

Multi-Party Computation

David Aspinall and David Butler

September 13, 2023

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

1	Uniform Sampling	7
2	Semi-Honest Security	14
2.1	Security definitions	14
2.1.1	Security for deterministic functionalities	14
2.1.2	Security definitions for non deterministic functionalities	17
2.1.3	Secret sharing schemes	18
2.2	Oblivious Transfer functionalities	18
2.3	ETP definitions	19
2.4	Oblivious transfer constructed from ETPs	21
2.4.1	RSA instantiation	30
2.5	Noar Pinkas OT	46
2.6	1-out-of-2 OT to 1-out-of-4 OT	54
2.7	1-out-of-4 OT to GMW	71
2.8	Secure multiplication protocol	83
2.9	DHH Extension	102
3	Malicious Security	104
3.1	Malicious Security Definitions	104
3.2	Malicious OT	106

```

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

context cyclic-group begin

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

lemma pow-generator-mod:  $\mathbf{g} [\wedge] (k \bmod \text{order } G) = \mathbf{g} [\wedge] k$ 
proof(cases order  $G > 0$ )
  case True
  obtain  $n$  where  $n: k = n * \text{order } G + k \bmod \text{order } G$  by (metis div-mult-mod-eq)
  have  $\mathbf{g} [\wedge] k = (\mathbf{g} [\wedge] \text{order } G) [\wedge] n \otimes \mathbf{g} [\wedge] (k \bmod \text{order } G)$ 
    by(subst  $n$ )(simp add: nat-pow-mult nat-pow-pow mult-ac)
  then show ?thesis by(simp add: generator-pow-order)
qed simp

lemma int-nat-pow:
  assumes  $a \geq 0$ 
  shows  $(\mathbf{g} [\wedge] (\text{int } (a :: \text{nat}))) [\wedge] (b :: \text{int}) = \mathbf{g} [\wedge] (a*b)$ 
  using assms
proof(cases  $a > 0$ )
  case True
  show ?thesis
    using int-pow-pow by blast
  next case False
  have  $(\mathbf{g} [\wedge] (\text{int } (a :: \text{nat}))) [\wedge] (b :: \text{int}) = \mathbf{1}$  using False by simp
  also have  $\mathbf{g} [\wedge] (a*b) = \mathbf{1}$  using False by simp
  ultimately show ?thesis by simp
qed

lemma pow-generator-mod-int:  $\mathbf{g} [\wedge] ((k :: \text{int}) \bmod \text{order } G) = \mathbf{g} [\wedge] k$ 
proof(cases order  $G > 0$ )
  case True
  obtain  $n :: \text{int}$  where  $n: k = \text{order } G * n + k \bmod \text{order } G$ 
    by (metis div-mult-mod-eq mult.commute)

```

then have $\mathbf{g} \ [\] \ k = \mathbf{g} \ [\] \ (\text{order } G * n) \otimes \mathbf{g} \ [\] \ (k \text{ mod } \text{order } G)$
using *int-pow-mult nat-pow-mult by (metis generator-closed)*
then have $\mathbf{g} \ [\] \ k = (\mathbf{g} \ [\] \ \text{order } G) \ [\] \ n \otimes \mathbf{g} \ [\] \ (k \text{ mod } \text{order } G)$
using *int-nat-pow by (simp add: int-pow-int)*
then show *?thesis by (simp add: generator-pow-order)*
qed *simp*

lemma *pow-gen-mod-mult:*
shows $(\mathbf{g} \ [\] \ (a::\text{nat}) \otimes \mathbf{g} \ [\] \ (b::\text{nat})) \ [\] \ ((c::\text{int}) * \text{int } (d::\text{nat})) = (\mathbf{g} \ [\] \ a \otimes \mathbf{g} \ [\] \ b) \ [\] \ ((c * \text{int } d) \text{ mod } (\text{order } G))$
proof –
have $(\mathbf{g} \ [\] \ (a::\text{nat}) \otimes \mathbf{g} \ [\] \ (b::\text{nat})) \in \text{carrier } G$ **by** *simp*
then obtain $n :: \text{nat}$ **where** $n: \mathbf{g} \ [\] \ n = (\mathbf{g} \ [\] \ (a::\text{nat}) \otimes \mathbf{g} \ [\] \ (b::\text{nat}))$
by *(simp add: monoid.nat-pow-mult)*
also obtain r **where** $r: r = c * \text{int } d$ **by** *simp*
have $(\mathbf{g} \ [\] \ (a::\text{nat}) \otimes \mathbf{g} \ [\] \ (b::\text{nat})) \ [\] \ ((c::\text{int}) * \text{int } (d::\text{nat})) = (\mathbf{g} \ [\] \ n) \ [\] \ r$
using $n \ r$ **by** *simp*
moreover have $\dots = (\mathbf{g} \ [\] \ n) \ [\] \ (r \text{ mod } (\text{order } G))$ **using** *pow-generator-mod-int pow-generator-mod*
by *(metis int-nat-pow int-pow-int mod-mult-right-eq zero-le)*
moreover have $\dots = (\mathbf{g} \ [\] \ a \otimes \mathbf{g} \ [\] \ b) \ [\] \ ((c * \text{int } d) \text{ mod } (\text{order } G))$ **using** r
by *simp*
ultimately show *?thesis by simp*
qed

lemma *pow-generator-eq-iff-cong:*
finite (carrier G) $\implies \mathbf{g} \ [\] \ x = \mathbf{g} \ [\] \ y \iff [x = y] \ (\text{mod } \text{order } G)$
by *(subst (1 2) pow-generator-mod[symmetric])(auto simp add: cong-def order-gt-0-iff-finite intro: inj-onD[OF inj-on-generator])*

lemma *cyclic-group-commute:*
assumes $a \in \text{carrier } G \ b \in \text{carrier } G$
shows $a \otimes b = b \otimes a$
(is ?lhs = ?rhs)
proof –
obtain $n :: \text{nat}$ **where** $n: a = \mathbf{g} \ [\] \ n$ **using** *generatorE assms by auto*
also obtain $k :: \text{nat}$ **where** $k: b = \mathbf{g} \ [\] \ k$ **using** *generatorE assms by auto*
ultimately have $?lhs = \mathbf{g} \ [\] \ n \otimes \mathbf{g} \ [\] \ k$ **by** *simp*
then have $\dots = \mathbf{g} \ [\] \ (n + k)$ **by** *(simp add: nat-pow-mult)*
then have $\dots = \mathbf{g} \ [\] \ (k + n)$ **by** *(simp add: add.commute)*
then show *?thesis by (simp add: nat-pow-mult n k)*
qed

lemma *cyclic-group-assoc:*
assumes $a \in \text{carrier } G \ b \in \text{carrier } G \ c \in \text{carrier } G$
shows $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
(is ?lhs = ?rhs)
proof –

obtain $n :: \text{nat}$ **where** $n: a = \mathbf{g} \ [\] \ n$ **using** *generatorE assms by auto*

obtain $k :: \text{nat}$ **where** $k: b = \mathbf{g} [\wedge] k$ **using** *generatorE assms* **by** *auto*
obtain $j :: \text{nat}$ **where** $j: c = \mathbf{g} [\wedge] j$ **using** *generatorE assms* **by** *auto*
have $?lhs = (\mathbf{g} [\wedge] n \otimes \mathbf{g} [\wedge] k) \otimes \mathbf{g} [\wedge] j$ **using** $n\ k\ j$ **by** *simp*
then have $\dots = \mathbf{g} [\wedge] (n + (k + j))$ **by** (*simp add: nat-pow-mult add.assoc*)
then show $?thesis$ **by** (*simp add: nat-pow-mult n k j*)
qed

lemma *l-cancel-inv:*

assumes $h \in \text{carrier } G$
shows $(\mathbf{g} [\wedge] (a :: \text{nat}) \otimes \text{inv } (\mathbf{g} [\wedge] a)) \otimes h = h$
(is ?lhs = ?rhs)
proof –
have $?lhs = (\mathbf{g} [\wedge] \text{int } a \otimes \text{inv } (\mathbf{g} [\wedge] \text{int } a)) \otimes h$ **by** *simp*
then have $\dots = (\mathbf{g} [\wedge] \text{int } a \otimes (\mathbf{g} [\wedge] (- a))) \otimes h$ **using** *int-pow-neg[symmetric]*
by *simp*
then have $\dots = \mathbf{g} [\wedge] (\text{int } a - a) \otimes h$ **by** (*simp add: int-pow-mult*)
then have $\dots = \mathbf{g} [\wedge] ((0 :: \text{int})) \otimes h$ **by** *simp*
then show $?thesis$ **by** (*simp add: assms*)
qed

lemma *inverse-split:*

assumes $a \in \text{carrier } G$ **and** $b \in \text{carrier } G$
shows $\text{inv } (a \otimes b) = \text{inv } a \otimes \text{inv } b$
by (*simp add: assms comm-group.inv-mult cyclic-group-commute group-comm-groupI*)

lemma *inverse-pow-pow:*

assumes $a \in \text{carrier } G$
shows $\text{inv } (a [\wedge] (r :: \text{nat})) = (\text{inv } a) [\wedge] r$
proof –
have $a [\wedge] r \in \text{carrier } G$
using *assms* **by** *blast*
then show $?thesis$
by (*simp add: assms nat-pow-inv*)
qed

lemma *l-neq-1-exp-neq-0:*

assumes $l \in \text{carrier } G$
and $l \neq 1$
and $l = \mathbf{g} [\wedge] (t :: \text{nat})$
shows $t \neq 0$
proof (*rule ccontr*)
assume $\neg (t \neq 0)$
hence $t = 0$ **by** *simp*
hence $\mathbf{g} [\wedge] t = 1$ **by** *simp*
then show *False* **using** *assms* **by** *simp*
qed

lemma *order-gt-1-gen-not-1:*

assumes $\text{order } G > 1$

```

shows  $g \neq 1$ 
proof(rule ccontr)
  assume  $\neg g \neq 1$ 
  hence  $g = 1$  by simp
  hence g-pow-eq-1:  $g [\wedge] n = 1$  for  $n :: nat$  by simp
  hence range  $(\lambda n :: nat. g [\wedge] n) = \{1\}$  by auto
  hence carrier  $G \subseteq \{1\}$  using generator by auto
  hence order  $G < 1$ 
    by (metis One-nat-def assms g-pow-eq-1 inj-onD inj-on-generator lessThan-iff
not-gr-zero zero-less-Suc)
  with assms show False by simp
qed

```

```

lemma power-swap:  $((g [\wedge] (\alpha 0 :: nat)) [\wedge] (r :: nat)) = ((g [\wedge] r) [\wedge] \alpha 0)$ 
(is ?lhs = ?rhs)

```

```

proof -
  have ?lhs =  $g [\wedge] (\alpha 0 * r)$ 
    using nat-pow-pow mult.commute by auto
  hence ... =  $g [\wedge] (r * \alpha 0)$ 
    by (metis mult.commute)
  thus ?thesis using nat-pow-pow by auto
qed

```

end

end

```

theory Number-Theory-Aux imports

```

```

  HOL-Number-Theory.Cong
  HOL-Number-Theory.Residues

```

```

begin

```

```

lemma bezv-inverse:

```

```

  assumes gcd  $(e :: nat) (N :: nat) = 1$ 
  shows  $[nat\ e * nat\ ((fst\ (bezv\ e\ N))\ mod\ N) = 1] (mod\ nat\ N)$ 

```

```

proof -

```

```

  have  $(fst\ (bezv\ e\ N) * e + snd\ (bezv\ e\ N) * N) mod\ N = 1 mod\ N$ 
    by (metis assms bezv-aux zmod-int)

```

```

  hence  $(fst\ (bezv\ e\ N) mod\ N * e mod\ N) = 1 mod\ N$ 

```

```

    by (simp add: mod-mult-right-eq mult.commute)

```

```

  hence cong-eq:  $[(fst\ (bezv\ e\ N) mod\ N * e) = 1] (mod\ N)$ 

```

```

    by (metis of-nat-1 zmod-int cong-def)

```

```

  hence  $[nat\ (fst\ (bezv\ e\ N) mod\ N) * e = 1] (mod\ N)$ 

```

```

proof -

```

```

  { assume int  $(nat\ (fst\ (bezv\ e\ N) mod\ int\ N)) \neq fst\ (bezv\ e\ N) mod\ int\ N$ 

```

```

    have  $N = 0 \longrightarrow 0 \leq fst\ (bezv\ e\ N) mod\ int\ N$ 

```

```

      by fastforce

```

```

    then have int  $(nat\ (fst\ (bezv\ e\ N) mod\ int\ N)) = fst\ (bezv\ e\ N) mod\ int\ N$ 

```

```

      by fastforce }

```

```

  then have  $[int\ (nat\ (fst\ (bezv\ e\ N) mod\ int\ N) * e) = int\ 1] (mod\ int\ N)$ 

```

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

lemma inverse:

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

lemma prod-not-prime:

assumes prime (x::nat)
 and prime y
 and $x > 2$
 and $y > 2$
 shows $\neg prime ((x-1)*(y-1))$
 by (metis assms One-nat-def Suc-diff-1 nat-neq-iff numeral-2-eq-2 prime-gt-0-nat
 prime-product)

lemma ex-inverse:

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

lemma ex-k1-k2:

assumes coprime: coprime (e :: nat) ((P-1)*(Q-1))
 and $[e*d = 1] \pmod{(P-1)}$

shows $\exists k1\ k2. e*d + k1*(P-1) = 1 + k2*(P-1)$
by (*metis assms(2) cong-iff-lin-nat*)

lemma *ex-k-mod*:

assumes *coprime*: *coprime* ($e :: nat$) $((P-1)*(Q-1))$

and $P \neq Q$

and *prime* P

and *prime* Q

and $d \neq 0$

and $[e*d = 1] \pmod{(P-1)}$

shows $\exists k. e*d = 1 + k*(P-1)$

proof -

have $e > 0$

using *assms(1) assms(2) prime-gt-0-nat* **by** *fastforce*

then have $e*d \geq 1$ **using** *assms* **by** *simp*

then obtain k **where** $k: e*d = 1 + k*(P-1)$

using *assms(6) cong-to-1'-nat* **by** *auto*

then show *?thesis*

by *simp*

qed

lemma *fermat-little*:

assumes *prime* ($P :: nat$)

shows $[x^P = x] \pmod{P}$

proof(*cases P dvd x*)

case *True*

hence $x \pmod{P} = 0$ **by** *simp*

moreover have $x^P \pmod{P} = 0$

by (*simp add: True assms prime-dvd-power-nat-iff prime-gt-0-nat*)

ultimately show *?thesis*

by (*simp add: cong-def*)

next

case *False*

hence $[x^{(P-1)} = 1] \pmod{P}$

using *fermat-theorem assms* **by** *blast*

then show *?thesis*

by (*metis assms cong-def diff-diff-cancel diff-is-0-eq' diff-zero mod-mult-right-eq power-eq-if power-one-right prime-ge-1-nat zero-le-one*)

qed

end

1 Uniform Sampling

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

theory *Uniform-Sampling*

imports

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

begin

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

definition *sample-uniform-units* :: $\text{nat} \Rightarrow \text{nat} \text{ spmf}$
where *sample-uniform-units* $q = \text{spmf-of-set } (\{..<q\} - \{0\})$

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

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

General lemma for mapping using uniform sampling from units.

