *sample-uniform* (*order* 𝒢);
v1 ← *sample-uniform* (*order* 𝒢);
let w0 = **g** [^] (*r* ∗ *u0* + *v0*);
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ ((*b0* ⊗ (*inv* (*h0* [^] *r*)))) [^] *u0*;
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← *𝒜3 e0 e1 s′*;
return-spmf ((), *out*)}
by(*simp add*: *P2-ideal-model-end-def l-neq-1 funct-OT-12-def w0-rewrite z0-rewrite*)
also have ... = *do* {
u0 ← *sample-uniform* (*order* 𝒢);
v0 ← *sample-uniform* (*order* 𝒢);
u1 ← *sample-uniform* (*order* 𝒢);
v1 ← *sample-uniform* (*order* 𝒢);
let w0 = **g** [^] (*r* ∗ *u0* + *v0*);
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*t* ∗ *u0*);
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← *𝒜3 e0 e1 s′*;
return-spmf ((), *out*)}

by(*simp add*: *z0-rewrite′*)
also have ... = *do* {
u0 ← *map-spmf* (λ *u0*. (*order* $\mathcal{G}$ ∗ *order* $\mathcal{G}$ − (*s* ∗ ((*nat* (((*fst* (*bezw* *t* (*order* $\mathcal{G}$)))) *mod* (*order* $\mathcal{G}$))))) + *u0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform* (*order* $\mathcal{G}$));
v0 ← *map-spmf* (λ *v0*. (*r* ∗ *s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)) + *v0*) *mod* (*order* $\mathcal{G}$)) (*sample-uniform* (*order* $\mathcal{G}$));
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let *w0* = **g** [^] (*r* ∗ *u0* + *v0*);
let *w1* = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let *z0* = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)))));
let *z1* = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let *e0* = (*w0*,*z0*);
let *e1* = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0* *e1* *s′*;
return-spmf ((), *out*)}
by(*simp add*: *bind-map-spmf* *o-def* *Let-def* *s-lt* *P2-case-l-new-1-gt-e0-rewrite* *cong*: *bind-spmf-cong-simp*)
also have ... = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let *w0* = **g** [^] (*r* ∗ *u0* + *v0*);
let *w1* = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let *z0* = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*t* ∗ (*u0* + (*s* ∗ (*nat* ((*fst* (*bezw* *t* (*order* $\mathcal{G}$))) *mod* *order* $\mathcal{G}$)))));
let *z1* = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let *e0* = (*w0*,*z0*);
let *e1* = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0* *e1* *s′*;
return-spmf ((), *out*)}
by(*simp add*: *samp-uni-plus-one-time-pad*)
also have ... = *do* {
u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let *w0* = **g** [^] (*r* ∗ *u0* + *v0*);
let *w1* = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let *z0* = **g** [^] (*r* ∗ α ∗ *u0* + *v0* ∗ α) ⊗ **g** [^] (*t* ∗ *u0*) ⊗ **g** [^] *s*;
let *z1* = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let *e0* = (*w0*,*z0*);
let *e1* = (*w1*,*z1*);
out ← $\mathcal{A}3$ *e0* *e1* *s′*;
return-spmf ((), *out*)}
by(*simp add*: *P2-case-l-neq-1-gt-x0-rewrite* *t-lt-order-g* *t-neq-0* *cyclic-group-assoc*)
also have ... = *do* {

u0 ← *sample-uniform* (*order* $\mathcal{G}$);
v0 ← *sample-uniform* (*order* $\mathcal{G}$);
u1 ← *sample-uniform* (*order* $\mathcal{G}$);
v1 ← *sample-uniform* (*order* $\mathcal{G}$);
let w0 = (**g** [^] (*r*::*nat*)) [^] *u0* ⊗ **g** [^] *v0*;
let w1 = (**g** [^] (*r*::*nat*)) [^] *u1* ⊗ **g** [^] *v1*;
let z0 = *b0* [^] *u0* ⊗ *h0* [^] *v0* ⊗ *x0*;
let z1 = (*b1* ⊗ *inv* **g**) [^] *u1* ⊗ *h1* [^] *v1* ⊗ *x1*;
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
out ← $\mathcal{A}$*3 e0 e1 s′*;
return-spmf ((), *out*)}
by(*simp add*: *w0-rewrite z0-rewrite′′*)
ultimately show *?thesis*
by(*simp add*: *P2-real-model-end-def*)
qed
qed

