Multi-Party Computation

David Aspinall and David Butler

May 26, 2024

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*⟩

lemma *pow-generator-mod*: **g** [^] (*k mod order G*) = **g** [^] *k*
⟨*proof*⟩

lemma *int-nat-pow*:
assumes *a* ≥ *0*
shows (**g** [^] (*int* (*a* ::*nat*))) [^] (*b*::*int*) = **g** [^] (*a*∗*b*)
⟨*proof*⟩

lemma *pow-generator-mod-int*: **g** [^] ((*k* :: *int*) *mod order G*) = **g** [^] *k*
⟨*proof*⟩

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*⟩

lemma *pow-generator-eq-iff-cong*:
finite (*carrier G*) ⟹ **g** [^] *x* = **g** [^] *y* ⟷ [*x* = *y*] (*mod order G*)
⟨*proof*⟩

lemma *cyclic-group-commute*:
assumes *a* ∈ *carrier G b* ∈ *carrier G*
shows *a* ⊗ *b* = *b* ⊗ *a*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *l-cancel-inv*:
assumes *h* ∈ *carrier G*
shows (**g** [^] (*a* :: *nat*) ⊗ *inv* (**g** [^] *a*)) ⊗ *h* = *h*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *inverse-split*:

assumes $a \in$ *carrier G* **and** $b \in$ *carrier G*
shows *inv* $(a \otimes b) =$ *inv* $a \otimes$ *inv* b
⟨*proof*⟩

lemma *inverse-pow-pow*:
assumes $a \in$ *carrier G*
shows *inv* (*a* [^] (*r*::*nat*)) = (*inv a*) [^] *r*
⟨*proof*⟩

lemma *l-neq-1-exp-neq-0*:
assumes $l \in$ *carrier G*
and $l \neq$ **1**
and $l =$ **g** [^] (*t*::*nat*)
shows $t \neq 0$
⟨*proof*⟩

lemma *order-gt-1-gen-not-1*:
assumes *order G* > *1*
shows **g** ≠ **1**
⟨*proof*⟩

lemma *power-swap*: ((**g** [^] (*α0*::*nat*)) [^] (*r*::*nat*)) = ((**g** [^] *r*) [^] *α0*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *inverse*:
assumes *gcd x* (*q*::*nat*) = *1*
and *q* > *0*
shows [*x* ∗ (*fst* (*bezw x q*)) = *1*] (*mod q*)
⟨*proof*⟩

lemma *prod-not-prime*:
assumes *prime* (*x*::*nat*)
and *prime y*
and *x* > *2*
and *y* > *2*

shows ¬ *prime* ((*x*−*1*)∗(*y*−*1*))
⟨*proof*⟩

lemma *ex-inverse*:
assumes *coprime*: *coprime* (*e* :: *nat*) ((*P*−*1*)∗(*Q*−*1*))
and *prime P*
and *prime Q*
and *P* ≠ *Q*
shows ∃ *d*. [*e*∗*d* = *1*] (*mod* (*P*−*1*)) ∧ *d* ≠ *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *fermat-little*:
assumes *prime* (*P* :: *nat*)
shows [*x*^*P* = *x*] (*mod P*)
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *lossless-sample-uniform-units*:
assumes *q* > *1*
shows *lossless-spmf* (*sample-uniform-units q*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *inj-uni-samp-add*: *inj-on* (λ(*b* :: *nat*). (*y* + *b*) *mod q*) {..<*q*}
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *samp-uni-plus-one-time-pad*:
shows *map-spmf* (λ*b*. (*y* + *b*) *mod q*) (*sample-uniform q*) = (*sample-uniform q*)
⟨*proof*⟩

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*⟩

lemma *inj-on-mult*:
assumes *coprime*: *coprime x* (*q*::*nat*)
shows *inj-on* (λ *b*. *x*∗*b mod q*) {..<*q*}
⟨*proof*⟩

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*}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*})
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

lemma *inj-on-add-mult*:
  assumes *coprime*: *coprime x* (*q*::*nat*)
  shows *inj-on* (λ *b*. (*y* + *x*∗*b*) *mod q*) {..<*q*}
  ⟨*proof*⟩

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*}
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*⟩

lemma *inj-on-minus*: *inj-on* (λ(*b* :: *nat*). (*y* + (*q* − *b*)) *mod q*) {..<*q*}
  ⟨*proof*⟩

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*}
  ⟨*proof*⟩

lemma *samp-uni-minus-one-time-pad*:
 shows *map-spmf*(λ *b*. (*y* + (*q* − *b*)) *mod q*) (*sample-uniform q*) = (*sample-uniform q*)
  ⟨*proof*⟩

lemma *not-coin-flip*: *map-spmf* (λ *a*. ¬ *a*) *coin-spmf* = *coin-spmf*
⟨*proof*⟩

lemma *xor-uni-samp*: *map-spmf*(λ *b*. *y* ⊕ *b*) (*coin-spmf*) = *map-spmf*(λ *b*. *b*) (*coin-spmf*)
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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* ≡ (*R2 m1 m2* = *funct m1 m2* ⤜ (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* ⇒ *'msg2* ⇒ *'view1 adversary-det* ⇒ *bool spmf*
  where *P1-game-alt m1 m2 D* = *do* {
    b ← *coin-spmf*;
    (*out1*, *out2*) ← *funct m1 m2*;
    rview :: *'view1* ← *R1 m1 m2*;
    sview :: *'view1* ← *S1 m1 out1*;
    b' ← *D* (*if b then rview else sview*);
    return-spmf (*b* = *b'*)}

definition *adv-P1-game* :: *'msg1* ⇒ *'msg2* ⇒ *'view1 adversary-det* ⇒ *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*: ∀ *view*. *lossless-spmf* ((*D*:: *'view1 adversary-det*) *view*)
  shows *adv-P1-game m1 m2 D* = *adv-P1 m1 m2 D*
  including *monad-normalisation*
⟨*proof*⟩