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

General lemma for mapping using uniform sampling.

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

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

The addition map case.

lemma *inj-add*:

assumes *x*: $x < q$
and *x'*: $x' < q$
and *map*: $((y :: nat) + x) \bmod q = (y + x') \bmod q$
shows $x = x'$

proof –

have *aa*: $((y :: nat) + x) \bmod q = (y + x') \bmod q \implies x \bmod q = x' \bmod q$

proof –

have *4*: $((y :: nat) + x) \bmod q = (y + x') \bmod q \implies [((y :: nat) + x) = (y + x')] \bmod q$

by(*simp add: cong-def*)

have *5*: $[((y :: nat) + x) = (y + x')] \bmod q \implies [x = x'] \bmod q$

by (*simp add: cong-add-lcancel-nat*)

have *6*: $[x = x'] \bmod q \implies x \bmod q = x' \bmod q$

by(*simp add: cong-def*)

then show *?thesis* **by**(*simp add: map 4 5 6*)

qed

also have *bb*: $x \bmod q = x' \bmod q \implies x = x'$

by(*simp add: x x'*)

ultimately show *?thesis* **by**(*simp add: map*)

qed

lemma *inj-uni-samp-add*: *inj-on* $(\lambda(b :: nat). (y + b) \bmod q) \{..<q\}$

by(*simp add: inj-on-def*)(*auto simp only: inj-add*)

lemma *surj-uni-samp*:

assumes *inj*: *inj-on* $(\lambda(b :: nat). (y + b) \bmod q) \{..<q\}$

shows $(\lambda(b :: nat). (y + b) \bmod q) \{..<q\} = \{..<q\}$

apply(*rule endo-inj-surj*) **using** *inj* **by** *auto*

lemma *samp-uni-plus-one-time-pad*:

shows *map-spmf* $(\lambda b. (y + b) \bmod q)$ (*sample-uniform* *q*) = (*sample-uniform* *q*)

using *inj-uni-samp-add surj-uni-samp one-time-pad* **by** *simp*

The multiplication map case.

lemma *inj-mult*:

assumes *coprime*: *coprime* *x* (*q*::*nat*)

and *y*: $y < q$

and *y'*: $y' < q$

and *map*: $x * y \bmod q = x * y' \bmod q$

shows $y = y'$

proof –

have $x*y \bmod q = x*y' \bmod q \implies y \bmod q = y' \bmod q$

proof –

```

have  $x*y \bmod q = x*y' \bmod q \implies [x*y = x*y'] \pmod{q}$ 
  by(simp add: cong-def)
also have  $[x*y = x*y'] \pmod{q} = [y = y'] \pmod{q}$ 
  by(simp add: cong-mult-lcancel-nat coprime)
also have  $[y = y'] \pmod{q} \implies y \bmod q = y' \bmod q$ 
  by(simp add: cong-def)
ultimately show ?thesis by(simp add: map)
qed
also have  $y \bmod q = y' \bmod q \implies y = y'$ 
  by(simp add: y y')
ultimately show ?thesis by(simp add: map)
qed

```

```

lemma inj-on-mult:
  assumes coprime: coprime x (q::nat)
  shows inj-on ( $\lambda b. x*b \bmod q$ )  $\{..<q\}$ 
  apply(auto simp add: inj-on-def)
  using coprime by(simp only: inj-mult)

```

```

lemma surj-on-mult:
  assumes coprime: coprime x (q::nat)
  and inj: inj-on ( $\lambda b. x*b \bmod q$ )  $\{..<q\}$ 
  shows ( $\lambda b. x*b \bmod q$ ) '  $\{..<q\} = \{..<q\}$ 
  apply(rule endo-inj-surj) using coprime inj by auto

```

```

lemma mult-one-time-pad:
  assumes coprime: coprime x q
  shows map-spmf ( $\lambda b. x*b \bmod q$ ) (sample-uniform q) = (sample-uniform q)
  using inj-on-mult surj-on-mult one-time-pad coprime by simp

```

The multiplication map for sampling from units.

```

lemma inj-on-mult-units:
  assumes 1: coprime x (q::nat) shows inj-on ( $\lambda b. x*b \bmod q$ ) ( $\{..<q\} - \{0\}$ )
  apply(auto simp add: inj-on-def)
  using 1 by(simp only: inj-mult)

```

```

lemma surj-on-mult-units:
  assumes coprime: coprime x (q::nat)
  and inj: inj-on ( $\lambda b. x*b \bmod q$ ) ( $\{..<q\} - \{0\}$ )
  shows ( $\lambda b. x*b \bmod q$ ) ' ( $\{..<q\} - \{0\}$ ) = ( $\{..<q\} - \{0\}$ )
proof(rule endo-inj-surj)
  show finite ( $\{..<q\} - \{0\}$ ) using coprime inj by(simp)
  show ( $\lambda b. x * b \bmod q$ ) ' ( $\{..<q\} - \{0\}$ )  $\subseteq$   $\{..<q\} - \{0\}$ 
  proof -
    obtain n :: nat set  $\implies$  (nat  $\implies$  nat)  $\implies$  nat set  $\implies$  nat where
       $\forall x0 x1 x2. (\exists v3. v3 \in x2 \wedge x1 v3 \notin x0) = (n x0 x1 x2 \in x2 \wedge x1 (n x0 x1$ 
x2)  $\notin x0)$ 
    by moura
    then have subset:  $\forall N f Na. n Na f N \in N \wedge f (n Na f N) \notin Na \vee f ' N \subseteq Na$ 

```

by (*meson image-subsetI*)
have *mem-insert*: $x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \notin \{..<q\} \vee x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \in \text{insert } 0 \{..<q\}$
by *force*
have *map-eq*: $(x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \in \text{insert } 0 \{..<q\} - \{0\}) = (x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \in \{..<q\} - \{0\})$
by *simp*
{ assume $x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q = x * 0 \bmod q$
then have $(0 \leq q) = (0 = q) \vee (n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \notin \{..<q\} \vee n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \in \{0\}) \vee n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \notin \{..<q\} - \{0\} \vee x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \in \{..<q\} - \{0\}$
by (*metis antisym-conv1 insertCI lessThan-iff local.coprime inj-mult*) }
moreover
{ assume $0 \neq x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q$
moreover
{ assume $x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \in \text{insert } 0 \{..<q\} \wedge x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \notin \{0\}$
then have $(\lambda n. x * n \bmod q) \text{ ' } (\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\}$
using *map-eq subset by (meson Diff-iff)* }
ultimately have $(\lambda n. x * n \bmod q) \text{ ' } (\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\} \vee (0 \leq q) = (0 = q)$
using *mem-insert by (metis antisym-conv1 lessThan-iff mod-less-divisor singletonD)* }
ultimately have $(\lambda n. x * n \bmod q) \text{ ' } (\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\} \vee n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \notin \{..<q\} - \{0\} \vee x * n (\{..<q\} - \{0\}) (\lambda n. x * n \bmod q) (\{..<q\} - \{0\}) \bmod q \in \{..<q\} - \{0\}$
by *force*
then show $(\lambda n. x * n \bmod q) \text{ ' } (\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\}$
using *subset by meson*
qed
show *inj-on* $(\lambda b. x * b \bmod q) (\{..<q\} - \{0\})$ **using** *assms by (simp)*
qed

lemma *mult-one-time-pad-units*:

assumes *coprime*: *coprime* x q

shows *map-spmf* $(\lambda b. x * b \bmod q)$ (*sample-uniform-units* q) = *sample-uniform-units* q

using *inj-on-mult-units surj-on-mult-units one-time-pad-units coprime by simp*

Addition and multiplication map.

lemma *samp-uni-add-mult*:

assumes *coprime*: *coprime* x $(q::\text{nat})$

and xa : $xa < q$

and ya : $ya < q$

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

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

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

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

Subtraction Map.

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

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

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

lemma *surj-on-minus*:
assumes *inj*: *inj-on* ($\lambda(b :: \text{nat}). (y + (q - b)) \text{ mod } q$) $\{..<q\}$
shows ($\lambda(b :: \text{nat}). (y + (q - b)) \text{ mod } q$) ‘ $\{..<q\} = \{..<q\}$
apply(*rule endo-inj-surj*)
using *inj* **by** *auto*

lemma *samp-uni-minus-one-time-pad*:
shows *map-spmf*($\lambda b. (y + (q - b)) \text{ mod } q$) (*sample-uniform* q) = (*sample-uniform* q)
using *inj-on-minus surj-on-minus one-time-pad* **by** *simp*

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

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

end

2 Semi-Honest Security

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

2.1 Security definitions

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

2.1.1 Security for deterministic functionalities

```
locale sim-det-def =
  fixes R1 :: 'msg1  $\Rightarrow$  'msg2  $\Rightarrow$  'view1 spmf
    and S1 :: 'msg1  $\Rightarrow$  'out1  $\Rightarrow$  'view1 spmf
    and R2 :: 'msg1  $\Rightarrow$  'msg2  $\Rightarrow$  'view2 spmf
    and S2 :: 'msg2  $\Rightarrow$  'out2  $\Rightarrow$  'view2 spmf
    and funct :: 'msg1  $\Rightarrow$  'msg2  $\Rightarrow$  ('out1  $\times$  'out2) spmf
    and protocol :: 'msg1  $\Rightarrow$  'msg2  $\Rightarrow$  ('out1  $\times$  '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'  $\Rightarrow$  bool spmf

definition correctness m1 m2  $\equiv$  (protocol m1 m2 = funct m1 m2)

definition adv-P1 :: 'msg1  $\Rightarrow$  'msg2  $\Rightarrow$  'view1 adversary-det  $\Rightarrow$  real
  where adv-P1 m1 m2 D  $\equiv$  |(spmf (R1 m1 m2  $\ggg$  D) True)
    - spmf (funct m1 m2  $\ggg$  ( $\lambda$  (o1, o2). S1 m1 o1  $\ggg$  D)) True|

definition perfect-sec-P1 m1 m2  $\equiv$  (R1 m1 m2 = funct m1 m2  $\ggg$  ( $\lambda$  (s1, s2).
S1 m1 s1))

definition adv-P2 :: 'msg1  $\Rightarrow$  'msg2  $\Rightarrow$  'view2 adversary-det  $\Rightarrow$  real
  where adv-P2 m1 m2 D = |spmf (R2 m1 m2  $\ggg$  ( $\lambda$  view. D view)) True
    - spmf (funct m1 m2  $\ggg$  ( $\lambda$  (o1, o2). S2 m2 o2  $\ggg$  ( $\lambda$  view. D view)))
  True|
```

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

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

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

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

We show the two definitions are equivalent

lemma *equiv-defs-P1*:

assumes *lossless-D*: $\forall\ \text{view}. \text{lossless-spmf}\ ((D:: 'view1\ \text{adversary-det})\ \text{view})$

shows *adv-P1-game* $m1\ m2\ D = \text{adv-P1}\ m1\ m2\ D$

including *monad-normalisation*

proof –

have *return-True-not-False*: $\text{spmf}\ (\text{return-spmf}\ (b))\ \text{True} = \text{spmf}\ (\text{return-spmf}\ (\neg b))\ \text{False}$

for b **by** (*cases* b ; *auto*)

have *lossless-ideal*: $\text{lossless-spmf}\ ((\text{funct}\ m1\ m2 \gg (\lambda (out1, out2). S1\ m1\ out1) \gg (\lambda sview. D\ sview \gg (\lambda b'. \text{return-spmf}\ (\text{False} = b')))))$

by (*simp* *add*: *lossless-S1* *lossless-funct* *lossless-weight-spmfD* *split-def* *lossless-D*)

have *return*: $\text{spmf}\ (\text{funct}\ m1\ m2 \gg (\lambda (o1, o2). S1\ m1\ o1 \gg D))\ \text{True}$
 $= \text{spmf}\ (\text{funct}\ m1\ m2 \gg (\lambda (o1, o2). S1\ m1\ o1 \gg (\lambda\ \text{view}. D\ \text{view} \gg (\lambda\ b. \text{return-spmf}\ b))))\ \text{True}$

by *simp*

have $2*(\text{spmf}\ (P1\text{-game-alt}\ m1\ m2\ D)\ \text{True}) - 1 = (\text{spmf}\ (R1\ m1\ m2 \gg (\lambda rview. D\ rview \gg (\lambda (b':: \text{bool}). \text{return-spmf}\ (\text{True} = b')))))\ \text{True}$