lemma *P2-ideal-real-eq*:
assumes *x1-in-carrier*: *x1* ∈ *carrier* $\mathcal{G}$
and *x0-in-carrier*: *x0* ∈ *carrier* $\mathcal{G}$
shows *P2-real-model* (*x0*,*x1*) σ *z* $\mathcal{A}$ = *P2-ideal-model* (*x0*,*x1*) σ *z* $\mathcal{A}$
proof−
have *P2-real-model′* (*x0*, *x1*) σ *z* $\mathcal{A}$ = *P2-ideal-model′* (*x0*, *x1*) σ *z* $\mathcal{A}$
proof−
have *1*:*do* {
let ($\mathcal{A}$*1*, $\mathcal{A}$*2*, $\mathcal{A}$*3*) = $\mathcal{A}$;
((*h0*,*h1*,*a*,*b0*,*b1*),*s*) ← $\mathcal{A}$*1* σ *z*;
- :: *unit* ← *assert-spmf* (*h0* ∈ *carrier* $\mathcal{G}$ ∧ *h1* ∈ *carrier* $\mathcal{G}$ ∧ *a* ∈ *carrier* $\mathcal{G}$ ∧ *b0* ∈ *carrier* $\mathcal{G}$ ∧ *b1* ∈ *carrier* $\mathcal{G}$);
(((*in1*, *in2*, *in3*), *r*),*s′*) ← $\mathcal{A}$*2* (*h0*,*h1*,*a*,*b0*,*b1*) *s*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
(*out-zk-funct*, -) ← *funct-DH-ZK* (*h*,*a*,*b*) ((*in1*, *in2*, *in3*), *r*);
- :: *unit* ← *assert-spmf out-zk-funct*;
let l = *b0* ⊗ (*inv* (*h0* [^] *r*));
P2-ideal-model-end (*x0*,*x1*) *l* ((*h0*,*h1*,*a*,*b0*,*b1*),*s′*) $\mathcal{A}$*3*} = *P2-ideal-model′* (*x0*,*x1*) σ *z* $\mathcal{A}$
unfolding *P2-ideal-model′-def* **by** *simp*
have *P2-real-model′* (*x0*, *x1*) σ *z* $\mathcal{A}$ = *do* {
let ($\mathcal{A}$*1*, $\mathcal{A}$*2*, $\mathcal{A}$*3*) = $\mathcal{A}$;
((*h0*,*h1*,*a*,*b0*,*b1*),*s*) ← $\mathcal{A}$*1* σ *z*;
- :: *unit* ← *assert-spmf* (*h0* ∈ *carrier* $\mathcal{G}$ ∧ *h1* ∈ *carrier* $\mathcal{G}$ ∧ *a* ∈ *carrier* $\mathcal{G}$ ∧ *b0* ∈ *carrier* $\mathcal{G}$ ∧ *b1* ∈ *carrier* $\mathcal{G}$);
(((*in1*, *in2*, *in3*), *r*),*s′*) ← $\mathcal{A}$*2* (*h0*,*h1*,*a*,*b0*,*b1*) *s*;
let (*h*,*a*,*b*) = (*h0* ⊗ *inv h1*, *a*, *b0* ⊗ *inv b1*);
(*out-zk-funct*, -) ← *funct-DH-ZK* (*h*,*a*,*b*) ((*in1*, *in2*, *in3*), *r*);
- :: *unit* ← *assert-spmf out-zk-funct*;
P2-real-model-end (*x0*, *x1*) ((*h0*,*h1*,*a*,*b0*,*b1*),*s′*) $\mathcal{A}$*3*}
by(*simp add*: *P2-real-model′-def*)

```
    also have ... = do {
    let (𝒜1, 𝒜2, 𝒜3) = 𝒜;