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*⟩

end

2.1.2 Security definitions for non deterministic functionalities

locale *sim-non-det-def* =
  fixes *R1* :: *'msg1* ⇒ *'msg2* ⇒ (*'view1* × (*'out1* × *'out2*)) *spmf*
    and *S1* :: *'msg1* ⇒ *'out1* ⇒ *'view1 spmf*
    and *Out1* :: *'msg1* ⇒ *'msg2* ⇒ *'out1* ⇒ (*'out1* × *'out2*) *spmf* — takes the input of the other party so can form the outputs of parties
    and *R2* :: *'msg1* ⇒ *'msg2* ⇒ (*'view2* × (*'out1* × *'out2*)) *spmf*

and *S2* :: *'msg2* ⇒ *'out2* ⇒ *'view2 spmf*
and *Out2* :: *'msg2* ⇒ *'msg1* ⇒ *'out2* ⇒ (*'out1* × *'out2*) *spmf*
and *funct* :: *'msg1* ⇒ *'msg2* ⇒ (*'out1* × *'out2*) *spmf*
begin

type-synonym (*'view'*, *'out1'*, *'out2'*) *adversary-non-det* = (*'view'* × (*'out1'* × *'out2'*)) ⇒ *bool spmf*

definition *Ideal1* :: *'msg1* ⇒ *'msg2* ⇒ *'out1* ⇒ (*'view1* × (*'out1* × *'out2*)) *spmf*
where *Ideal1 m1 m2 out1* = *do* {
view1 :: *'view1* ← *S1 m1 out1*;
out1 ← *Out1 m1 m2 out1*;
return-spmf (*view1*, *out1*)}

definition *Ideal2* :: *'msg2* ⇒ *'msg1* ⇒ *'out2* ⇒ (*'view2* × (*'out1* × *'out2*)) *spmf*
where *Ideal2 m2 m1 out2* = *do* {
view2 :: *'view2* ← *S2 m2 out2*;
out2 ← *Out2 m2 m1 out2*;
return-spmf (*view2*, *out2*)}

definition *adv-P1* :: *'msg1* ⇒ *'msg2* ⇒ (*'view1*, *'out1*, *'out2*) *adversary-non-det* ⇒ *real*
where *adv-P1 m1 m2 D* ≡ |(*spmf* (*R1 m1 m2* ⤜ (λ *view*. *D view*)) *True*) − *spmf* (*funct m1 m2* ⤜ (λ (*o1*, *o2*). *Ideal1 m1 m2 o1* ⤜ (λ *view*. *D view*))) *True*|

definition *perfect-sec-P1 m1 m2* ≡ (*R1 m1 m2* = *funct m1 m2* ⤜ (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)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* = $'index \Rightarrow bool \Rightarrow bool \Rightarrow ('index,'range)$ *dist2* $\Rightarrow$ $'range \Rightarrow bool$ *spmf*

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

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

lemma *lossless-S*: $y \in$ *set-spmf* $I \longrightarrow$ *lossless-spmf* (S (*fst* y))
  ⟨*proof*⟩

lemma *set-spmf-S* [*simp*]: $y \in$ *set-spmf* $I \longrightarrow$ *set-spmf* (S (*fst* y)) = *range* (*fst* y)

  ⟨*proof*⟩

lemma *f-inj-on*: $y \in$ *set-spmf* $I \longrightarrow$ *inj-on* (F (*fst* y)) (*range* (*fst* y))
  ⟨*proof*⟩

lemma *range-f*: $y \in$ *set-spmf* $I \longrightarrow x \in$ *range* (*fst* y) $\longrightarrow F$ (*fst* y) $x \in$ *range* (*fst* y)
  ⟨*proof*⟩

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
  ⟨*proof*⟩

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
  ⟨*proof*⟩

lemma *B-F-inv-rewrite*: $(B\ \alpha\ (F_{inv}\ \alpha\ \tau\ y_\sigma') = (B\ \alpha\ (F_{inv}\ \alpha\ \tau\ y_\sigma') = m1)) = m1$
  ⟨*proof*⟩

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*⟩

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

definition *HCP-game* :: ('*index*,'*range*) *advP2* ⇒ *bool* ⇒ *bool* ⇒ ('*index*,'*range*) *dist2* ⇒ *bool spmf*
where *HCP-game A* = (λ σ b_σ *D*. *do* {
(α, τ) ← *I*;
x ← *S* α;
b' ← *A* α σ b_σ *D x*;
let b = *B* α (F_{inv} α τ *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*)
⟨*proof*⟩

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

lemma *if-else-True* [*simp*]: (*if b then x else True*) $\longleftrightarrow$ ($b \longrightarrow x$)
⟨*proof*⟩

lemma *inj-on-Not* [*simp*]: *inj-on Not A*
⟨*proof*⟩

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_σ') = $input_1$;
(α :: *'index*, τ :: *'trap*) ← *I*;
x_σ :: *'range* ← *etp.S* α;
y_σ' :: *'range* ← *etp.S* α;
let (y_σ :: *'range*) = *F* α x_σ;
let (x_σ :: *'range*) = F_{inv} α τ y_σ;
let (x_σ' :: *'range*) = F_{inv} α τ y_σ';
let (β_σ :: *bool*) = *xor* (*B* α x_σ) b_σ;
let (β_σ' :: *bool*) = *xor* (*B* α x_σ') b_σ';
return-spmf ((), *if* σ *then xor* (*B* α x_σ') β_σ' *else xor* (*B* α x_σ) β_σ)}

lemma *correctness*: *protocol* (*m0*,*m1*) *c* = *funct-OT-12* (*m0*,*m1*) *c*
⟨*proof*⟩

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_σ' ← *etp.S* α;
let y_σ = *F* α x_σ;
return-spmf ((b_0, b_1), *if* σ *then* y_σ' *else* y_σ, *if* σ *then* y_σ *else* y_σ')}

lemma *lossless-R1*: *lossless-spmf* (*R1 msgs* σ)
⟨*proof*⟩

definition *S1* :: (*bool* × *bool*) ⇒ *unit* ⇒ *'range viewP1*
where *S1* == (λ $input_1$ (). *do* {
let (b_0, b_1) = $input_1$;