$- (1 - (\text{spmf}\ (\text{funct}\ m1\ m2 \gg (\lambda (out1, out2). S1\ m1\ out1 \gg (\lambda sview. D\ sview \gg (\lambda b'. \text{return-spmf}\ (\text{False} = b')))))\ \text{True}))\ \text{True}$

by (*simp* *add*: *spmf-bind* *integral-spmf-of-set* *adv-P1-game-def* *P1-game-alt-def* *spmf-of-set*)

$\text{UNIV-bool}\ \text{bind-spmf-const}\ \text{lossless-R1}\ \text{lossless-S1}\ \text{lossless-funct}\ \text{lossless-weight-spmfD}$

hence *adv-P1-game* $m1\ m2\ D = |(\text{spmf}\ (R1\ m1\ m2 \gg (\lambda rview. D\ rview \gg (\lambda (b':: \text{bool}). \text{return-spmf}\ (\text{True} = b')))))\ \text{True}$

$- (1 - (\text{spmf}\ (\text{funct}\ m1\ m2 \gg (\lambda (out1, out2). S1\ m1\ out1 \gg (\lambda sview. D\ sview \gg (\lambda b'. \text{return-spmf}\ (\text{False} = b')))))\ \text{True}))\ \text{True}|$

using *adv-P1-game-def* **by** *simp*

also have $|(\text{spmf}\ (R1\ m1\ m2 \gg (\lambda rview. D\ rview \gg (\lambda (b':: \text{bool}). \text{return-spmf}\ (\text{True} = b')))))\ \text{True}$

$$- (1 - (\text{spmf } (\text{funct } m1 \ m2 \gg (\lambda(\text{out1}, \text{out2}). S1 \ m1 \ \text{out1} \gg (\lambda \text{sview}. D \ \text{sview} \gg (\lambda b'. \text{return-spmf } (\text{False} = b')))))) \ \text{True}) = \text{adv-P1 } m1 \ m2 \ D$$
apply(*simp only: adv-P1-def spmf-False-conv-True[symmetric] lossless-ideal; simp*)
by(*simp only: return*)(*simp only: split-def spmf-bind return-True-not-False*)
ultimately show *?thesis* **by** *simp*
qed

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

definition *adv-P2-game* :: 'msg1 \Rightarrow 'msg2 \Rightarrow 'view2 adversary-det \Rightarrow 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*: \forall *view*. *lossless-spmf* ((*D*:: 'view2 adversary-det) *view*)
shows *adv-P2-game* m1 m2 D = *adv-P2* m1 m2 D
including *monad-normalisation*

proof –

have *return-True-not-False*: *spmf* (*return-spmf* (*b*)) *True* = *spmf* (*return-spmf* (\neg *b*)) *False*

for *b* **by**(*cases b; auto*)

have *lossless-ideal*: *lossless-spmf* ((*funct* m1 m2 \gg ($\lambda(\text{out1}, \text{out2}). S2 \ m2 \ \text{out2} \gg (\lambda \text{sview}. D \ \text{sview} \gg (\lambda b'. \text{return-spmf } (\text{False} = b'))))))$

by(*simp add: lossless-S2 lossless-funct lossless-weight-spmfD split-def lossless-D*)

have *return*: *spmf* (*funct* m1 m2 \gg ($\lambda(o1, o2). S2 \ m2 \ o2 \gg D$)) *True* = *spmf* (*funct* m1 m2 \gg ($\lambda(o1, o2). S2 \ m2 \ o2 \gg (\lambda \text{view}. D \ \text{view} \gg (\lambda b. \text{return-spmf } b))$)) *True*

by *simp*

have

$2 * (\text{spmf } (P2\text{-game-alt } m1 \ m2 \ D) \ \text{True}) - 1 = (\text{spmf } (R2 \ m1 \ m2 \gg (\lambda rview. D \ rview \gg (\lambda(b':: \text{bool}). \text{return-spmf } (\text{True} = b')))))) \ \text{True}$

$- (1 - (\text{spmf } (\text{funct } m1 \ m2 \gg (\lambda(\text{out1}, \text{out2}). S2 \ m2 \ \text{out2} \gg (\lambda \text{sview}. D \ \text{sview} \gg (\lambda b'. \text{return-spmf } (\text{False} = b')))))) \ \text{True})$

by(*simp add: spmf-bind integral-spmf-of-set adv-P1-game-def P2-game-alt-def spmf-of-set*)

$UNIV\text{-bool } \text{bind-spmf-const } \text{lossless-R2 } \text{lossless-S2 } \text{lossless-funct } \text{lossless-weight-spmfD}$

hence *adv-P2-game* m1 m2 D = |(*spmf* (*R2* m1 m2 \gg ($\lambda rview. D \ rview \gg (\lambda(b':: \text{bool}). \text{return-spmf } (\text{True} = b')))))) \ \text{True}$


```

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

end

```

2.1.2 Security definitions for non deterministic functionalities

locale *sim-non-det-def* =

```

    fixes R1 :: 'msg1 ⇒ '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 \Rightarrow 'msg2 \Rightarrow ('view2, 'out1, 'out2) adversary-non-det \Rightarrow real

where *adv-P2 m1 m2 D* = |*spmf* (*R2 m1 m2* $\gg=$ (λ view. *D view*)) *True* - *spmf* (*funct m1 m2* $\gg=$ (λ (o1, o2). *Ideal2 m2 m1 o2* $\gg=$ (λ view. *D view*))) *True*|

definition *perfect-sec-P2 m1 m2* \equiv (*R2 m1 m2* = *funct m1 m2* $\gg=$ (λ (s1, s2). *Ideal2 m2 m1 s2*))

end

2.1.3 Secret sharing schemes

locale *secret-sharing-scheme* =

fixes *share* :: 'input-out \Rightarrow ('share \times 'share) *spmf*
and *reconstruct* :: ('share \times 'share) \Rightarrow 'input-out *spmf*
and *F* :: ('input-out \Rightarrow 'input-out \Rightarrow 'input-out *spmf*) *set*

begin

definition *sharing-correct input* \equiv (*share input* $\gg=$ (λ (s1,s2). *reconstruct* (s1,s2)))
= *return-spmf input*)

definition *correct-share-eval input1 input2* \equiv (\forall *gate-eval* \in *F*.

\exists *gate-protocol* :: ('share \times 'share) \Rightarrow ('share \times 'share) \Rightarrow ('share \times 'share) *spmf*.

share input1 $\gg=$ (λ (s1,s2). *share input2*
 $\gg=$ (λ (s3,s4). *gate-protocol* (s1,s3) (s2,s4)
 $\gg=$ (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 \times 'a) \Rightarrow bool \Rightarrow (unit \times 'a) *spmf*

where *funct-OT-12 input1* σ = *return-spmf* ((), if σ then (snd *input1*) else (fst *input1*))

lemma *lossless-funct-OT-12*: *lossless-spmf* (*funct-OT-12* *msgs* σ)

by(*simp add: funct-OT-12-def*)

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

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

end

2.3 ETP definitions

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

theory *ETP imports*
CryptHOL.CryptHOL
begin

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

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

locale *etp* =

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

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

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

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

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

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

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

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

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

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

lemma *uni-set-samp*:
assumes $y \in \text{set-spmf } I$
shows $\text{map-spmf } (\lambda x. F \text{ (fst } y) x) (S \text{ (fst } y)) = (S \text{ (fst } y))$
(is ?lhs = ?rhs)

proof –

have *rhs*: $?rhs = \text{spmof-of-set (range (fst } y))$
unfolding *S-def* **by**(*simp*)
also have $\text{map-spmf } (\lambda x. F \text{ (fst } y) x) (\text{spmof-of-set (range (fst } y))) = \text{spmof-of-set } ((\lambda x. F \text{ (fst } y) x) \text{ ` (range (fst } y)))$
using *f-inj-on assms*
by (*metis map-spmf-of-set-inj-on*)
also have $(\lambda x. F \text{ (fst } y) x) \text{ ` (range (fst } y)) = \text{range (fst } y)$
apply(*rule endo-inj-surj*)
using *bij-betw*
by (*auto simp add: bij-betw-def dom-eq-ran f-inj-on bij-betw finite-range assms*)

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

qed

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

definition *HCP-game* :: $('index, 'range) \text{advP2} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow ('index, 'range) \text{dist2} \Rightarrow \text{bool} \text{ spmf}$
where *HCP-game* $A = (\lambda \sigma b_{\sigma} D. \text{do } \{$
 $(\alpha, \tau) \leftarrow I;$
 $x \leftarrow S \alpha;$
 $b' \leftarrow A \alpha \sigma b_{\sigma} D x;$

```

let b = B  $\alpha$  (Finv  $\alpha$   $\tau$  x);
return-spmf (b = b')

```

definition *HCP-adv* A σ b _{σ} D = |((*spmf* (HCP-game A σ b _{σ} D) True) - 1/2)|

end

end

2.4 Oblivious transfer constructed from ETPs

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

```

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

```

```

type-synonym 'range viewP1 = ((bool  $\times$  bool)  $\times$  'range  $\times$  'range) spmf
type-synonym 'range dist1 = ((bool  $\times$  bool)  $\times$  'range  $\times$  'range)  $\Rightarrow$  bool spmf
type-synonym 'index viewP2 = (bool  $\times$  'index  $\times$  (bool  $\times$  bool)) spmf
type-synonym 'index dist2 = (bool  $\times$  'index  $\times$  bool  $\times$  bool)  $\Rightarrow$  bool spmf
type-synonym ('index, 'range) advP2 = 'index  $\Rightarrow$  bool  $\Rightarrow$  bool  $\Rightarrow$  'index dist2  $\Rightarrow$ 
'range  $\Rightarrow$  bool spmf

```

```

lemma if-False-True: (if x then False else  $\neg$  False)  $\longleftrightarrow$  (if x then False else True)
  by simp

```

```

lemma if-then-True [simp]: (if b then True else x)  $\longleftrightarrow$  ( $\neg$  b  $\longrightarrow$  x)
  by simp

```

```

lemma if-else-True [simp]: (if b then x else True)  $\longleftrightarrow$  (b  $\longrightarrow$  x)
  by simp

```

```

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

```

```

locale ETP-base = etp I domain range F Finv B
  for I :: ('index  $\times$  'trap) spmf — samples index and trapdoor
  and domain :: 'index  $\Rightarrow$  'range set
  and range :: 'index  $\Rightarrow$  'range set
  and B :: 'index  $\Rightarrow$  'range  $\Rightarrow$  bool — hard core predicate
  and F :: 'index  $\Rightarrow$  'range  $\Rightarrow$  'range
  and Finv :: 'index  $\Rightarrow$  'trap  $\Rightarrow$  'range  $\Rightarrow$  '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 –

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

Party 1 views

definition *R1* :: (bool × bool) ⇒ bool ⇒ 'range viewP1
where *R1* *input*₁ σ = do {
 let (b₀, b₁) = *input*₁;
 (α, τ) ← I;
 x_σ ← etp.S α;
 y_{σ'} ← etp.S α;
 let y_σ = F α x_σ;
 return-spmf ((b₀, b₁), if σ then y_{σ'} else y_σ, if σ then y_σ else y_{σ'})}

lemma *lossless-R1*: *lossless-spmf* (*R1* *msgs* σ)

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

definition *S1* :: (bool × bool) ⇒ unit ⇒ 'range viewP1

where *S1* == (λ *input*₁ (). do {
 let (b₀, b₁) = *input*₁;
 (α, τ) ← I;
 y₀ :: 'range ← etp.S α;
 y₁ ← etp.S α;
 return-spmf ((b₀, b₁), y₀, y₁)}

lemma *lossless-S1*: *lossless-spmf* (*S1* *msgs* ())

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

Party 2 views

definition $R2 :: (bool \times bool) \Rightarrow bool \Rightarrow 'index\ viewP2$

where $R2\ msgs\ \sigma = do \{$
 $let\ (b0, b1) = msgs;$
 $(\alpha, \tau) \leftarrow I;$
 $x_\sigma \leftarrow etp.S\ \alpha;$
 $y_{\sigma'} \leftarrow etp.S\ \alpha;$
 $let\ y_\sigma = F\ \alpha\ x_\sigma;$
 $let\ x_\sigma = F_{inv}\ \alpha\ \tau\ y_\sigma;$
 $let\ x_{\sigma'} = F_{inv}\ \alpha\ \tau\ y_{\sigma'};$
 $let\ \beta_\sigma = (B\ \alpha\ x_\sigma) \oplus (if\ \sigma\ then\ b1\ else\ b0);$
 $let\ \beta_{\sigma'} = (B\ \alpha\ x_{\sigma'}) \oplus (if\ \sigma\ then\ b0\ else\ b1);$
 $return\ -\ spmf\ (\sigma, \alpha, (\beta_\sigma, \beta_{\sigma'}))\}$

lemma $lossless-R2: lossless\ -\ spmf\ (R2\ msgs\ \sigma)$

by $(simp\ add: R2\ -\ def\ split\ -\ def\ local.\ etp.\ lossless\ -\ I\ etp.\ lossless\ -\ S)$

definition $S2 :: bool \Rightarrow bool \Rightarrow 'index\ viewP2$

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

lemma $lossless-S2: lossless\ -\ spmf\ (S2\ \sigma\ b_\sigma)$

by $(simp\ add: S2\ -\ def\ local.\ etp.\ lossless\ -\ I\ etp.\ lossless\ -\ S\ split\ -\ def)$

Security for Party 1

We have information theoretic security for Party 1.

lemma $P1\ -\ security: R1\ input_1\ \sigma = funct\ -\ OT\ -\ 12\ x\ y \ggg (\lambda (s1, s2). S1\ input_1\ s1)$

including $monad\ -\ normalisation$

proof –

have $R1\ input_1\ \sigma = do \{$
 $let\ (b0, b1) = input_1;$
 $(\alpha, \tau) \leftarrow I;$
 $y_{\sigma'} :: 'range \leftarrow etp.S\ \alpha;$
 $y_\sigma \leftarrow map\ -\ spmf\ (\lambda x_\sigma. F\ \alpha\ x_\sigma)\ (etp.S\ \alpha);$
 $return\ -\ spmf\ ((b0, b1), if\ \sigma\ then\ y_{\sigma'}\ else\ y_\sigma, if\ \sigma\ then\ y_\sigma\ else\ y_{\sigma'})\}$
by $(simp\ add: bind\ -\ map\ -\ spmf\ o\ -\ def\ Let\ -\ def\ R1\ -\ def)$

also have $\dots = do \{$
 $let\ (b0, b1) = input_1;$
 $(\alpha, \tau) \leftarrow I;$
 $y_{\sigma'} :: 'range \leftarrow etp.S\ \alpha;$
 $y_\sigma \leftarrow etp.S\ \alpha;$