    ((h0,h1,a,b0,b1),s) ← 𝒜1 σ z;
    - :: unit ← assert-spmf (h0 ∈ carrier 𝒢 ∧ h1 ∈ carrier 𝒢  ∧ a ∈ carrier 𝒢 ∧
b0 ∈ carrier 𝒢 ∧ b1 ∈ carrier 𝒢);
    (((in1, in2, in3), r),s′) ← 𝒜2 (h0,h1,a,b0,b1) s;
    let (h,a,b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
    (out-zk-funct, -) ← funct-DH-ZK (h,a,b) ((in1, in2, in3), r);
    - :: unit ← assert-spmf out-zk-funct;
    let l = b0 ⊗ (inv (h0 [^] r));
    P2-ideal-model-end (x0,x1) l ((h0,h1,a,b0,b1),s′) 𝒜3}
      by(simp add: P2-ideal-real-end-eq assms cong: bind-spmf-cong-simp)
   ultimately show ?thesis by(simp add: P2-real-model′-def P2-ideal-model′-def)
  qed
  thus ?thesis by(simp add: P2-ideal-model-rewrite P2-real-model-rewrite)
qed

lemma malicious-sec-P2:
  assumes x1-in-carrier: x1 ∈ carrier 𝒢
    and x0-in-carrier: x0 ∈ carrier 𝒢
  shows mal-def.perfect-sec-P2 (x0,x1) σ z (P2-S1, P2-S2) 𝒜
  unfolding malicious-base.perfect-sec-P2-def
  by (simp add: P2-ideal-real-eq P2-ideal-view-unfold assms)


lemma correct:
  assumes x0 ∈ carrier 𝒢
    and x1 ∈ carrier 𝒢
  shows funct-OT-12 (x0, x1) σ = protocol-ot (x0,x1) σ
proof−
  have σ-eq-0-output-correct:
    ((g [^] α0) [^] r) [^] u0 ⊗ (g [^] α0) [^] v0 ⊗ x0 ⊗
                      inv (((g [^] r) [^] u0 ⊗ g [^] v0) [^] α0) = x0
    (is ?lhs = ?rhs)
    for α0 r u0 v0 :: nat
  proof−
    have mult-com: r ∗ u0 ∗ α0 = α0 ∗ r ∗ u0 by simp
    have in-carrier1: ((g [^] (r ∗ u0 ∗ α0))) ⊗ (g [^] (v0 ∗ α0)) ∈ carrier 𝒢 by
simp
    have in-carrier2: inv ((g [^] (r ∗ u0 ∗ α0))) ⊗ (g [^] (v0 ∗ α0)) ∈ carrier 𝒢
by simp
    have ?lhs = ((g [^] (α0 ∗ r ∗ u0))) ⊗ (g [^] (α0 ∗ v0)) ⊗ x0 ⊗
                       inv (((g [^] (r ∗ u0 ∗ α0)) ⊗ g [^] (v0 ∗ α0)))
     by (simp add: nat-pow-pow pow-mult-distrib cyclic-group-commute monoid-comm-monoidI)
    also have ... = (((g [^] (r ∗ u0 ∗ α0))) ⊗ (g [^] (v0 ∗ α0))) ⊗ x0 ⊗
                       (inv (((g [^] (r ∗ u0 ∗ α0)) ⊗ g [^] (v0 ∗ α0))))
      using mult.commute mult.assoc mult-com
      by (metis (no-types) mult.commute)
```