$(\alpha, \tau) \leftarrow I;$
$y_0 :: {}'range \leftarrow etp.S\ \alpha;$
$y_1 \leftarrow etp.S\ \alpha;$
$return\text{-}spmf\ ((b_0, b_1), y_0, y_1)\})$

lemma *lossless-S1*: *lossless-spmf* (*S1 msgs* ())
⟨*proof*⟩

Party 2 views

definition *R2* :: (*bool* × *bool*) ⇒ *bool* ⇒ *'index viewP2*
where *R2 msgs* σ = *do* {
$let\ (b0,b1) = msgs;$
$(\alpha, \tau) \leftarrow I;$
$x_\sigma \leftarrow etp.S\ \alpha;$
$y_\sigma{}' \leftarrow etp.S\ \alpha;$
$let\ y_\sigma = F\ \alpha\ x_\sigma;$
$let\ x_\sigma = F_{inv}\ \alpha\ \tau\ y_\sigma;$
$let\ x_\sigma{}' = F_{inv}\ \alpha\ \tau\ y_\sigma{}';$
$let\ \beta_\sigma = (B\ \alpha\ x_\sigma) \oplus (if\ \sigma\ then\ b1\ else\ b0)\ ;$
$let\ \beta_\sigma{}' = (B\ \alpha\ x_\sigma{}') \oplus (if\ \sigma\ then\ b0\ else\ b1);$
$return\text{-}spmf\ (\sigma, \alpha,(\beta_\sigma, \beta_\sigma{}'))\}$

lemma *lossless-R2*: *lossless-spmf* (*R2 msgs* σ)
⟨*proof*⟩

definition *S2* :: *bool* ⇒ *bool* ⇒ *'index viewP2*
where *S2* $\sigma\ b_\sigma$ = *do* {
$(\alpha, \tau) \leftarrow I;$
$x_\sigma \leftarrow etp.S\ \alpha;$
$y_\sigma{}' \leftarrow etp.S\ \alpha;$
$let\ x_\sigma{}' = F_{inv}\ \alpha\ \tau\ y_\sigma{}';$
$let\ \beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma;$
$let\ \beta_\sigma{}' = B\ \alpha\ x_\sigma{}';$
$return\text{-}spmf\ (\sigma, \alpha, (\beta_\sigma, \beta_\sigma{}'))\}$

lemma *lossless-S2*: *lossless-spmf* (*S2* $\sigma\ b_\sigma$)
⟨*proof*⟩

Security for Party 1

We have information theoretic security for Party 1.

lemma *P1-security*: $R1\ input_1\ \sigma = funct\text{-}OT\text{-}12\ x\ y \ggg= (\lambda\ (s1, s2).\ S1\ input_1\ s1)$
including *monad-normalisation*
⟨*proof*⟩

The adversary used in proof of security for party 2

definition $\mathcal{A}$ :: (*'index*, *'range*) *advP2*
where $\mathcal{A}\ \alpha\ \sigma\ b_\sigma\ D2\ x$ = *do* {

$\beta_\sigma{}' \leftarrow$ *coin-spmf*;
$x_\sigma \leftarrow$ *etp.S* α;
let $\beta_\sigma = (B\ \alpha\ x_\sigma) \oplus b_\sigma$;
$d \leftarrow D2(\sigma,\ \alpha,\ \beta_\sigma,\ \beta_\sigma{}')$;
return-spmf(*if d then* $\beta_\sigma{}'$ *else* $\neg\ \beta_\sigma{}'$)}

lemma *lossless-*$\mathcal{A}$:
assumes $\forall$ *view. lossless-spmf* (*D2 view*)
shows $y \in$ *set-spmf I* $\longrightarrow$ *lossless-spmf* ($\mathcal{A}$ (*fst y*) σ b_σ *D2 x*)
⟨*proof*⟩

lemma *assm-bound-funct-OT-12*:
assumes *etp.HCP-adv* $\mathcal{A}$ σ (*if* σ *then b1 else b0*) $D \leq$ *HCP-ad*
shows |*spmf* (*funct-OT-12* (*b0*,*b1*) $\sigma \gg\!=$ (λ (*out1*,*out2*).
etp.HCP-game $\mathcal{A}$ σ *out2 D*)) *True* $-$ *1/2*| $\leq$ *HCP-ad*
(**is** *?lhs* $\leq$ *HCP-ad*)
⟨*proof*⟩

lemma *assm-bound-funct-OT-12-collapse*:
assumes $\forall$ b_σ. *etp.HCP-adv* $\mathcal{A}$ σ b_σ $D \leq$ *HCP-ad*
shows |*spmf* (*funct-OT-12 m1* $\sigma \gg\!=$ (λ (*out1*,*out2*). *etp.HCP-game* $\mathcal{A}$ σ *out2 D*)) *True* $-$ *1/2*| $\leq$ *HCP-ad*
⟨*proof*⟩

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*) $\sigma \gg\!=$ (*D2* :: (*bool* $\times$ *'index* $\times$ *bool* $\times$ *bool*) $\Rightarrow$ *bool spmf*)) *True*
= *spmf* (*funct-OT-12* (*b0*,*b1*) $\sigma \gg\!=$ (λ (*out1*,*out2*). *S2* σ *out2* $\gg\!=$ *D2*)) *True*
⟨*proof*⟩

lemma *R2-S2-True*:
assumes ((*if* σ *then b0 else b1*) = *True*)
and *lossless-D*: $\forall$ *a. lossless-spmf* (*D2 a*)
shows |(*spmf* (*bind-spmf* (*R2* (*b0*,*b1*) σ) *D2*) *True*) $-$ *spmf* (*funct-OT-12* (*b0*,*b1*) $\sigma \gg\!=$ (λ (*out1*, *out2*). *S2* σ *out2* $\gg\!=$ (λ *view. D2 view*))) *True*|
= |*2*$*$((*spmf* (*etp.HCP-game* $\mathcal{A}$ σ (*if* σ *then b1 else b0*) *D2*) *True*) $-$ *1/2*)|
⟨*proof*⟩
including *monad-normalisation*
⟨*proof*⟩

lemma *P2-adv-bound*:
assumes *lossless-D*: $\forall$ *a. lossless-spmf* (*D2 a*)
shows |(*spmf* (*bind-spmf* (*R2* (*b0*,*b1*) σ) *D2*) *True*) $-$ *spmf* (*funct-OT-12* (*b0*,*b1*) $\sigma \gg\!=$ (λ (*out1*, *out2*). *S2* σ *out2* $\gg\!=$ (λ *view. D2 view*))) *True*|