$\text{return-spmf } ((b0, b1), \text{if } \sigma \text{ then } y_{\sigma'} \text{ else } y_{\sigma}, \text{if } \sigma \text{ then } y_{\sigma} \text{ else } y_{\sigma'})$
 $\text{by}(\text{simp add: etp.uni-set-samp Let-def split-def cong: bind-spmf-cong})$
also have $\dots = \text{funct-OT-12 } x \ y \gg (\lambda (s1, s2). S1 \text{ input}_1 \ s1)$
 $\text{by}(\text{cases } \sigma; \text{simp add: S1-def R1-def Let-def funct-OT-12-def})$
ultimately show *?thesis* **by auto**
qed

The adversary used in proof of security for party 2

definition $\mathcal{A} :: ('index, 'range) \text{advP2}$
where $\mathcal{A} \ \alpha \ \sigma \ b_{\sigma} \ D2 \ x = \text{do } \{$
 $\beta_{\sigma'} \leftarrow \text{coin-spmf};$
 $x_{\sigma} \leftarrow \text{etp.S } \alpha;$
 $\text{let } \beta_{\sigma} = (B \ \alpha \ x_{\sigma}) \oplus b_{\sigma};$
 $d \leftarrow D2(\sigma, \alpha, \beta_{\sigma}, \beta_{\sigma'});$
 $\text{return-spmf}(\text{if } d \text{ then } \beta_{\sigma'} \text{ else } \neg \beta_{\sigma}')\}$

lemma lossless-A:

assumes $\forall \text{ view. lossless-spmf } (D2 \ \text{view})$
shows $y \in \text{set-spmf } I \longrightarrow \text{lossless-spmf } (\mathcal{A} \ (\text{fst } y) \ \sigma \ b_{\sigma} \ D2 \ x)$
 $\text{by}(\text{simp add: A-def etp.lossless-S assms})$

lemma assem-bound-funct-OT-12:

assumes $\text{etp.HCP-adv } \mathcal{A} \ \sigma \ (\text{if } \sigma \text{ then } b1 \text{ else } b0) \ D \leq \text{HCP-ad}$
shows $|\text{spmf } (\text{funct-OT-12 } (b0, b1) \ \sigma \gg (\lambda (out1, out2). \text{etp.HCP-game } \mathcal{A} \ \sigma \ out2 \ D)) \ \text{True} - 1/2| \leq \text{HCP-ad}$
 $(\text{is } ?lhs \leq \text{HCP-ad})$

proof –

have $?lhs = |\text{spmf } (\text{etp.HCP-game } \mathcal{A} \ \sigma \ (\text{if } \sigma \text{ then } b1 \text{ else } b0) \ D) \ \text{True} - 1/2|$
 $\text{by}(\text{simp add: funct-OT-12-def})$
thus *?thesis* **using** *assms* etp.HCP-adv-def **by simp**

qed

lemma assem-bound-funct-OT-12-collapse:

assumes $\forall b_{\sigma}. \text{etp.HCP-adv } \mathcal{A} \ \sigma \ b_{\sigma} \ D \leq \text{HCP-ad}$
shows $|\text{spmf } (\text{funct-OT-12 } m1 \ \sigma \gg (\lambda (out1, out2). \text{etp.HCP-game } \mathcal{A} \ \sigma \ out2 \ D)) \ \text{True} - 1/2| \leq \text{HCP-ad}$
using *assem-bound-funct-OT-12 surj-pair assms* **by metis**

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

lemma R2-S2-False:

assumes $((\text{if } \sigma \text{ then } b0 \text{ else } b1) = \text{False})$
shows $\text{spmf } (R2 \ (b0, b1) \ \sigma \gg (D2 :: (\text{bool} \times 'index \times \text{bool} \times \text{bool}) \Rightarrow \text{bool} \ \text{spmf})) \ \text{True}$
 $= \text{spmf } (\text{funct-OT-12 } (b0, b1) \ \sigma \gg (\lambda (out1, out2). S2 \ \sigma \ out2 \gg D2)) \ \text{True}$

proof –

have $\sigma \Rightarrow \neg b0$ **using** *assms* **by simp**
moreover have $\neg \sigma \Rightarrow \neg b1$ **using** *assms* **by simp**

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

lemma *R2-S2-True:*

assumes $((\text{if } \sigma \text{ then } b0 \text{ else } b1) = \text{True})$
and *lossless-D: $\forall a. \text{lossless-spmf } (D2 a)$*
shows $|(\text{spmf } (\text{bind-spmf } (R2 (b0,b1) \sigma) D2) \text{True}) - \text{spmf } (\text{funct-OT-12 } (b0,b1) \sigma \gg (\lambda (out1, out2). S2 \sigma out2 \gg (\lambda \text{view}. D2 \text{view}))) \text{True}|$
 $= |2 * ((\text{spmf } (\text{etp.HCP-game } \mathcal{A} \sigma (\text{if } \sigma \text{ then } b1 \text{ else } b0) D2)$
 $\text{True}) - 1/2)|$

proof–

have $(\text{spmf } (\text{funct-OT-12 } (b0,b1) \sigma \gg (\lambda (out1, out2). S2 \sigma out2 \gg D2))$
 True
 $- \text{spmf } (\text{bind-spmf } (R2 (b0,b1) \sigma) D2) \text{True})$
 $= 2 * ((\text{spmf } (\text{etp.HCP-game } \mathcal{A} \sigma (\text{if } \sigma \text{ then } b1 \text{ else } b0) D2)$
 $\text{True}) - 1/2)$

proof–

have $((\text{spmf } (\text{etp.HCP-game } \mathcal{A} \sigma (\text{if } \sigma \text{ then } b1 \text{ else } b0) D2) \text{True}) - 1/2)$
 $=$

$$1/2 * (\text{spmf } (\text{bind-spmf } (S2 \sigma (\text{if } \sigma \text{ then } b1 \text{ else } b0)) D2) \text{True} - \text{spmf } (\text{bind-spmf } (R2 (b0,b1) \sigma) D2) \text{True})$$

including *monad-normalisation*

proof–

have $\sigma\text{-true-}b0\text{-true}: \sigma \implies b0 = \text{True}$ **using** *assms(1)* **by** *simp*

have $\sigma\text{-false-}b1\text{-true}: \neg \sigma \implies b1$ **using** *assms(1)* **by** *simp*

have *return-True-False: $\text{spmf } (\text{return-spmf } (\neg d)) \text{True} = \text{spmf } (\text{return-spmf } d) \text{False}$*

d) False

for *d* **by**(*cases d; simp*)

define *HCP-game-true* **where** $\text{HCP-game-true} == \lambda \sigma b_\sigma. \text{do } \{$

$(\alpha, \tau) \leftarrow I;$
 $x_\sigma \leftarrow \text{etp.S } \alpha;$
 $x \leftarrow (\text{etp.S } \alpha);$
 $\text{let } \beta_\sigma = (B \alpha x_\sigma) \oplus b_\sigma;$
 $\text{let } \beta_{\sigma'} = B \alpha (F_{\text{inv}} \alpha \tau x);$
 $d \leftarrow D2(\sigma, \alpha, \beta_\sigma, \beta_{\sigma'});$
 $\text{let } b' = (\text{if } d \text{ then } \beta_{\sigma'} \text{ else } \neg \beta_{\sigma'});$
 $\text{let } b = B \alpha (F_{\text{inv}} \alpha \tau x);$
 $\text{return-spmf } (b = b')$

define *HCP-game-false* **where** $\text{HCP-game-false} == \lambda \sigma b_\sigma. \text{do } \{$

$(\alpha, \tau) \leftarrow I;$
 $x_\sigma \leftarrow \text{etp.S } \alpha;$
 $x \leftarrow (\text{etp.S } \alpha);$
 $\text{let } \beta_\sigma = (B \alpha x_\sigma) \oplus b_\sigma;$
 $\text{let } \beta_{\sigma'} = \neg B \alpha (F_{\text{inv}} \alpha \tau x);$
 $d \leftarrow D2(\sigma, \alpha, \beta_\sigma, \beta_{\sigma'});$
 $\text{let } b' = (\text{if } d \text{ then } \beta_{\sigma'} \text{ else } \neg \beta_{\sigma'});$
 $\text{let } b = B \alpha (F_{\text{inv}} \alpha \tau x);$

```

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

```

```

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

```

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

lemma *P2-adv-bound*:

assumes *lossless-D*: $\forall a. \text{lossless-spmf } (D2 a)$
shows $|(\text{spmf } (\text{bind-spmf } (R2 (b0, b1) \sigma) D2) \text{ True}) - \text{spmf } (\text{funct-OT-12 } (b0, b1) \sigma \gg (\lambda (out1, out2). S2 \sigma out2 \gg (\lambda \text{view}. D2 \text{view}))) \text{ True}|$
 $\leq |2*((\text{spmf } (\text{etp.HCP-game } \mathcal{A} \sigma \text{ (if } \sigma \text{ then } b1 \text{ else } b0)) D2) \text{ True}) - 1/2|$
by(*cases (if σ then $b0$ else $b1$); auto simp add: R2-S2-False R2-S2-True assms*)

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

unfolding *sim-det-def-def*

by(*simp add: lossless-R1 lossless-S1 lossless-R2 lossless-S2 funct-OT-12-def*)

```

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

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

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

```

We also consider the asymptotic case for security proofs

```

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

```

```

begin

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

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

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

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

end

end

```

2.4.1 RSA instantiation

It is known that the RSA collection forms an ETP. Here we instantiate 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  $\times$  nat)
type-synonym trap = nat
type-synonym range = nat
type-synonym domain = nat

```

```

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

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

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

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

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

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

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

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

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

```

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

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

lemma *set-spmf-sample-primes*:
shows *set-spmf sample-primes* \subseteq $\{x. \text{prime } x \wedge x > 2\}$
by(*auto simp add: sample-primes-def prime-set-ass finite-prime-set*)

lemma *mem-samp-primes-gt-2*:
shows $x \in \text{set-spmf } \text{sample-primes} \implies x > 2$
apply (*simp add: finite-prime-set sample-primes-def*)
using *prime-set-ass* **by** *blast*

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

definition *sample-primes-excl* :: nat set \Rightarrow nat spmf
where *sample-primes-excl* $P = \text{spmf-of-set } (\text{prime-set} - P)$

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

definition *sample-set-excl* :: nat set \Rightarrow nat set \Rightarrow nat spmf
where *sample-set-excl* $Q P = \text{spmf-of-set } (Q - P)$

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

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

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

lemma *mem-samp-primes-excl-prime* :

shows $x \in \text{set-spmf } (\text{sample-set-excl prime-set } \{y\}) \implies \text{prime } x$
apply (*simp add: finite-prime-set sample-set-excl-def*)
using *prime-set-ass* **by** *blast*

lemma *sample-coprime-lem*:

assumes $x \in \text{set-spmf } \text{sample-primes}$
and $y \in \text{set-spmf } (\text{sample-set-excl prime-set } \{x\})$
shows *lossless-spmf* (*sample-coprime* $((x - \text{Suc } 0) * (y - \text{Suc } 0))$)

proof –

have *gt-2*: $x > 2 \ y > 2$

using *mem-samp-primes-gt-2* *assms* *mem-samp-primes-excl-gt-2* **by** *auto*

have $\neg \text{prime } ((x-1)*(y-1))$

proof –

have *prime* x *prime* y

using *mem-samp-primes-prime* *mem-samp-primes-excl-prime* *assms* **by** *auto*

then show *?thesis* **using** *prod-not-prime-gt-2* **by** *simp*

qed

also have $((x-1)*(y-1)) > 2$

by (*metis* (*no-types*, *lifting*) *gt-2* *One-nat-def* *Suc-diff-1* *assms(1)* *assms(2)*)

calculation

divisors-zero less-2-cases nat-1-eq-mult-iff nat-neq-iff not-numeral-less-one

numeral-2-eq-2

prime-gt-0-nat *rsa-base.mem-samp-primes-excl-prime* *rsa-base.mem-samp-primes-prime*

rsa-base-axioms *two-is-prime-nat*)

ultimately show *?thesis* **using** *lossless-sample-coprime* **by** *simp*

qed

definition *I* :: $(\text{index} \times \text{trap}) \text{ spmf}$

where $I = \text{do } \{$

$P \leftarrow \text{sample-primes};$

$Q \leftarrow \text{sample-set-excl prime-set } \{P\};$

$\text{let } N = P * Q;$

$\text{let } N' = (P-1) * (Q-1);$

$e \leftarrow \text{sample-coprime } N';$

$\text{let } d = \text{nat } ((\text{fst } (\text{bezw } e \ N')) \bmod N');$

$\text{return-spmf } ((N, e), d)\}$

lemma *lossless-I*: *lossless-spmf* *I*

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

lemma *set-spmf-I-N*:

assumes $((N, e), d) \in \text{set-spmf } I$

obtains $P \ Q$ **where** $N = P * Q$

and $P \neq Q$

and *prime* P

and *prime* Q

```

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

```

lemma *set-spmf-I-e-d*:

```

⟨e > 1⟩ ⟨d > 1⟩ if ⟨(N, e), d⟩ ∈ set-spmf I

```

proof –

from *that* **obtain** *M* **where**