also have ... = $x0 \otimes (((\mathbf{g} [\widehat{\ }] (r * u0 * \alpha0))) \otimes (\mathbf{g} [\widehat{\ }] (v0 * \alpha0))) \otimes$
$(inv (((\mathbf{g} [\widehat{\ }] (r * u0 * \alpha0)) \otimes \mathbf{g} [\widehat{\ }] (v0 * \alpha0))))$
using *cyclic-group-commute in-carrier1 assms* **by** *simp*
also have ... = $x0 \otimes ((((\mathbf{g} [\widehat{\ }] (r * u0 * \alpha0))) \otimes (\mathbf{g} [\widehat{\ }] (v0 * \alpha0))) \otimes$
$(inv (((\mathbf{g} [\widehat{\ }] (r * u0 * \alpha0)) \otimes \mathbf{g} [\widehat{\ }] (v0 * \alpha0)))))$
using *cyclic-group-assoc in-carrier1 in-carrier2 assms* **by** *auto*
ultimately show *?thesis* **using** *assms* **by** *simp*
qed
have *σ-eq-1-output-correct*:
$((\mathbf{g} [\widehat{\ }] \alpha1) [\widehat{\ }] r \otimes \mathbf{g} \otimes inv\ \mathbf{g}) [\widehat{\ }] u1 \otimes (\mathbf{g} [\widehat{\ }] \alpha1) [\widehat{\ }] v1 \otimes x1 \otimes$
$inv (((\mathbf{g} [\widehat{\ }] r) [\widehat{\ }] u1 \otimes \mathbf{g} [\widehat{\ }] v1) [\widehat{\ }] \alpha1) = x1$
(**is** *?lhs* = *?rhs*)
for $\alpha1\ r\ u1\ v1$ :: *nat*
proof−
have *com1*: $\alpha1 * r * u1 = r * u1 * \alpha1\ v1 * \alpha1 = \alpha1 * v1$ **by** *simp+*
have *in-carrier1*: $(\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes (\mathbf{g} [\widehat{\ }] (v1 * \alpha1)) \in carrier\ \mathcal{G}$ **by**
simp
have *in-carrier2*: $inv ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes (\mathbf{g} [\widehat{\ }] (v1 * \alpha1))) \in carrier\ \mathcal{G}$
by *simp*
have *lhs*: *?lhs* = $((\mathbf{g} [\widehat{\ }] (\alpha1 * r)) \otimes \mathbf{g} \otimes inv\ \mathbf{g}) [\widehat{\ }] u1 \otimes (\mathbf{g} [\widehat{\ }] (\alpha1 * v1)) \otimes x1$
$\otimes$
$inv ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes \mathbf{g} [\widehat{\ }] (v1 * \alpha1))$
by (*simp add*: *nat-pow-pow pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*)
also have *lhs1*: ... = $(\mathbf{g} [\widehat{\ }] (\alpha1 * r)) [\widehat{\ }] u1 \otimes (\mathbf{g} [\widehat{\ }] (\alpha1 * v1)) \otimes x1 \otimes$
$inv ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes \mathbf{g} [\widehat{\ }] (v1 * \alpha1))$
by (*simp add*: *cyclic-group-assoc*)
also have *lhs2*: ... = $(\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes (\mathbf{g} [\widehat{\ }] (v1 * \alpha1)) \otimes x1 \otimes$
$inv ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes \mathbf{g} [\widehat{\ }] (v1 * \alpha1))$
by (*simp add*: *nat-pow-pow pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*
com1)
also have ... = $(((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes (\mathbf{g} [\widehat{\ }] (v1 * \alpha1))) \otimes x1) \otimes$
$inv ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes \mathbf{g} [\widehat{\ }] (v1 * \alpha1))$
using *in-carrier1 in-carrier2 assms cyclic-group-assoc* **by** *blast*
also have ... = $(x1 \otimes ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes (\mathbf{g} [\widehat{\ }] (v1 * \alpha1)))) \otimes$
$inv ((\mathbf{g} [\widehat{\ }] (r * u1 * \alpha1)) \otimes \mathbf{g} [\widehat{\ }] (v1 * \alpha1))$
using *in-carrier1 assms cyclic-group-commute* **by** *simp*
ultimately show *?thesis*
using *cyclic-group-assoc assms in-carrier1 in-carrier1 assms cyclic-group-commute*
lhs1 lhs2 lhs **by** *force*
qed
show *?thesis*
unfolding *funct-OT-12-def protocol-ot-def Let-def*
by(*cases σ*; *auto simp add*: *assms σ-eq-1-output-correct σ-eq-0-output-correct*
bind-spmf-const
lossless-sample-uniform-units order-gt-0 P1-assert-correct1 P1-assert-correct2
lossless-weight-spmfD)
qed