$\leq |2*((spmf\ (etp.HCP\text{-}game\ \mathcal{A}\ \sigma\ (if\ \sigma\ then\ b1\ else\ b0)\ D2)$
$True) - 1/2)|$
⟨*proof*⟩

sublocale *OT-12*: *sim-det-def R1 S1 R2 S2 funct-OT-12 protocol*
⟨*proof*⟩

lemma *correct*: *OT-12.correctness m1 m2*
⟨*proof*⟩

lemma *P1-security-inf-the*: *OT-12.perfect-sec-P1 m1 m2*
⟨*proof*⟩

lemma *P2-security*:
assumes $\forall\ a.\ lossless\text{-}spmf\ (D\ a)$
and $\forall\ b_\sigma.\ etp.HCP\text{-}adv\ \mathcal{A}\ m2\ b_\sigma\ D \leq HCP\text{-}ad$
shows $OT\text{-}12.adv\text{-}P2\ m1\ m2\ D \leq 2 * HCP\text{-}ad$
⟨*proof*⟩

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*: $\bigwedge n.\ ETP\text{-}base\ (I\ n)\ domain\ range\ F\ F_{inv}$
begin

sublocale *ETP-base* (*I n*) *domain range*
⟨*proof*⟩

lemma *correct-asym*: *OT-12.correctness n m1 m2*
⟨*proof*⟩

lemma *P1-sec-asym*: *OT-12.perfect-sec-P1 n m1 m2*
⟨*proof*⟩

lemma *P2-sec-asym*:
assumes $\forall\ a.\ lossless\text{-}spmf\ (D\ a)$
and *HCP-adv-neg*: $negligible\ (\lambda\ n.\ etp\text{-}advantage\ n)$
and *etp-adv-bound*: $\forall\ b_\sigma\ n.\ etp.HCP\text{-}adv\ n\ \mathcal{A}\ m2\ b_\sigma\ D \leq etp\text{-}advantage\ n$
shows $negligible\ (\lambda\ n.\ OT\text{-}12.adv\text{-}P2\ n\ m1\ m2\ D)$
⟨*proof*⟩

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* ≠ {}
⟨*proof*⟩

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* ≠ {}
⟨*proof*⟩

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

lemma *sample-coprime-e-gt-1*:
  assumes $e \in$ *set-spmf* (*sample-coprime N*)
  shows $e > 1$
  ⟨*proof*⟩

lemma *lossless-sample-coprime*:
  assumes ¬ *prime N*
    and $N > 2$
  shows *lossless-spmf* (*sample-coprime N*)
⟨*proof*⟩

lemma *set-spmf-sample-coprime*:
  shows *set-spmf* (*sample-coprime N*) = $\{x.$ *coprime x N* $\wedge x > 1 \wedge x < N\}$
  ⟨*proof*⟩

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

lemma *lossless-sample-primes*:
  shows *lossless-spmf sample-primes*
    ⟨*proof*⟩

lemma *set-spmf-sample-primes*:
  shows *set-spmf sample-primes* $\subseteq \{x.$ *prime x* $\wedge x > 2\}$
  ⟨*proof*⟩

lemma *mem-samp-primes-gt-2*:
  shows $x \in$ *set-spmf sample-primes* $\Longrightarrow x > 2$
  ⟨*proof*⟩

lemma *mem-samp-primes-prime*:
  shows $x \in$ *set-spmf sample-primes* $\Longrightarrow$ *prime x*
  ⟨*proof*⟩