```

e: ⟨e ∈ set-spmf (sample-coprime M)⟩

```

```

and d: ⟨d = nat (fst (bezw e M) mod M)⟩

```

```

by (auto simp add: I-def Let-def)

```

from *e set-spmf-sample-coprime [of M]*

```

have ⟨coprime e M⟩ ⟨1 < e⟩ ⟨e < M⟩

```

```

by simp-all

```

```

then have ⟨2 < M⟩

```

```

by simp

```

```

from ⟨1 < e⟩ show ⟨e > 1⟩.

```

```

from d ⟨coprime e M⟩ bezw-inverse [of e M]

```

```

have ⟨[e * d = 1] (mod M)⟩

```

```

by simp

```

```

with ⟨e > 1⟩ ⟨2 < M⟩ show ⟨d > 1⟩

```

```

by (cases ⟨d = 0 ∨ d = 1⟩) (auto simp add: ⟨e < M⟩ cong-def)

```

qed

definition *domain* :: *index* ⇒ *nat set*

```

where domain index ≡ {..fst index}

```

definition *range* :: *index* ⇒ *nat set*

```

where range index ≡ {..fst index}

```

lemma *finite-range*: *finite (range index)*

```

by(simp add: range-def)

```

lemma *dom-eq-ran*: *domain index = range index*

```

by(simp add: range-def domain-def)

```

definition *F* :: *index* ⇒ (*nat* ⇒ *nat*)

```

where F index x = x ^ (snd index) mod (fst index)

```

definition *F_{inv}* :: *index* ⇒ *trap* ⇒ *nat* ⇒ *nat*

```

where Finv 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:  $P \neq Q$ 
and x-lt-pq:  $x < P * Q$ 
and y-lt-pd:  $y < P * Q$ 
and rsa-map-eq:  $x \hat{=} e \text{ mod } (P * Q) = y \hat{=} e \text{ mod } (P * Q)$ 
shows  $x = y$ 
proof –
  have flt-xP:  $[x \hat{=} P = x] \text{ (mod } P)$ 
    using fermat-little prime-P by blast
  have flt-yP:  $[y \hat{=} P = y] \text{ (mod } P)$ 
    using fermat-little prime-P by blast
  have flt-xQ:  $[x \hat{=} Q = x] \text{ (mod } Q)$ 
    using fermat-little prime-Q by blast
  have flt-yQ:  $[y \hat{=} Q = y] \text{ (mod } Q)$ 
    using fermat-little prime-Q by blast
  show ?thesis
  proof(cases y ≥ x)
    case True
      hence ye-gt-xe:  $y \hat{=} e \geq x \hat{=} e$ 
        by (simp add: power-mono)
      have x-y-exp-e:  $[x \hat{=} e = y \hat{=} e] \text{ (mod } P)$ 
        using cong-modulus-mult-nat cong-altdef-nat True ye-gt-xe cong-sym cong-def
assms by blast
      obtain d where d:  $[e*d = 1] \text{ (mod } (P-1)) \wedge d \neq 0$ 
        using ex-inverse assms by blast
      then obtain k where k:  $e*d = 1 + k*(P-1)$ 
        using ex-k-mod assms by blast
      hence xk-yk:  $[x \hat{=} (1 + k*(P-1)) = y \hat{=} (1 + k*(P-1))] \text{ (mod } P)$ 
        by(metis k power-mult x-y-exp-e cong-pow)
      have xk-x:  $[x \hat{=} (1 + k*(P-1)) = x] \text{ (mod } P)$ 
      proof(induct k)
        case 0
          then show ?case by simp
        next
          case (Suc k)
            assume asm:  $[x \hat{=} (1 + k * (P - 1)) = x] \text{ (mod } P)$ 
            then show ?case
              proof –
                have exp-rewrite:  $(k * (P - 1) + P) = (1 + (k + 1) * (P - 1))$ 
                by (smt add.assoc add.commute le-add-diff-inverse nat-le-linear not-add-less1
prime-P prime-gt-1-nat semiring-normalization-rules(3))
                have  $[x * x \hat{=} (k * (P - 1)) = x] \text{ (mod } P)$  using asm by simp
                hence  $[x \hat{=} (k * (P - 1)) * x \hat{=} P = x] \text{ (mod } P)$  using flt-xP
                  by (metis cong-scalar-right cong-trans mult.commute)
                hence  $[x \hat{=} (k * (P - 1) + P) = x] \text{ (mod } P)$ 
                  by (simp add: power-add)
                hence  $[x \hat{=} (1 + (k + 1) * (P - 1)) = x] \text{ (mod } P)$ 
                  using exp-rewrite by argo
                thus ?thesis by simp

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

lemma *rsa-bij-betw*:

```

assumes  $\text{coprime } e \ ((P - 1) * (Q - 1))$ 
and prime P
and prime Q
and  $P \neq Q$ 
shows bij-betw ( $F \ ((P * Q), e)$ ) (range ( $(P * Q), e$ )) (range ( $(P * Q), e$ ))
proof -
have  $PQ\text{-not-}0: \text{prime } P \longrightarrow \text{prime } Q \longrightarrow P * Q \neq 0$ 
using assms by auto
have inj-on ( $\lambda x. x \wedge \text{snd } (P * Q, e) \text{ mod } \text{fst } (P * Q, e) \{.. < \text{fst } (P * Q, e)\}$ )
apply (simp add: inj-on-def)
using rsa-bijection assms by blast
moreover have ( $\lambda x. x \wedge \text{snd } (P * Q, e) \text{ mod } \text{fst } (P * Q, e) \{.. < \text{fst } (P * Q, e)\}$ )
apply (simp add: assms(2) assms(3) prime-gt-0-nat PQ-not-0)
apply (rule endo-inj-surj; auto simp add: assms(2) assms(3) image-subsetI
prime-gt-0-nat PQ-not-0 inj-on-def)
using rsa-bijection assms by blast
ultimately show ?thesis
unfolding bij-betw-def F-def range-def by blast
qed

```

lemma *bij-betw1*:

```

assumes  $((N, e), d) \in \text{set-spmf } I$ 
shows bij-betw ( $F \ ((N), e)$ ) (range ( $(N), e$ )) (range ( $(N), e$ ))
proof -
obtain  $P \ Q$  where  $N = P * Q$  and bij-betw ( $F \ ((P * Q), e)$ ) (range ( $(P * Q), e$ ))
(range ( $(P * Q), e$ ))
proof -
obtain  $P \ Q$  where prime P and prime Q and  $N = P * Q$  and  $P \neq Q$  and
coprime e ((P - 1) * (Q - 1))
using set-spmf-I-N assms by blast
then show ?thesis
using rsa-bij-betw that by blast
qed
thus ?thesis by blast
qed

```

```

lemma
  assumes  $P \text{ dvd } x$ 
  shows  $[x = 0] \pmod{P}$ 
  using assms cong-def by force

lemma rsa-inv:
  assumes  $d: d = \text{nat } (\text{fst } (\text{bezv } e \ ((P-1)*(Q-1))) \text{ mod int } ((P-1)*(Q-1)))$ 
    and coprime:  $\text{coprime } e \ ((P-1)*(Q-1))$ 
    and prime-P:  $\text{prime } (P::\text{nat})$ 
    and prime-Q:  $\text{prime } Q$ 
    and P-neq-Q:  $P \neq Q$ 
    and e-gt-1:  $e > 1$ 
    and d-gt-1:  $d > 1$ 
  shows  $((x \wedge e) \text{ mod } (P*Q)) \wedge d \text{ mod } (P*Q) = x \text{ mod } (P*Q)$ 
  proof(cases  $x = 0 \vee x = 1$ )
    case True
      then show ?thesis
        by (metis assms(6) assms(7) le-simps(1) nat-power-eq-Suc-0-iff neq0-conv
          not-one-le-zero numeral-nat(7) power-eq-0-iff power-mod)
    next
      case False
        hence x-gt-1:  $x > 1$  by simp
        define  $z$  where  $z = (x \wedge e) \wedge d - x$ 
        hence z-gt-0:  $z > 0$ 
        proof-
          have  $(x \wedge e) \wedge d - x = x \wedge (e * d) - x$ 
            by (simp add: power-mult)
          also have  $\dots > 0$ 
            by (metis x-gt-1 e-gt-1 d-gt-1 le-neq-implies-less less-one linorder-not-less
              n-less-m-mult-n not-less-zero numeral-nat(7) power-increasing-iff power-one-right
              zero-less-diff)
          ultimately show ?thesis using z-def by argo
        qed
        hence  $[z = 0] \pmod{P}$ 
        proof(cases  $[x = 0] \pmod{P}$ )
          case True
            then show ?thesis
              proof -
                have  $0 \neq d * e$ 
                  by (metis (no-types) assms assms mult-is-0 not-one-less-zero)
                then show ?thesis
                  by (metis (no-types) Groups.add-ac(2) True add-diff-inverse-nat cong-def
                    cong-dvd-iff cong-mult-self-right dvd-0-right dvd-def dvd-trans mod-add-self2 more-arith-simps(5)
                    nat-diff-split power-eq-if power-mult semiring-normalization-rules(7) z-def)
              qed
          next
            case False
              have  $[e * d = 1] \pmod{(P-1)*(Q-1)}$ 

```

by (*metis d bezw-inverse coprime coprime-imp-gcd-eq-1 nat-int*)
 hence $[e * d = 1] \pmod{(P - 1)}$
 using *assms cong-modulus-mult-nat* by *blast*
 then obtain k where $k: e*d = 1 + k*(P-1)$
 using *ex-k-mod assms* by *force*
 hence $x \wedge (e * d) = x * ((x \wedge (P - 1)) \wedge k)$
 by (*metis power-add power-one-right mult commute power-mult*)
 hence $[x \wedge (e * d) = x * ((x \wedge (P - 1)) \wedge k)] \pmod{P}$
 using *cong-def* by *simp*
 moreover have $[x \wedge (P - 1) = 1] \pmod{P}$
 using *prime-P fermat-theorem False*
 by (*simp add: cong-0-iff*)
 moreover have $[x \wedge (e * d) = x * ((1) \wedge k)] \pmod{P}$
 by (*metis $\langle x \wedge (e * d) = x * (x \wedge (P - 1)) \wedge k \rangle$ calculation(2) cong-pow*
cong-scalar-left)
 hence $[x \wedge (e * d) = x] \pmod{P}$ by *simp*
 thus *?thesis* using *z-def z-gt-0*
 by (*simp add: cong-diff-iff-cong-0-nat power-mult*)
 qed
 moreover have $[z = 0] \pmod{Q}$
 proof(*cases [x = 0] (mod Q)*)
 case *True*
 then show *?thesis*
 proof –
 have $0 \neq d * e$
 by (*metis (no-types) assms mult-is-0 not-one-less-zero*)
 then show *?thesis*
 by (*metis (no-types) Groups.add-ac(2) True add-diff-inverse-nat cong-def*
cong-dvd-iff cong-mult-self-right dvd-0-right dvd-def dvd-trans mod-add-self2 more-arith-simps(5)
nat-diff-split power-eq-if power-mult semiring-normalization-rules(7) z-def)
 qed
 next
 case *False*
 have $[e * d = 1] \pmod{((P - 1) * (Q - 1))}$
 by (*metis d bezw-inverse coprime coprime-imp-gcd-eq-1 nat-int*)
 hence $[e * d = 1] \pmod{(Q - 1)}$
 using *assms cong-modulus-mult-nat mult commute* by *metis*
 then obtain k where $k: e*d = 1 + k*(Q-1)$
 using *ex-k-mod assms* by *force*
 hence $x \wedge (e * d) = x * ((x \wedge (Q - 1)) \wedge k)$
 by (*metis power-add power-one-right mult commute power-mult*)
 hence $[x \wedge (e * d) = x * ((x \wedge (Q - 1)) \wedge k)] \pmod{P}$
 using *cong-def* by *simp*
 moreover have $[x \wedge (Q - 1) = 1] \pmod{Q}$
 using *prime-Q fermat-theorem False*
 by (*simp add: cong-0-iff*)
 moreover have $[x \wedge (e * d) = x * ((1) \wedge k)] \pmod{Q}$
 by (*metis $\langle x \wedge (e * d) = x * (x \wedge (Q - 1)) \wedge k \rangle$ calculation(2) cong-pow*
cong-scalar-left)

hence $[x \wedge (e * d) = x] \pmod{Q}$ **by** *simp*
thus *?thesis using z-def z-gt-0*
by (*simp add: cong-diff-iff-cong-0-nat power-mult*)
qed
ultimately have $Q \text{ dvd } (x \wedge e) \wedge d - x$
 $P \text{ dvd } (x \wedge e) \wedge d - x$
using *z-def assms cong-0-iff by blast +*
hence $P * Q \text{ dvd } ((x \wedge e) \wedge d - x)$
using *assms divides-mult primes-coprime-nat by blast*
hence $[(x \wedge e) \wedge d = x] \pmod{P * Q}$
using *z-gt-0 cong-altdef-nat z-def by auto*
thus *?thesis*
by (*simp add: cong-def power-mod*)
qed

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

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

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

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

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

```

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

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

end

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

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

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

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

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

end

end

```

2.5 Noar Pinkas OT

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

theory *Noar-Pinkas-OT* **imports**

Cyclic-Group-Ext
Game-Based-Crypto.Diffie-Hellman
OT-Functionalities
Semi-Honest-Def
Uniform-Sampling

begin

locale *np-base* =

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

begin

lemma *prime-field: a < (order* \mathcal{G}) $\implies a \neq 0 \implies \text{coprime } a$ (*order* \mathcal{G})

by(*metis dvd-imp-le neq0-conv not-le prime-imp-coprime prime-order coprime-commute*)

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

using *lossless-spmf-def lossless-sample-uniform-units prime-order prime-gt-1-nat*
by *auto*

definition *protocol* :: ('grp × 'grp) \Rightarrow bool \Rightarrow (unit × 'grp) *spmf*

where *protocol* M $v = \text{do}$ {

let ($m0, m1$) = M ;

$a :: \text{nat} \leftarrow \text{sample-uniform}$ (*order* \mathcal{G});