lemma *correctness*:

assumes *x0* ∈ *carrier* $\mathcal{G}$
and *x1* ∈ *carrier* $\mathcal{G}$
shows *mal-def.correct* (*x0*,*x1*) σ
unfolding *mal-def.correct-def*
by(*simp add*: *correct assms*)

end

locale *OT-asymp* =
fixes $\mathcal{G}$ :: *nat* ⇒ *'grp cyclic-group*
assumes *ot*: $\bigwedge\eta$. *ot* ($\mathcal{G}$ η)
begin

sublocale *ot* $\mathcal{G}$ *n* **for** *n* **using** *ot* **by** *simp*

lemma *correctness-asym*:
assumes *x0* ∈ *carrier* ($\mathcal{G}$ *n*)
and *x1* ∈ *carrier* ($\mathcal{G}$ *n*)
shows *mal-def.correct n* (*x0*,*x1*) σ
using *assms correctness* **by** *simp*

lemma *P1-security-asym*:
negligible (λ *n*. *mal-def.adv-P1 n M* σ *z* (*P1-S1 n*, *P1-S2*) $\mathcal{A}$ *D*)
if *neg1*: *negligible* (λ *n*. *ddh.advantage n* (*P1-DDH-mal-adv-*σ*-true n M z* $\mathcal{A}$ *D*))
and *neg2*: *negligible* (λ *n*. *ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-*σ*-true n M z* $\mathcal{A}$ *D*)))
and *neg3*: *negligible* (λ *n*. *ddh.advantage n* (*P1-DDH-mal-adv-*σ*-false n M z* $\mathcal{A}$ *D*))
and *neg4*: *negligible* (λ *n*. *ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-*σ*-false n M z* $\mathcal{A}$ *D*)))
proof−
have *neg-add1*: *negligible* (λ *n*. *ddh.advantage n* (*P1-DDH-mal-adv-*σ*-true n M z* $\mathcal{A}$ *D*)
\+ *ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-*σ*-true n M z* $\mathcal{A}$ *D*)))
and *neg-add2*: *negligible* (λ *n*. *ddh.advantage n* (*P1-DDH-mal-adv-*σ*-false n M z* $\mathcal{A}$ *D*)
\+ *ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-*σ*-false n M z* $\mathcal{A}$ *D*)))

using *neg1 neg2 neg3 neg4 negligible-plus* **by**(*blast*)+
show *?thesis*
proof(*cases* σ)
case *True*
have *bound-mod*: |*mal-def.adv-P1 n M* σ *z* (*P1-S1 n*, *P1-S2*) $\mathcal{A}$ *D*|
≤ *ddh.advantage n* (*P1-DDH-mal-adv-*σ*-true n M z* $\mathcal{A}$ *D*)
\+ *ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-*σ*-true n M z* $\mathcal{A}$ *D*)) **for** *n*
by (*metis* (*no-types*) *True abs-idempotent P1-adv-real-ideal-model-def P1-advantages-eq P1-real-ideal-DDH-advantage-true-bound*)
then show *?thesis*

using *P1-real-ideal-DDH-advantage-true-bound that bound-mod that negligible-le neg-add1* **by** *presburger*
next
case *False*
have *bound-mod*: $|$*mal-def.adv-P1 n M σ z* (*P1-S1 n*, *P1-S2*) $\mathcal{A}$ *D*$|$
$\leq$ *ddh.advantage n* (*P1-DDH-mal-adv-σ-false n M z* $\mathcal{A}$ *D*)
\+ *ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-σ-false n M z* $\mathcal{A}$ *D*)) **for** *n*
proof −
have $|$*spmf* (*P1-real-model n M σ z* $\mathcal{A} \ggg$ *D*) *True* − *spmf* (*P1-ideal-model n M σ z* $\mathcal{A} \ggg$ *D*) *True*$|$
$\leq$ *local.ddh.advantage n* (*P1-DDH-mal-adv-σ-false n M z* $\mathcal{A}$ *D*)
\+ *local.ddh.advantage n* (*ddh.DDH-*$\mathcal{A}'$ *n* (*P1-DDH-mal-adv-σ-false n M z* $\mathcal{A}$ *D*))
by (*metis* (*no-types*) *False P1-adv-real-ideal-model-def P1-advantages-eq P1-real-ideal-DDH-advantage-false-bound*)
then show *?thesis*
by (*simp add*: *P1-adv-real-ideal-model-def P1-advantages-eq*)
qed
then show *?thesis* **using** *P1-real-ideal-DDH-advantage-false-bound bound-mod that negligible-le neg-add2* **by** *presburger*
qed
qed

lemma *P2-security-asym*:
assumes *x1-in-carrier*: *x1* $\in$ *carrier* ($\mathcal{G}$ *n*)
and *x0-in-carrier*: *x0* $\in$ *carrier* ($\mathcal{G}$ *n*)
shows *mal-def.perfect-sec-P2 n* (*x0*,*x1*) *σ z* (*P2-S1 n*, *P2-S2 n*) $\mathcal{A}$
using *assms malicious-sec-P2* **by** *fast*

end

end

References

[1] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *IACR Cryptology ePrint Archive*, 2017:753, 2017.

[2] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[3] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.

[4] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.

[5] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[6] A. Lochbihler. CryptHOL. *Archive of Formal Proofs*, 2017. http://isa-afp.org/entries/CryptHOL.shtml, Formal proof development.

[7] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457. ACM/SIAM, 2001.

[8] A. C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.