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*})
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

```
lemma lossless-sample-set-excl:
  assumes finite Q
    and card Q > 2
  shows lossless-spmf (sample-set-excl Q {P})
  ⟨proof⟩
```

```
lemma mem-samp-primes-excl-gt-2:
  shows x ∈ set-spmf (sample-set-excl prime-set {y}) ⟹ x > 2
  ⟨proof⟩
```

```
lemma mem-samp-primes-excl-prime :
  shows x ∈ set-spmf (sample-set-excl prime-set {y}) ⟹ prime x
  ⟨proof⟩
```

```
lemma sample-coprime-lem:
  assumes x ∈ set-spmf sample-primes
    and  y ∈ set-spmf (sample-set-excl prime-set {x})
  shows lossless-spmf (sample-coprime ((x − Suc 0) ∗ (y − Suc 0)))
⟨proof⟩
```

```
definition I :: (index × trap) spmf
  where I = do {
    P ← sample-primes;
    Q ← sample-set-excl prime-set {P};
    let N = P∗Q;
    let N' = (P−1)∗(Q−1);
    e ← sample-coprime N';
    let d = nat ((fst (bezw e N')) mod N');
    return-spmf ((N, e), d)}
```

```
lemma lossless-I: lossless-spmf I
  ⟨proof⟩
```

```
lemma set-spmf-I-N:
  assumes ((N,e),d) ∈ set-spmf I
  obtains P Q where N = P ∗ Q
    and P ≠ 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)))
  ⟨proof⟩
```

```
lemma set-spmf-I-e-d:
  ‹e > 1› ‹d > 1› if ‹((N, e), d) ∈ set-spmf I›
⟨proof⟩
```

```
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*)
  ⟨*proof*⟩

lemma *dom-eq-ran*: *domain index* = *range index*
  ⟨*proof*⟩

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*⟩

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*⟩

lemma *bij-betw1*:
  assumes ((*N*,*e*),*d*) ∈ *set-spmf I*
  shows *bij-betw* (*F* ((*N*), *e*)) (*range* ((*N*), *e*)) (*range* ((*N*), *e*))
⟨*proof*⟩

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 $((((x) \ \widehat{}\ e)\ mod\ (P{*}Q))\ \widehat{}\ d)\ mod\ (P{*}Q) = x\ mod\ (P{*}Q)$
⟨*proof*⟩

lemma *rsa-inv-set-spmf-I*:
assumes $((N,\ e),\ d) \in set\text{-}spmf\ I$
shows $((((x{::}nat)\ \widehat{}\ e)\ mod\ N)\ \widehat{}\ d)\ mod\ N = x\ mod\ N$
⟨*proof*⟩

sublocale *etp-rsa*: *etp I domain range F* F_{inv}
⟨*proof*⟩

sublocale *etp*: *ETP-base I domain range B F* F_{inv}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *P1-security-rsa*: *etp.OT-12.perfect-sec-P1 m1 m2*
⟨*proof*⟩

lemma *P2-security-rsa*:
assumes $\forall\ a.\ lossless\text{-}spmf\ (D\ a)$
and $\bigwedge b_\sigma.\ local.etp\text{-}rsa.HCP\text{-}adv\ etp.\mathcal{A}\ m2\ b_\sigma\ D \le HCP\text{-}ad$
shows $etp.OT\text{-}12.adv\text{-}P2\ m1\ m2\ D \le 2 * HCP\text{-}ad$
⟨*proof*⟩

end

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

sublocale *rsa-base* (*prime-set n*) *B*
⟨*proof*⟩

lemma *correctness-rsa-asymp*:
shows *etp.OT-12.correctness n m1 m2*
⟨*proof*⟩

lemma *P1-sec-asymp*: *etp.OT-12.perfect-sec-P1 n m1 m2*
⟨*proof*⟩

lemma *P2-sec-asym*:
assumes $\forall\ a.\ lossless\text{-}spmf\ (D\ a)$

and *HCP-adv-neg*: *negligible* (λ *n. hcp-advantage n*)
and *hcp-adv-bound*: $\forall$ b_σ *n. local.etp-rsa.HCP-adv n etp.*$\mathcal{A}$ *m2* b_σ *D* $\leq$ *hcp-advantage n*
shows *negligible* (λ *n. etp.OT-12.adv-P2 n m1 m2 D*)
⟨*proof*⟩

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*: $a < (order\ \mathcal{G}) \Longrightarrow a \neq 0 \Longrightarrow coprime\ a\ (order\ \mathcal{G})$
⟨*proof*⟩

lemma *weight-sample-uniform-units*: *weight-spmf* (*sample-uniform-units* (*order* $\mathcal{G}$)) = *1*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)*
⟨*proof*⟩

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)*
⟨*proof*⟩

fun *R1-inter-adversary* :: *′grp dist-adversary ⇒ (′grp × ′grp) ⇒ ′grp ⇒ ′grp ⇒ ′grp ⇒ bool spmf*

where *R1-inter-adversary* 𝒜 *msgs* α β γ = *do* {
c ← *sample-uniform* (*order* 𝒢);
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* 𝒜 *msgs* α β γ = *do* {
c ← *sample-uniform* (*order* 𝒢);
𝒜 (*msgs*, α, β, **g** [^] *c*, γ)}

sublocale *ddh*: *ddh* 𝒢 ⟨*proof*⟩

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* 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 z = ((**g** [^] c_v*'*) [^] *s0*) ⊗ ((**g** [^] *b*) [^] *r0*);
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 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)
⟨*proof*⟩

definition *S2* :: *bool* ⇒ *'grp* ⇒ (*bool* × *'grp* × *'grp* × *'grp* × *'grp* × *'grp* × *'grp* × *'grp*) *spmf*
where *S2 v m* = *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*;
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*);
s' ← *sample-uniform* (*order* 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*)
⟨*proof*⟩

sublocale *sim-def*: *sim-det-def R1 S1 R2 S2 funct-OT-12 protocol*
⟨*proof*⟩

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*⟩

lemma *correctness*:
assumes *m0* ∈ *carrier* $\mathcal{G}$ *m1* ∈ *carrier* $\mathcal{G}$
shows *sim-def.correctness* (*m0*,*m1*) *σ*
⟨*proof*⟩

lemma *security-P1*:
shows *sim-def.adv-P1 msgs σ D* ≤ *ddh.advantage* (*R1-inter-adversary D msgs*) + *ddh.advantage* (*inter-S1-adversary D msgs*)
(**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩ **including** *monad-normalisation*
⟨*proof*⟩

lemma *add-mult-one-time-pad*:
assumes *s0* < *order* $\mathcal{G}$
and *s0* ≠ *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*⟩

lemma *security-P2*:
assumes *m0* ∈ *carrier* $\mathcal{G}$ *m1* ∈ *carrier* $\mathcal{G}$
shows *sim-def.perfect-sec-P2* (*m0*,*m1*) *σ*
⟨*proof*⟩
including *monad-normalisation*
⟨*proof*⟩

end

locale *np-asymp* =
  fixes $\mathcal{G}$ :: *security* ⇒ *'grp cyclic-group*
  assumes *np*: ⋀η. *np* ($\mathcal{G}$ η)
begin

sublocale *np* $\mathcal{G}$ η **for** η ⟨*proof*⟩

theorem *correctness-asymp*:
  assumes *m0* ∈ *carrier* ($\mathcal{G}$ η) *m1* ∈ *carrier* ($\mathcal{G}$ η)
  shows *sim-def.correctness* η (*m0*, *m1*) σ
  ⟨*proof*⟩

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*⟩

theorem *security-P2-asymp*:
  assumes *m0* ∈ *carrier* ($\mathcal{G}$ η) *m1* ∈ *carrier* ($\mathcal{G}$ η)
  shows *sim-def.perfect-sec-P2* η (*m0*,*m1*) σ
  ⟨*proof*⟩

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* × *bool* × *bool* × *bool*
type-synonym *input2* = *bool* × *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*
⟨*proof*⟩

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*⟩

lemma *OT-12-P2-assm*: *R2-OT12* (*m0*,*m1*) σ = *funct-OT-12* (*m0*,*m1*) σ ≫= (λ (*out1*, *out2*). *S2-OT12* σ *out2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩
  including *monad-normalisation* ⟨*proof*⟩

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*⟩
  including *monad-normalisation* ⟨*proof*⟩
  including *monad-normalisation* ⟨*proof*⟩

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*⟩

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*
 ⟨*proof*⟩

Party 2 security.

lemma *coin-coin*: *map-spmf* (λ *S0*. *S0* ⊕ *S3* ⊕ *m1*) *coin-spmf* = *coin-spmf*
 (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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⟩

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⟩

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⟩

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*)
⟨proof⟩

lemma *P2-sec-OT-14*: *R2-14* *M* *C* = *funct-OT-14* *M* *C* ⤜ (λ (*out1*, *out2*). *S2-14* *C* *out2*)
⟨proof⟩

sublocale *OT-14*: *sim-det-def R1-14 S1-14 R2-14 S2-14 funct-OT-14 protocol-14-OT*
⟨proof⟩

lemma *correctness-OT-14*:
shows *funct-OT-14* *M* *C* = *protocol-14-OT* *M* *C*
⟨proof⟩

lemma *OT-14-correct*: *OT-14*.*correctness* *M* *C*
⟨proof⟩

lemma *OT-14-P2-sec*: *OT-14*.*perfect-sec-P2* *m1* *m2*
⟨proof⟩

lemma *OT-14-P1-sec*: *OT-14*.*adv-P1* *m1* *m2* *D* ≤ 3 ∗ *adv-OT12*
⟨proof⟩

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*) ⟨*proof*⟩

lemma *OT-14-P1-sec*: *OT-14.adv-P1* (*R1-12-0T n*) *n m1 m2 D* ≤ *3* ∗ (*adv-OT12 n*)
⟨*proof*⟩

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*⟩

theorem *OT-14-P2-asym-sec*: *OT-14.perfect-sec-P2 R2-OT12 n m1 m2*
⟨*proof*⟩

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*))
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *inf-th-14-OT-P4*: *R2-14-OT msgs C* = (*funct-OT-14 msgs C* ⤜ (λ (*s1*, *s2*). *S2-14-OT C s2*))
  ⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *reconstruct-share* : (*bind-spmf* (*share x*) *reconstruct*) = (*return-spmf x*)
⟨*proof*⟩

lemma (*reconstruct* (*s1*,*s2*) ⤜ (λ *rec*. *share rec* ⤜ (λ *shares*. *reconstruct shares*)))
= *return-spmf* (*s1* ⊕ *s2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *P1-xor-inf-th*: *R1-xor A B* = *xor-funct A B* ⤜ (λ (*out1*, *out2*). *S1-xor A out1*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *P2-xor-inf-th*: *R2-xor A B* = *xor-funct A B* ⤜ (λ (*out1*, *out2*). *S2-xor B out2*)
  ⟨*proof*⟩

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

  ⟨*proof*⟩

lemma *xor-sim-det.perfect-sec-P1 m1 m2*
  ⟨*proof*⟩

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

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*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *and-correct*: *and-protocol* (*a1*, *b1*) (*a2*,*b2*) = *and-funct* (*a1*, *b1*) (*a2*,*b2*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *ideal-S1-and*: *and-secret-sharing.Ideal1* (*a1*, *b1*) (*a2*, *b2*) *s2* = *S1-and′* (*a1*, *b1*) (*a2*, *b2*) *s2*
⟨*proof*⟩

lemma *and-P2-security*: *and-secret-sharing.perfect-sec-P2 m1 m2*
⟨*proof*⟩

lemma *and-P1-security*: *and-secret-sharing.adv-P1 m1 m2 D* ≤ *adv-14-OT*
⟨*proof*⟩

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

lemma *share-reconstruct-xor*: *share x* ⤜ (λ(*a1*, *a2*). *share y* ⤜ (λ(*b1*, *b2*).
xor-protocol (*a1*, *b1*) (*a2*, *b2*) ⤜ (λ(*a*, *b*).
reconstruct (*a*, *b*)))) = *xor-evaluate x y*
⟨*proof*⟩

sublocale *share-correct*: *secret-sharing-scheme share reconstruct F* ⟨*proof*⟩

lemma *share-correct.sharing-correct input*
⟨*proof*⟩

lemma *share-correct.correct-share-eval input1 input2*
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *xor-sim-det.perfect-sec-P1 m1 m2*
⟨*proof*⟩

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

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*⟩

lemma *and-P2-security*: *and-secret-sharing.perfect-sec-P2 n m1 m2*
⟨*proof*⟩

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*}
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *minus-mod*:
  assumes *a* > *b*
  shows [*a* − *b mod q* = *a* − *b*] (*mod q*)
  ⟨*proof*⟩

lemma *q-cong*:[*a* = *q* + *a*] (*mod q*)
  ⟨*proof*⟩

lemma *q-cong-reverse*: [*q* + *a* = *a*] (*mod q*)
  ⟨*proof*⟩

lemma *qq-cong*: [*a* = *q*∗*q* + *a*] (*mod q*)
  ⟨*proof*⟩

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*⟩

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*⟩

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*)

⟨proof⟩

lemma *c1*:
assumes $e2 = (x + c1)$ *mod* q
and $x < q$ $c1 < q$
shows $c1 = (e2 + q - x)$ *mod* q
⟨proof⟩

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⟩

lemma *minus-mod-cancel*:
assumes $a - b > 0$ $a - b \ mod \ q > 0$
shows $[a - b + c = a - b \ mod \ q + c] \ (mod \ q)$
⟨proof⟩

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⟩

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⟩

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*⟩