$b :: \text{nat} \leftarrow \text{sample-uniform}$ (*order* \mathcal{G});

let $c_v = (a * b) \bmod (\text{order } \mathcal{G})$;

$c_v' :: \text{nat} \leftarrow \text{sample-uniform}$ (*order* \mathcal{G});

$r0 :: \text{nat} \leftarrow \text{sample-uniform-units}$ (*order* \mathcal{G});

$s0 :: \text{nat} \leftarrow \text{sample-uniform-units}$ (*order* \mathcal{G});

let $w0 = (\mathbf{g} [\wedge] a) [\wedge] s0 \otimes \mathbf{g} [\wedge] r0$;

let $z0' = ((\mathbf{g} [\wedge] (\text{if } v \text{ then } c_v' \text{ else } c_v)) [\wedge] s0) \otimes ((\mathbf{g} [\wedge] b) [\wedge] r0)$;

$r1 :: \text{nat} \leftarrow \text{sample-uniform-units}$ (*order* \mathcal{G});

$s1 :: \text{nat} \leftarrow \text{sample-uniform-units}$ (*order* \mathcal{G});

let $w1 = (\mathbf{g} [\wedge] a) [\wedge] s1 \otimes \mathbf{g} [\wedge] r1$;

let $z1' = ((\mathbf{g} [\wedge] ((\text{if } v \text{ then } c_v \text{ else } c_v'))) [\wedge] s1) \otimes ((\mathbf{g} [\wedge] b) [\wedge] r1)$;

let $\text{enc-}m0 = z0' \otimes m0$;

let $\text{enc-}m1 = z1' \otimes m1$;

let $\text{out-}2 = (\text{if } v \text{ then } \text{enc-}m1 \otimes \text{inv } (w1 [\wedge] b) \text{ else } \text{enc-}m0 \otimes \text{inv } (w0 [\wedge] b))$;

return-spmf (($\text{out-}2$))

lemma *lossless-protocol: lossless-spmf* (*protocol* M σ)

apply(*auto simp add: protocol-def Let-def split-def lossless-sample-uniform-units or-gt-0*)

using prime-order prime-gt-1-nat lossless-sample-uniform-units by simp

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

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

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

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

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

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

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

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

fun inter-S1-adversary :: 'grp dist-adversary ⇒ ('grp × 'grp) ⇒ 'grp ⇒ 'grp ⇒ 'grp ⇒ bool spmf
where inter-S1-adversary \mathcal{A} msgs α β γ = do {

$c \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $\mathcal{A}(\text{msgs}, \alpha, \beta, \mathbf{g}[\ulcorner c, \gamma])$

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

definition $R2 :: ('grp \times 'grp) \Rightarrow \text{bool} \Rightarrow (\text{bool} \times 'grp \times 'grp \times 'grp \times 'grp \times 'grp \times 'grp \times 'grp) \text{ spmf}$

where $R2 M v = \text{do} \{$
 $\text{let } (m0, m1) = M;$
 $a :: \text{nat} \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $b :: \text{nat} \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $\text{let } c_v = (a*b) \text{ mod }(\text{order } \mathcal{G});$
 $c_v' :: \text{nat} \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $r0 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $s0 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $\text{let } w0 = (\mathbf{g}[\ulcorner a] [\ulcorner s0] \otimes \mathbf{g}[\ulcorner r0];$
 $\text{let } z = ((\mathbf{g}[\ulcorner c_v'] [\ulcorner s0]) \otimes ((\mathbf{g}[\ulcorner b] [\ulcorner r0]);$
 $r1 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $s1 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $\text{let } w1 = (\mathbf{g}[\ulcorner a] [\ulcorner s1] \otimes \mathbf{g}[\ulcorner r1];$
 $\text{let } z' = ((\mathbf{g}[\ulcorner (c_v)] [\ulcorner s1]) \otimes ((\mathbf{g}[\ulcorner b] [\ulcorner r1];$
 $\text{let } \text{enc-m} = z \otimes (\text{if } v \text{ then } m0 \text{ else } m1);$
 $\text{let } \text{enc-m}' = z' \otimes (\text{if } v \text{ then } m1 \text{ else } m0);$
 $\text{return-spmf}(v, \mathbf{g}[\ulcorner a], \mathbf{g}[\ulcorner b], \mathbf{g}[\ulcorner c_v], w0, \text{enc-m}, w1, \text{enc-m}')\}$

lemma lossless-R2 : $\text{lossless-spmf}(R2 M \sigma)$

apply($\text{simp add: R2-def Let-def split-def lossless-sample-uniform-units or-gt-0}$)
using $\text{prime-order prime-gt-1-nat lossless-sample-uniform-units by simp}$

definition $S2 :: \text{bool} \Rightarrow 'grp \Rightarrow (\text{bool} \times 'grp \times 'grp \times 'grp \times 'grp \times 'grp \times 'grp \times 'grp) \times 'grp) \text{ spmf}$

where $S2 v m = \text{do} \{$
 $a :: \text{nat} \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $b :: \text{nat} \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $\text{let } c_v = (a*b) \text{ mod }(\text{order } \mathcal{G});$
 $r0 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $s0 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $\text{let } w0 = (\mathbf{g}[\ulcorner a] [\ulcorner s0] \otimes \mathbf{g}[\ulcorner r0];$
 $r1 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $s1 :: \text{nat} \leftarrow \text{sample-uniform-units}(\text{order } \mathcal{G});$
 $\text{let } w1 = (\mathbf{g}[\ulcorner a] [\ulcorner s1] \otimes \mathbf{g}[\ulcorner r1];$
 $\text{let } z' = ((\mathbf{g}[\ulcorner (c_v)] [\ulcorner s1]) \otimes ((\mathbf{g}[\ulcorner b] [\ulcorner r1];$
 $s' \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$
 $\text{let } \text{enc-m} = \mathbf{g}[\ulcorner s'];$
 $\text{let } \text{enc-m}' = z' \otimes m;$
 $\text{return-spmf}(v, \mathbf{g}[\ulcorner a], \mathbf{g}[\ulcorner b], \mathbf{g}[\ulcorner c_v], w0, \text{enc-m}, w1, \text{enc-m}')\}$

lemma lossless-S2 : $\text{lossless-spmf}(S2 \sigma \text{ out2})$

apply($\text{simp add: S2-def Let-def lossless-sample-uniform-units or-gt-0}$)

using *prime-order prime-gt-1-nat lossless-sample-uniform-units* by *simp*

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

unfolding *sim-det-def-def*

by(*auto simp add: lossless-R1 lossless-S1 lossless-R2 lossless-S2 lossless-protocol lossless-funct-OT-12*)

end

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

begin

lemma *protocol-inverse*:

assumes $m0 \in \text{carrier } \mathcal{G}$ $m1 \in \text{carrier } \mathcal{G}$

shows $((\mathbf{g} [\uparrow] ((a*b) \text{ mod } (\text{order } \mathcal{G}))) [\uparrow] (s1 :: \text{nat})) \otimes ((\mathbf{g} [\uparrow] b) [\uparrow] (r1 :: \text{nat}))$
 $\otimes (\text{if } v \text{ then } m0 \text{ else } m1) \otimes \text{inv } (((\mathbf{g} [\uparrow] a) [\uparrow] s1 \otimes \mathbf{g} [\uparrow] r1) [\uparrow] b)$
 $= (\text{if } v \text{ then } m0 \text{ else } m1)$

(is ?lhs = ?rhs)

proof –

have $1: (a*b)*(s1) + b*r1 = ((a::\text{nat})*(s1) + r1)*b$ **using** *mult.commute mult.assoc add-mult-distrib* **by** *auto*

have *?lhs =*

$((\mathbf{g} [\uparrow] (a*b)) [\uparrow] s1) \otimes ((\mathbf{g} [\uparrow] b) [\uparrow] r1) \otimes (\text{if } v \text{ then } m0 \text{ else } m1) \otimes \text{inv } (((\mathbf{g} [\uparrow] a) [\uparrow] s1 \otimes \mathbf{g} [\uparrow] r1) [\uparrow] b)$

by(*simp add: pow-generator-mod*)

also have $\dots = (\mathbf{g} [\uparrow] ((a*b)*(s1))) \otimes ((\mathbf{g} [\uparrow] (b*r1))) \otimes ((\text{if } v \text{ then } m0 \text{ else } m1) \otimes \text{inv } (((\mathbf{g} [\uparrow] ((a*(s1) + r1)*b))))$

by(*auto simp add: nat-pow-pow nat-pow-mult assms cyclic-group-assoc*)

also have $\dots = \mathbf{g} [\uparrow] ((a*b)*(s1)) \otimes \mathbf{g} [\uparrow] (b*r1) \otimes ((\text{inv } (((\mathbf{g} [\uparrow] ((a*(s1) + r1)*b)))) \otimes (\text{if } v \text{ then } m0 \text{ else } m1))$

by(*simp add: nat-pow-mult cyclic-group-commute assms*)

also have $\dots = (\mathbf{g} [\uparrow] ((a*b)*(s1) + b*r1) \otimes \text{inv } (((\mathbf{g} [\uparrow] ((a*(s1) + r1)*b)))) \otimes (\text{if } v \text{ then } m0 \text{ else } m1)$

by(*simp add: nat-pow-mult cyclic-group-assoc assms*)

also have $\dots = (\mathbf{g} [\uparrow] ((a*b)*(s1) + b*r1) \otimes \text{inv } (((\mathbf{g} [\uparrow] (((a*b)*(s1) + r1)*b)))) \otimes (\text{if } v \text{ then } m0 \text{ else } m1)$

using 1 **by** (*simp add: mult.commute*)

ultimately show *?thesis*

using *l-cancel-inv assms* **by** (*simp add: mult.commute*)

qed

lemma *correctness*:

assumes $m0 \in \text{carrier } \mathcal{G}$ $m1 \in \text{carrier } \mathcal{G}$

shows *sim-def.correctness (m0,m1) σ*

proof –

have *protocol (m0, m1) σ = funct-OT-12 (m0, m1) σ*

proof –

have *protocol (m0, m1) σ = do {*

a :: nat ← sample-uniform (order G);

```

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

```

lemma *security-P1*:

```

  shows sim-def.adv-P1 msgs  $\sigma$   $D \leq$  ddh.advantage (R1-inter-adversary  $D$  msgs)
+ ddh.advantage (inter-S1-adversary  $D$  msgs)
  (is ?lhs  $\leq$  ?rhs)

```

proof(cases σ)

case *True*

have *R1 msgs $\sigma = S1 msgs out1$ for $out1$*

by(simp add: R1-def S1-def True)

then have *sim-def.adv-P1 msgs $\sigma D = 0$*

by(simp add: sim-def.adv-P1-def funct-OT-12-def)

also have *ddh.advantage $A \geq 0$ for A using ddh.advantage-def* by simp

ultimately show ?thesis by simp

next

case *False*

have *bounded-advantage: $|a :: real - b| = e1 \implies |b - c| = e2 \implies |a - c| \leq e1 + e2$*

for *a b e1 c e2* by simp

also have *R1-inter-dist: $|spmf (R1 msgs False \gg D) True - spmf ((inter msgs) \gg D) True| = ddh.advantage (R1-inter-adversary D msgs)$*

unfolding *R1-def inter-def ddh.advantage-def ddh.ddh-0-def ddh.ddh-1-def Let-def split-def* by(simp)

also have *inter-S1-dist: $|spmf ((inter msgs) \gg D) True - spmf (S1 msgs out1 \gg D) True| = ddh.advantage (inter-S1-adversary D msgs)$*

for *out1* including *monad-normalisation*

by(simp add: S1-def inter-def ddh.advantage-def ddh.ddh-0-def ddh.ddh-1-def)

ultimately have *$|spmf (R1 msgs False \gg (\lambda view. D view)) True - spmf (S1 msgs out1 \gg (\lambda view. D view)) True| \leq ?rhs$*

for *out1* using *R1-inter-dist* by auto

thus ?thesis by(simp add: sim-def.adv-P1-def funct-OT-12-def False)

qed

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

lemma *security-P2*:
assumes $m0 \in \text{carrier } \mathcal{G}$ $m1 \in \text{carrier } \mathcal{G}$
shows *sim-def.perfect-sec-P2* $(m0, m1) \sigma$
proof –
have $R2 (m0, m1) \sigma = S2 \sigma$ (*if σ then $m1$ else $m0$*)
including *monad-normalisation*
proof –
have $R2 (m0, m1) \sigma = \text{do } \{$
 $a :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $b :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $\text{let } c_v = (a * b) \text{ mod } (\text{order } \mathcal{G});$
 $c_v' :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $r0 :: \text{nat} \leftarrow \text{sample-uniform-units } (\text{order } \mathcal{G});$
 $s0 :: \text{nat} \leftarrow \text{sample-uniform-units } (\text{order } \mathcal{G});$
 $\text{let } w0 = (\mathbf{g} [\wedge] a) [\wedge] s0 \otimes \mathbf{g} [\wedge] r0;$
 $\text{let } s' = (((b * r0) + ((c_v') * (s0))) \text{ mod } (\text{order } \mathcal{G}));$
 $\text{let } z = \mathbf{g} [\wedge] s';$
 $r1 :: \text{nat} \leftarrow \text{sample-uniform-units } (\text{order } \mathcal{G});$
 $s1 :: \text{nat} \leftarrow \text{sample-uniform-units } (\text{order } \mathcal{G});$
 $\text{let } w1 = (\mathbf{g} [\wedge] a) [\wedge] s1 \otimes \mathbf{g} [\wedge] r1;$
 $\text{let } z' = ((\mathbf{g} [\wedge] (c_v)) [\wedge] s1) \otimes ((\mathbf{g} [\wedge] b) [\wedge] r1);$
 $\text{let } \text{enc-m} = z \otimes (\text{if } \sigma \text{ then } m0 \text{ else } m1);$
 $\text{let } \text{enc-m}' = z' \otimes (\text{if } \sigma \text{ then } m1 \text{ else } m0);$
 $\text{return-spmf}(\sigma, \mathbf{g} [\wedge] a, \mathbf{g} [\wedge] b, \mathbf{g} [\wedge] c_v, w0, \text{enc-m}, w1, \text{enc-m}')$
by (*simp add: R2-def nat-pow-pow nat-pow-mult pow-generator-mod add commute*)

also have $\dots = \text{do } \{$
 $a :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $b :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $\text{let } c_v = (a * b) \text{ mod } (\text{order } \mathcal{G});$
 $r0 :: \text{nat} \leftarrow \text{sample-uniform-units } (\text{order } \mathcal{G});$
 $s0 :: \text{nat} \leftarrow \text{sample-uniform-units } (\text{order } \mathcal{G});$
 $\text{let } w0 = (\mathbf{g} [\wedge] a) [\wedge] s0 \otimes \mathbf{g} [\wedge] r0;$
 $s' \leftarrow \text{map-spmf } (\lambda c_v'. (((b * r0) + ((c_v') * (s0))) \text{ mod } (\text{order } \mathcal{G}))) (\text{sample-uniform } (\text{order } \mathcal{G}));$
 $\text{let } z = \mathbf{g} [\wedge] s';$

```

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