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*⟩

theorem *P2-security*:
assumes $x < q \ y < q$
shows *sim-non-det-def.perfect-sec-P2 x y*
including *monad-normalisation*
⟨*proof*⟩

lemma *s1-s2-P1*: **assumes** $x < q \ xa < q \ xb < q \ xc < q \ y < q$
shows $((x, xa, xb, (y + q - xc) \ mod \ q), (x * ((y + q - xc) \ mod \ q) + q - xb) \ mod \ q, ((x + xa) \ mod \ q * xc + q - (xa * xc + q - xb) \ mod \ q) \ mod \ q) =$
$((x, xa, xb, (y + q - xc) \ mod \ q), (x * ((y + q - xc) \ mod \ q) + q - xb) \ mod \ q, (x * y + q - (x * ((y + q - xc) \ mod \ q) + q - xb) \ mod \ q) \ mod \ q)$
⟨*proof*⟩

lemma *mod-minus*: **assumes** $a - b > 0$ **and** $c - d > 0$
shows $(a - b + (c - d \ mod \ q)) \ mod \ q = (a - b + (c - d)) \ mod \ q$
⟨*proof*⟩

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*⟩

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*⟩

theorem *P1-security*:
assumes $x < q$ $y < q$
shows *sim-non-det-def.perfect-sec-P1 x y*
including *monad-normalisation*
⟨*proof*⟩

end

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

sublocale *secure-mult q n* **for** *n*
⟨*proof*⟩

theorem *P1-secure*:
assumes $x < q\ n$ $y < q\ n$
shows *sim-non-det-def.perfect-sec-P1 n x y*
⟨*proof*⟩

theorem *P2-secure*:
assumes $x < q\ n$ $y < q\ n$
shows *sim-non-det-def.perfect-sec-P2 n x y*
⟨*proof*⟩

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* $\mathcal{G}$ $\mathcal{A}$
⟨*proof*⟩

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*|
⟨*proof*⟩

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*|
⟨*proof*⟩

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*⟩

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*|
⟨*proof*⟩

lemma *DDH-advantage-bound*: *ddh.DDH-advantage* $\mathcal{G}$ $\mathcal{A}$ ≤ *ddh.advantage* $\mathcal{G}$ $\mathcal{A}$ + *ddh.advantage* $\mathcal{G}$ (*ddh.DDH-$\mathcal{A}'$* $\mathcal{G}$ $\mathcal{A}$)
⟨*proof*⟩

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* 𝒜 = (*let* (*S1*, *S2*) = *S in* (*ideal-game-1 x y z* (*S1* 𝒜, *S2* 𝒜)))

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

definition *perfect-sec-P1 x y z S* 𝒜 ⟷ (*ideal-view-1 x y z S* 𝒜 = *P1-real-view x y z* 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* 𝒜 (*D* :: (*'adv-out1* × *'out2*) ⇒ *bool spmf*) =
  |*spmf* (*P1-real-view x y z* 𝒜 ⤜ (λ *view*. *D view*)) *True*
    − *spmf* (*ideal-view-1 x y z S* 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* 𝒜 = (*let* (*S1*, *S2*) = *S in* (*ideal-game-2 x y z* (*S1* 𝒜, *S2* 𝒜)))

definition *perfect-sec-P2 x y z S* 𝒜 ⟷ (*ideal-view-2 x y z S* 𝒜 = *P2-real-view x y z* 𝒜)

definition *adv-P2 x y z S* 𝒜 (*D* :: (*'out1* × *'adv-out2*) ⇒ *bool spmf*) =
  |*spmf* (*P2-real-view x y z* 𝒜 ⤜ (λ *view*. *D view*)) *True*

$- \mathit{spmf}\ (\mathit{ideal\text{-}view\text{-}2}\ x\ y\ z\ S\ \mathcal{A} \gg= (\lambda\ \mathit{view}.\ D\ \mathit{view}))\ \mathit{True}\ |$

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* 𝒢)
begin

lemma *prime-field*: *a* < (*order* 𝒢) ⟹ *a* ≠ *0* ⟹ *coprime a* (*order* 𝒢)
⟨*proof*⟩

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* 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* ⊗ **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 ← *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*);
return-spmf ((), (*if σ then* (*z1* ⊗ *inv* (*w1* [^] *α1*)) *else* (*z0* ⊗ *inv* (*w0* [^] *α0*))))}

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 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* ⊗ (*if σ then* **g** *else* **1**);
let b1 = *h1* [^] *r* ⊗ (*if σ then* **g** *else* **1**);
((*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*);
(*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 :: *'adv-out1* ← *𝒜3 s'*;
return-spmf (*adv-out*, (*if σ then* (*z1* ⊗ (*inv w1* [^] *α1*)) *else* (*z0* ⊗ (*inv w0* [^] *α0*)))))}

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 𝒜 M z = 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*));
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 xσ else* **1**);
let e0 = (*w0*,*z0*);
let e1 = (*w1*,*z1*);
𝒜3 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* ⟨*proof*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

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 (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
    α0 ← *sample-uniform* (*order 𝒢*);
    let h0 = **g** [^] *α0*;
    let h1 = *h*;
    let b0 = *a* [^] *α0*;
    let b1 = *t*;
    ((*in1* :: ′*grp*, *in2* ::′*grp*, *in3* :: ′*grp*),*s*) ← *𝒜1 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*′) ← *𝒜2 h0 h1 a b0 b1 M s*;
    let x0 = (*z0* ⊗ (*inv w0* [^] *α0*));
    adv-out :: ′*adv-out1* ← *𝒜3 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 (*𝒜1*, *𝒜2*, *𝒜3*) = *𝒜*;
    α1 :: *nat* ← *sample-uniform* (*order 𝒢*);
    let h1 = **g** [^] *α1*;
    let h0 = *h*;
    let b0 = *t*;
    let b1 = *a* [^] *α1* ⊗ **g**;
    ((*in1* :: ′*grp*, *in2* ::′*grp*, *in3* :: ′*grp*),*s*) ← *𝒜1 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*′) ← *𝒜2 h0 h1 a b0 b1 M s*;
    let x1 = (*z1* ⊗ (*inv w1* [^] *α1*));
    adv-out :: ′*adv-out1* ← *𝒜3 s*′;
    D (*adv-out*, *x1*)}

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*), *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*));
    let σ′ = (*if l* = **1** *then False else True*);
    (- :: *unit*, *xσ*) ← *funct-OT-12* (*x0*, *x1*) *σ*′;

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 xσ else **1**);
let e0 = (w0,z0);
let e1 = (w1,z1);
out ← 𝒜3 e0 e1 s′;
return-spmf ((), out)}