```

```

    s1 :: nat ← sample-uniform-units (order G);
    let w1 = (g [∧] a) [∧] s1 ⊗ g [∧] r1;
    let z' = ((g [∧] (c_v)) [∧] s1) ⊗ ((g [∧] b) [∧] r1);
    enc-m ← map-spmf (λ s'. g [∧] s') (sample-uniform (order G));
    let enc-m' = z' ⊗ (if σ then m1 else m0) ;
    return-spmf(σ, g [∧] a, g [∧] b, g [∧] c_v, w0, enc-m, w1, enc-m')
  by(simp add: sample-uniform-one-time-pad assms)
ultimately show ?thesis by(simp add: S2-def Let-def bind-map-spmf o-def)
qed
thus ?thesis
  by(simp add: sim-def.perfect-sec-P2-def funct-OT-12-def)
qed

end

locale np-asymp =
  fixes G :: security ⇒ 'grp cyclic-group
  assumes np: ∧η. np (G η)
begin

sublocale np G η for η by(simp add: np)

theorem correctness-asymp:
  assumes m0 ∈ carrier (G η) m1 ∈ carrier (G η)
  shows sim-def.correctness η (m0, m1) σ
  by(simp add: correctness assms)

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

theorem security-P2-asymp:
  assumes m0 ∈ carrier (G η) m1 ∈ carrier (G η)
  shows sim-def.perfect-sec-P2 η (m0,m1) σ
  by(simp add: security-P2 assms)

```


lemma *OT-12-P1-assms-bound'*: $| \text{spmf} (\text{bind-spmf} (R1-OT12 (m0,m1) c) (\lambda \text{view.} ((D::'v-OT121 \Rightarrow \text{bool spmf}) \text{view}))) \text{True} - \text{spmf} (\text{bind-spmf} (S1-OT12 (m0,m1) ()) (\lambda \text{view.} (D \text{view}))) \text{True} | \leq \text{adv-OT12}$

proof –

have *sim-det-def.adv-P1 R1-OT12 S1-OT12 funct-OT-12* $(m0,m1) c D = | \text{spmf} (\text{bind-spmf} (R1-OT12 (m0,m1) c) (\lambda \text{view.} (D \text{view}))) \text{True} - \text{spmf} (\text{funct-OT-12} (m0,m1) c \gg (\lambda ((out1::\text{unit}), (out2::\text{bool})). S1-OT12 (m0,m1) out1 \gg (\lambda \text{view.} D \text{view}))) \text{True} |$

using *sim-det-def.adv-P1-def*

using *OT-12-sim.adv-P1-def* **by** *auto*

also have $\dots = | \text{spmf} (\text{bind-spmf} (R1-OT12 (m0,m1) c) (\lambda \text{view.} ((D::'v-OT121 \Rightarrow \text{bool spmf}) \text{view}))) \text{True} - \text{spmf} (\text{bind-spmf} (S1-OT12 (m0,m1) ()) (\lambda \text{view.} (D \text{view}))) \text{True} |$

by(*simp add: funct-OT-12-def*)

ultimately show *?thesis*

by(*metis ass-adv-OT12*)

qed

lemma *OT-12-P2-assm*: $R2-OT12 (m0,m1) \sigma = \text{funct-OT-12} (m0,m1) \sigma \gg (\lambda (out1, out2). S2-OT12 \sigma out2)$

using *inf-th-OT12-P2 OT-12-sim.perfect-sec-P2-def* **by** *blast*

definition *protocol-14-OT* :: $\text{input1} \Rightarrow \text{input2} \Rightarrow (\text{unit} \times \text{bool}) \text{spmf}$

where *protocol-14-OT M C* = *do* {

let $(c0,c1) = C$;

let $(m00, m01, m10, m11) = M$;

$S0 \leftarrow \text{coin-spmf}$;

$S1 \leftarrow \text{coin-spmf}$;

$S2 \leftarrow \text{coin-spmf}$;

$S3 \leftarrow \text{coin-spmf}$;

$S4 \leftarrow \text{coin-spmf}$;

$S5 \leftarrow \text{coin-spmf}$;

let $a0 = S0 \oplus S2 \oplus m00$;

let $a1 = S0 \oplus S3 \oplus m01$;

let $a2 = S1 \oplus S4 \oplus m10$;

let $a3 = S1 \oplus S5 \oplus m11$;

$(-,Si) \leftarrow \text{protocol-OT12} (S0, S1) c0$;

$(-,Sj) \leftarrow \text{protocol-OT12} (S2, S3) c1$;

$(-,Sk) \leftarrow \text{protocol-OT12} (S4, S5) c1$;

let $s2 = Si \oplus (\text{if } c0 \text{ then } Sk \text{ else } Sj) \oplus (\text{if } c0 \text{ then } (\text{if } c1 \text{ then } a3 \text{ else } a2) \text{ else } (\text{if } c1 \text{ then } a1 \text{ else } a0))$;

return-spmf $(((), s2)$ }

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

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

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

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

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

definition *R1-14-interm2* :: *input1* \Rightarrow *input2* \Rightarrow '*v-OT121 view1* *spmf*
where *R1-14-interm2 msgs choice* = *do* {
let (*m00, m01, m10, m11*) = *msgs*;
let (*c0, c1*) = *choice*;
S0 :: *bool* \leftarrow *coin-spmf*;
S1 :: *bool* \leftarrow *coin-spmf*;
S2 :: *bool* \leftarrow *coin-spmf*;
S3 :: *bool* \leftarrow *coin-spmf*;
S4 :: *bool* \leftarrow *coin-spmf*;
S5 :: *bool* \leftarrow *coin-spmf*;


```

a :: 'v-OT121 ← S1-OT12 (S0, S1) ();
b :: 'v-OT121 ← S1-OT12 (S2, S3) ();
c :: 'v-OT121 ← R1-OT12 (S4, S5) c1;
return-spmf (msgs, (S0, S1, S2, S3, S4, S5), a, b, c)

```

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

definition *S1-14* :: *input1 ⇒ unit ⇒ 'v-OT121 view1 spmf*

```

where S1-14 msgs = do {
  let (m00, m01, m10, m11) = msgs;
  S0 :: bool ← coin-spmf;
  S1 :: bool ← coin-spmf;
  S2 :: bool ← coin-spmf;
  S3 :: bool ← coin-spmf;
  S4 :: bool ← coin-spmf;
  S5 :: bool ← coin-spmf;
  a :: 'v-OT121 ← S1-OT12 (S0, S1) ();
  b :: 'v-OT121 ← S1-OT12 (S2, S3) ();
  c :: 'v-OT121 ← S1-OT12 (S4, S5) ();
  return-spmf (msgs, (S0, S1, S2, S3, S4, S5), a, b, c)
}

```

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

lemma *reduction-step1*:

```

shows ∃ A1. |spmf (bind-spmf (R1-14 M (c0, c1)) D) True – spmf (bind-spmf
(R1-14-interm1 M (c0, c1)) D) True| =
  |spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf
(R1-OT12 (m0,m1) c0) (λ view. (A1 view (m0,m1)))) True –
  spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1).
bind-spmf (S1-OT12 (m0,m1) ()) (λ view. (A1 view (m0,m1)))) True|

```

including *monad-normalisation*

proof–

```

define A1' where A1' == λ (view :: 'v-OT121) (m0,m1). do {
  S2 :: bool ← coin-spmf;
  S3 :: bool ← coin-spmf;
  S4 :: bool ← coin-spmf;
  S5 :: bool ← coin-spmf;
  b :: 'v-OT121 ← R1-OT12 (S2, S3) c1;
  c :: 'v-OT121 ← R1-OT12 (S4, S5) c1;
  let R = (M, (m0,m1, S2, S3, S4, S5), view, b, c);
  D R}

```

```

have |spmf (bind-spmf (R1-14 M (c0, c1)) D) True – spmf (bind-spmf (R1-14-interm1
M (c0, c1)) D) True| =
  |spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf
(R1-OT12 (m0,m1) c0) (λ view. (A1' view (m0,m1)))) True –
  spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf
(S1-OT12 (m0,m1) ()) (λ view. (A1' view (m0,m1)))) True|

```

```

apply(simp add: pair-spmf-alt-def R1-14-def R1-14-interm1-def A1'-def Let-def
split-def)
  apply(subst bind-commute-spmf[of S1-OT12 -])
  apply(subst bind-commute-spmf[of S1-OT12 -])
  apply(subst bind-commute-spmf[of S1-OT12 -])
  apply(subst bind-commute-spmf[of S1-OT12 -])
  apply(subst bind-commute-spmf[of S1-OT12 -])
  by auto
then show ?thesis by auto
qed

lemma reduction-step1':
  shows |spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1). bind-spmf
(R1-OT12 (m0,m1) c0) (λ view. (A1 view (m0,m1)))))) True –
      |spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (λ(m0, m1).
bind-spmf (S1-OT12 (m0,m1) ()) (λ view. (A1 view (m0,m1)))))) True|
      ≤ adv-OT12
  (is ?lhs ≤ adv-OT12)
proof –
  have int1: integrable (measure-spmf (pair-spmf coin-spmf coin-spmf)) (λx. spmf
(case x of (m0, m1) ⇒ R1-OT12 (m0, m1) c0) ≫ (λview. A1 view (m0, m1)))
True)
  and int2: integrable (measure-spmf (pair-spmf coin-spmf coin-spmf)) (λx. spmf
(case x of (m0, m1) ⇒ S1-OT12 (m0, m1) ()) ≫ (λview. A1 view (m0, m1)))
True)
  by(rule measure-spmf.integrable-const-bound[where B=1]; simp add: pmf-le-1)+
  have ?lhs =
    |LINT x|measure-spmf (pair-spmf coin-spmf coin-spmf). spmf (case x of
(m0, m1) ⇒ R1-OT12 (m0, m1) c0) ≫ (λview. A1 view (m0, m1))) True
    – spmf (case x of (m0, m1) ⇒ S1-OT12 (m0, m1) ()) ≫ (λview. A1
view (m0, m1))) True|
  apply(subst (1 2) spmf-bind) using int1 int2 by simp
  also have ... ≤ LINT x|measure-spmf (pair-spmf coin-spmf coin-spmf).
|spmf (R1-OT12 x c0) ≫ (λview. A1 view x)) True – spmf (S1-OT12
x ()) ≫ (λview. A1 view x)) True|
  by(rule integral-abs-bound[THEN order-trans]; simp add: split-beta)
  ultimately have ?lhs ≤ LINT x|measure-spmf (pair-spmf coin-spmf coin-spmf).
|spmf (R1-OT12 x c0) ≫ (λview. A1 view x)) True – spmf
(S1-OT12 x ()) ≫ (λview. A1 view x)) True|
  by simp
  also have LINT x|measure-spmf (pair-spmf coin-spmf coin-spmf).
|spmf (R1-OT12 x c0) ≫ (λview::'v-OT121. A1 view x)) True
  – spmf (S1-OT12 x ()) ≫ (λview::'v-OT121. A1 view x)) True|
≤ adv-OT12
  apply(rule integral-mono[THEN order-trans])
  apply(rule measure-spmf.integrable-const-bound[where B=2])
  apply clarsimp
  apply(rule abs-triangle-ineq4[THEN order-trans])

```

subgoal for $m0\ m1$
using $pmf-le-1$ [of $R1-OT12\ (m0, m1)\ c0 \gg (\lambda view. A1\ view\ (m0, m1))$
Some True]
 $pmf-le-1$ [of $S1-OT12\ (m0, m1)\ () \gg (\lambda view. A1\ view\ (m0, m1))$ *Some*
True]
by *simp*
apply *simp*
apply(*rule measure-spmf.integrable-const*)
apply *clarify*
apply(*rule OT-12-P1-assms-bound'[rule-format]*)
by *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *reduction-step2*:

shows $\exists A1. |spmf\ (bind-spmf\ (R1-14-interm1\ M\ (c0, c1))\ D)\ True - spmf$
 $(bind-spmf\ (R1-14-interm2\ M\ (c0, c1))\ D)\ True| =$
 $|spmf\ (bind-spmf\ (pair-spmf\ coin-spmf\ coin-spmf)\ (\lambda(m0, m1). bind-spmf$
 $(R1-OT12\ (m0,m1)\ c1)\ (\lambda\ view.\ (A1\ view\ (m0,m1)))))\ True -$
 $spmf\ (bind-spmf\ (pair-spmf\ coin-spmf\ coin-spmf)\ (\lambda(m0, m1). bind-spmf$
 $(S1-OT12\ (m0,m1)\ ())\ (\lambda\ view.\ (A1\ view\ (m0,m1)))))\ True|$

proof –

define $A1'$ **where** $A1' == \lambda\ (view :: 'v-OT121)\ (m0,m1). do\ \{$

$S2 :: bool \leftarrow coin-spmf;$
 $S3 :: bool \leftarrow coin-spmf;$
 $S4 :: bool \leftarrow coin-spmf;$
 $S5 :: bool \leftarrow coin-spmf;$
 $a :: 'v-OT121 \leftarrow S1-OT12\ (S2,S3)\ ();$
 $c :: 'v-OT121 \leftarrow R1-OT12\ (S4, S5)\ c1;$
 $let\ R = (M, (S2,S3, m0, m1, S4, S5), a, view, c);$
 $D\ R\}$

have $|spmf\ (bind-spmf\ (R1-14-interm1\ M\ (c0, c1))\ D)\ True - spmf\ (bind-spmf$
 $(R1-14-interm2\ M\ (c0, c1))\ D)\ True| =$