definition P2-ideal-model-end :: (′grp × ′grp) ⇒ ′grp ⇒ ((′grp × ′grp × ′grp × ′grp × ′grp) × ′state)
⇒ (′grp, ′adv-out2, ′state) adv-3-P2 ⇒ (unit × ′adv-out2) spmf
where P2-ideal-model-end M l bs 𝒜3 = do {
let (x0,x1) = M;
let ((h0,h1,a,b0,b1),s) = bs;
let σ′ = (if l = **1** then False else True);
(-:: unit, xσ) ← funct-OT-12 (x0, x1) σ′;
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 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*
⟨*proof*⟩

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 (*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*;
P2-real-model-end M ((*h0*,*h1*,*a*,*b0*,*b1*),*s*′) *𝒜3*}

lemma *P2-real-model-rewrite*: *P2-real-model M σ z 𝒜 = P2-real-model′ M σ z 𝒜*
⟨*proof*⟩

lemma *P2-ideal-view-unfold*: *mal-def.ideal-view-2* (*x0*,*x1*) *σ z* (*P2-S1*, *P2-S2*) *𝒜* = *P2-ideal-model* (*x0*,*x1*) *σ z 𝒜*
⟨*proof*⟩

end

locale *ot* = *ot-base* + *cyclic-group* 𝒢

begin

lemma *P1-assert-correct1*:
shows $((\mathbf{g} \ [\char94] \ (\alpha 0::nat)) \ [\char94] \ (r::nat) \otimes \mathbf{g} \otimes inv \ ((\mathbf{g} \ [\char94] \ (\alpha 1::nat)) \ [\char94] \ r \otimes \mathbf{g})$
$= (\mathbf{g} \ [\char94] \ \alpha 0 \otimes inv \ (\mathbf{g} \ [\char94] \ \alpha 1)) \ [\char94] \ r)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *P1-assert-correct2*:
shows $(\mathbf{g} \ [\char94] \ (\alpha 0::nat)) \ [\char94] \ (r::nat) \otimes inv \ ((\mathbf{g} \ [\char94] \ (\alpha 1::nat)) \ [\char94] \ r) = (\mathbf{g} \ [\char94] \ \alpha 0 \otimes inv \ (\mathbf{g} \ [\char94] \ \alpha 1)) \ [\char94] \ r$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

sublocale *ddh*: *ddh-ext*
⟨*proof*⟩

lemma *P1-real-ddh0-σ-false*:
assumes σ = *False*
shows ((*P1-real-model M σ z 𝒜*) ⤜ (λ *view*. *D view*)) = (*ddh.DDH0* (*P1-DDH-mal-adv-σ-false M z 𝒜 D*))
including *monad-normalisation*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *P1-real-ideal-DDH-advantage-false*:
assumes σ = *False*
shows *mal-def.adv-P1 M σ z* (*P1-S1*, *P1-S2*) *𝒜 D* = *ddh.DDH-advantage* (*P1-DDH-mal-adv-σ-false M z 𝒜 D*)

⟨proof⟩

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*
    ≤ *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*))
  ⟨proof⟩

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*)
  ⟨proof⟩

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*
    ≤ *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*))
  ⟨proof⟩

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⟩

lemma *P2-inv-g-rewrite*:
  assumes *s* < *order* $\mathcal{G}$
  shows (*inv* **g**) [^] (*u1'* + (*order* $\mathcal{G}$ − *s*)) = **g** [^] *s* ⊗ *inv* (**g** [^] *u1'*)
⟨proof⟩

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⟩

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 [^] ($r * \alpha * ((order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G}) + (r * s + xa)\ mod\ order\ \mathcal{G} * \alpha$)
⊗ **g** [^] ($(order\ \mathcal{G} - s + x)\ mod\ order\ \mathcal{G} + s$))
⟨*proof*⟩

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 [^] ($r * \alpha * ((order\ \mathcal{G} * order\ \mathcal{G} - s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G})))\ mod\ order\ \mathcal{G})) + x)\ mod\ order\ \mathcal{G})$
$+ (r * s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G})))\ mod\ order\ \mathcal{G})) + xa)\ mod\ order\ \mathcal{G} * \alpha$) ⊗
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** [^] ($r * \alpha * x + xa * \alpha$) ⊗ **g** [^] ($t * x$))
⟨*proof*⟩

lemma *P2-case-l-neq-1-gt-x0-rewrite*:
assumes $t < order\ \mathcal{G}$
and $t \neq 0$
shows **g** [^] ($t * (u0 + (s * (nat\ ((fst\ (bezw\ t\ (order\ \mathcal{G})))\ mod\ order\ \mathcal{G}))))$) = **g** [^] ($t * u0$) ⊗ **g** [^] s
⟨*proof*⟩

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*: $b0 \otimes inv\ b1 = (h0 \otimes inv\ h1)$ [^] r
and *assert-in-carrier*: $h0 \in carrier\ \mathcal{G} \wedge h1 \in carrier\ \mathcal{G} \wedge b0 \in carrier\ \mathcal{G} \wedge b1 \in carrier\ \mathcal{G}$
and *x1-in-carrier*: $x1 \in carrier\ \mathcal{G}$
and *x0-in-carrier*: $x0 \in carrier\ \mathcal{G}$
shows *P2-ideal-model-end* $(x0,x1)$ ($b0 \otimes (inv\ (h0$ [^] $r)))$ (($h0,h1$, **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*⟩

lemma *P2-ideal-real-eq*:
assumes *x1-in-carrier*: $x1 \in carrier\ \mathcal{G}$
and *x0-in-carrier*: $x0 \in carrier\ \mathcal{G}$
shows *P2-real-model* $(x0,x1)\ \sigma\ z\ \mathcal{A}$ = *P2-ideal-model* $(x0,x1)\ \sigma\ z\ \mathcal{A}$
⟨*proof*⟩

lemma *malicious-sec-P2*:
  assumes *x1-in-carrier*: *x1* ∈ *carrier* $\mathcal{G}$
    and *x0-in-carrier*: *x0* ∈ *carrier* $\mathcal{G}$
  shows *mal-def.perfect-sec-P2* (*x0*,*x1*) σ *z* (*P2-S1*, *P2-S2*) $\mathcal{A}$
  ⟨*proof*⟩

lemma *correct*:
  assumes *x0* ∈ *carrier* $\mathcal{G}$
    and *x1* ∈ *carrier* $\mathcal{G}$
  shows *funct-OT-12* (*x0*, *x1*) σ = *protocol-ot* (*x0*,*x1*) σ
⟨*proof*⟩

lemma *correctness*:
  assumes *x0* ∈ *carrier* $\mathcal{G}$
    and *x1* ∈ *carrier* $\mathcal{G}$
  shows *mal-def.correct* (*x0*,*x1*) σ
  ⟨*proof*⟩

end

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

sublocale *ot* $\mathcal{G}$ *n* **for** *n* ⟨*proof*⟩

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

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*⟩

lemma *P2-security-asym*:
  assumes *x1-in-carrier*: *x1* ∈ *carrier* ($\mathcal{G}$ *n*)
    and *x0-in-carrier*: *x0* ∈ *carrier* ($\mathcal{G}$ *n*)

shows *mal-def.perfect-sec-P2 n (x0,x1)* σ *z (P2-S1 n, P2-S2 n)* $\mathcal{A}$
⟨*proof*⟩

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.