$|spmf\ (bind-spmf\ (pair-spmf\ coin-spmf\ coin-spmf)\ (\lambda(m0, m1). bind-spmf$
 $(R1-OT12\ (m0,m1)\ c1)\ (\lambda\ view.\ (A1'\ view\ (m0,m1)))))\ True -$
 $spmf\ (bind-spmf\ (pair-spmf\ coin-spmf\ coin-spmf)\ (\lambda(m0, m1). bind-spmf$
 $(S1-OT12\ (m0,m1)\ ())\ (\lambda\ view.\ (A1'\ view\ (m0,m1)))))\ True|$

proof –

have $(bind-spmf\ (R1-14-interm1\ M\ (c0, c1))\ D) = (bind-spmf\ (pair-spmf$
 $coin-spmf\ coin-spmf)\ (\lambda(m0, m1). bind-spmf\ (R1-OT12\ (m0,m1)\ c1)\ (\lambda\ view.$
 $(A1'\ view\ (m0,m1))))$

unfolding *R1-14-interm1-def R1-14-interm2-def A1'-def Let-def split-def*

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

apply(*rewrite in bind-spmf - \sqcap in bind-spmf - \sqcap in - = \sqcap bind-commute-spmf*)

apply(*rewrite in bind-spmf - \sqcap in bind-spmf - \sqcap in bind-spmf - \sqcap in - = \sqcap*
bind-commute-spmf)

including *monad-normalisation* **by**(*simp*)

also have $(bind-spmf\ (R1-14-interm2\ M\ (c0, c1))\ D) = (bind-spmf\ (pair-spmf$
 $coin-spmf\ coin-spmf)\ (\lambda(m0, m1). bind-spmf\ (S1-OT12\ (m0,m1)\ ())\ (\lambda\ view.$

```

(A1' view (m0,m1))))))
  unfolding R1-14-interm1-def R1-14-interm2-def A1'-def Let-def split-def
  apply(simp add: pair-spmf-alt-def)
  apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
  apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$ 
bind-commute-spmf)
  apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
  apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
  apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
  apply(rewrite in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
  apply(rewrite in - =  $\sqcap$  bind-commute-spmf)
  apply(rewrite in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
  by(simp)
  ultimately show ?thesis by simp
qed
then show ?thesis by auto
qed

```

lemma *reduction-step3*:

```

shows  $\exists A1. |spmf (bind-spmf (R1-14-interm2 M (c0, c1)) D) True - spmf
(bind-spmf (S1-14 M out) D) True| =
|spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (\lambda(m0, m1). bind-spmf
(R1-OT12 (m0,m1) c1) (\lambda view. (A1' view (m0,m1)))))) True -
spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (\lambda(m0, m1). bind-spmf
(S1-OT12 (m0,m1) ()) (\lambda view. (A1' view (m0,m1)))))) True|$ 
```

proof –

```

define A1' where A1' ==  $\lambda (view :: 'v-OT121) (m0,m1). do \{$ 

```

```

  S2 :: bool  $\leftarrow$  coin-spmf;
  S3 :: bool  $\leftarrow$  coin-spmf;
  S4 :: bool  $\leftarrow$  coin-spmf;
  S5 :: bool  $\leftarrow$  coin-spmf;
  a :: 'v-OT121  $\leftarrow$  S1-OT12 (S2,S3) ();
  b :: 'v-OT121  $\leftarrow$  S1-OT12 (S4, S5) ();
  let R = (M, (S2,S3, S4, S5,m0, m1), a, b, view);
  D R}

```

```

have |spmf (bind-spmf (R1-14-interm2 M (c0, c1)) D) True - spmf (bind-spmf
(S1-14 M out) D) True| =

```

```

|spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (\lambda(m0, m1). bind-spmf
(R1-OT12 (m0,m1) c1) (\lambda view. (A1' view (m0,m1)))))) True -
spmf (bind-spmf (pair-spmf coin-spmf coin-spmf) (\lambda(m0, m1). bind-spmf
(S1-OT12 (m0,m1) ()) (\lambda view. (A1' view (m0,m1)))))) True|

```

proof –

```

have (bind-spmf (R1-14-interm2 M (c0, c1)) D) = (bind-spmf (pair-spmf
coin-spmf coin-spmf) (\lambda(m0, m1). bind-spmf (R1-OT12 (m0,m1) c1) (\lambda view.
(A1' view (m0,m1))))))

```

```

unfolding R1-14-interm2-def A1'-def Let-def split-def
apply(simp add: pair-spmf-alt-def)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$ 
bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
including monad-normalisation by(simp)
also have (bind-spmf (S1-14 M out) D) = (bind-spmf (pair-spmf coin-spmf
coin-spmf) ( $\lambda$ (m0, m1). bind-spmf (S1-OT12 (m0,m1) ()) ( $\lambda$  view. (A1' view
(m0,m1))))))
unfolding S1-14-def Let-def A1'-def split-def
apply(simp add: pair-spmf-alt-def)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$ 
bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$  bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf
-  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in - =  $\sqcap$ 
bind-commute-spmf)
apply(rewrite in  $\sqcap$  = - bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in  $\sqcap$  = - bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in  $\sqcap$  = - bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in  $\sqcap$  = -
bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in  $\sqcap$  = - bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in  $\sqcap$  = - bind-commute-spmf)
apply(rewrite in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in bind-spmf -  $\sqcap$  in  $\sqcap$  = -
bind-commute-spmf)
including monad-normalisation by(simp)
ultimately show ?thesis by simp
qed
then show ?thesis by auto
qed

```

lemma reduction-P1-interm:

```

shows |spmf (bind-spmf (R1-14 M (c0,c1)) (D)) True - spmf (bind-spmf (S1-14
M out) (D)) True|  $\leq$  3 * adv-OT12
(is ?lhs  $\leq$  ?rhs)

```

proof -

```

have lhs: ?lhs  $\leq$  |spmf (bind-spmf (R1-14 M (c0, c1)) D) True - spmf (bind-spmf
(R1-14-interm1 M (c0, c1)) D) True| +

```

$| \text{spmf} (\text{bind-spmf} (R1-14\text{-interm1 } M (c0, c1)) D) \text{ True} - \text{spmf}$
 $(\text{bind-spmf} (R1-14\text{-interm2 } M (c0, c1)) D) \text{ True}| +$
 $| \text{spmf} (\text{bind-spmf} (R1-14\text{-interm2 } M (c0, c1)) D) \text{ True} - \text{spmf}$
 $(\text{bind-spmf} (S1-14 M \text{ out}) D) \text{ True}|$
by simp
obtain A1 where A1: $| \text{spmf} (\text{bind-spmf} (R1-14 M (c0, c1)) D) \text{ True} - \text{spmf}$
 $(\text{bind-spmf} (R1-14\text{-interm1 } M (c0, c1)) D) \text{ True}| =$
 $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c0) (\lambda \text{ view. } (A1 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0,$
 $m1). \text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A1 \text{ view } (m0, m1)))) \text{ True}|$
using reduction-step1 by blast
obtain A2 where A2: $| \text{spmf} (\text{bind-spmf} (R1-14\text{-interm1 } M (c0, c1)) D) \text{ True}$
 $- \text{spmf} (\text{bind-spmf} (R1-14\text{-interm2 } M (c0, c1)) D) \text{ True}| =$
 $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c1) (\lambda \text{ view. } (A2 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0,$
 $m1). \text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A2 \text{ view } (m0, m1)))) \text{ True}|$
using reduction-step2 by blast
obtain A3 where A3: $| \text{spmf} (\text{bind-spmf} (R1-14\text{-interm2 } M (c0, c1)) D) \text{ True}$
 $- \text{spmf} (\text{bind-spmf} (S1-14 M \text{ out}) D) \text{ True}| =$
 $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c1) (\lambda \text{ view. } (A3 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0,$
 $m1). \text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A3 \text{ view } (m0, m1)))) \text{ True}|$
using reduction-step3 by blast
have lhs-bound: $?lhs \leq | \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0,$
 $m1). \text{bind-spmf} (R1\text{-OT12 } (m0, m1) c0) (\lambda \text{ view. } (A1 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A1 \text{ view } (m0, m1)))) \text{ True}| +$
 $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c1) (\lambda \text{ view. } (A2 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A2 \text{ view } (m0, m1)))) \text{ True}| +$
 $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c1) (\lambda \text{ view. } (A3 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A3 \text{ view } (m0, m1)))) \text{ True}|$
using A1 A2 A3 lhs by simp
have bound1: $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c0) (\lambda \text{ view. } (A1 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A1 \text{ view } (m0, m1)))) \text{ True}|$
 $\leq \text{adv-OT12}$
and bound2: $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (R1\text{-OT12 } (m0, m1) c1) (\lambda \text{ view. } (A2 \text{ view } (m0, m1)))) \text{ True} -$
 $\text{spmf} (\text{bind-spmf} (\text{pair-spmf coin-spmf coin-spmf}) (\lambda(m0, m1).$
 $\text{bind-spmf} (S1\text{-OT12 } (m0, m1) ()) (\lambda \text{ view. } (A2 \text{ view } (m0, m1)))) \text{ True}|$
 $\leq \text{adv-OT12}$

and *bound3*: $| \text{spmf} (\text{bind-spmf} (\text{pair-spmf} \text{ coin-spmf} \text{ coin-spmf}) (\lambda(m0, m1). \text{bind-spmf} (R1-OT12 (m0,m1) c1) (\lambda \text{view}. (A3 \text{view} (m0,m1)))))) \text{ True} - \text{spmf} (\text{bind-spmf} (\text{pair-spmf} \text{ coin-spmf} \text{ coin-spmf}) (\lambda(m0, m1). \text{bind-spmf} (S1-OT12 (m0,m1) ())) (\lambda \text{view}. (A3 \text{view} (m0,m1)))))) \text{ True} | \leq \text{adv-OT12}$
using *reduction-step1'* **by** *auto*
thus *?thesis*
using *reduction-step1' lhs-bound* **by** *argo*
qed

lemma *reduction-P1*: $| \text{spmf} (\text{bind-spmf} (R1-14 M (c0,c1)) (D)) \text{ True} - \text{spmf} (\text{funct-OT-14} M (c0,c1) \gg (\lambda (\text{out1}, \text{out2}). S1-14 M \text{out1} \gg (\lambda \text{view}. D \text{view}))) \text{ True} | \leq 3 * \text{adv-OT12}$
by(*simp add: funct-OT-14-def split-def Let-def reduction-P1-interm*)

Party 2 security.

lemma *coin-coin*: $\text{map-spmf} (\lambda S0. S0 \oplus S3 \oplus m1) \text{ coin-spmf} = \text{coin-spmf}$
(is ?lhs = ?rhs)

proof –

have *lhs*: $?lhs = \text{map-spmf} (\lambda S0. S0 \oplus (S3 \oplus m1)) \text{ coin-spmf}$ **by** *blast*

also have *op-eq*: $\dots = \text{map-spmf} ((\oplus) (S3 \oplus m1)) \text{ coin-spmf}$

by (*metis xor-bool-def*)

also have $\dots = ?rhs$

using *xor-uni-samp* **by** *fastforce*

ultimately show *?thesis*

using *op-eq* **by** *auto*

qed

lemma *coin-coin'*: $\text{map-spmf} (\lambda S3. S0 \oplus S3 \oplus m1) \text{ coin-spmf} = \text{coin-spmf}$

proof –

have $\text{map-spmf} (\lambda S3. S0 \oplus S3 \oplus m1) \text{ coin-spmf} = \text{map-spmf} (\lambda S3. S3 \oplus S0 \oplus m1) \text{ coin-spmf}$

by (*metis xor-left-commute*)

thus *?thesis* **using** *coin-coin* **by** *simp*

qed

definition *R2-14*:: $\text{input1} \Rightarrow \text{input2} \Rightarrow 'v\text{-OT122} \text{ view2} \text{ spmf}$

where *R2-14* $M C = \text{do} \{$

let $(m0,m1,m2,m3) = M;$

let $(c0,c1) = C;$

$S0 :: \text{bool} \leftarrow \text{coin-spmf};$

$S1 :: \text{bool} \leftarrow \text{coin-spmf};$

$S2 :: \text{bool} \leftarrow \text{coin-spmf};$

$S3 :: \text{bool} \leftarrow \text{coin-spmf};$

$S4 :: \text{bool} \leftarrow \text{coin-spmf};$

$S5 :: \text{bool} \leftarrow \text{coin-spmf};$

let $a0 = S0 \oplus S2 \oplus m0;$

let $a1 = S0 \oplus S3 \oplus m1;$

let $a2 = S1 \oplus S4 \oplus m2;$

```

let a3 = S1 ⊕ S5 ⊕ m3;
a :: 'v-OT122 ← R2-OT12 (S0,S1) c0;
b :: 'v-OT122 ← R2-OT12 (S2,S3) c1;
c :: 'v-OT122 ← R2-OT12 (S4,S5) c1;
return-spmf (C, (a0,a1,a2,a3), a,b,c)

```

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

definition *S2-14* :: *input2* ⇒ *bool* ⇒ 'v-OT122 *view2* *spmf*
where *S2-14 C out* = *do* {
 let ((*c0*::*bool*),(*c1*::*bool*)) = *C*;
 S0 :: *bool* ← *coin-spmf*;
 S1 :: *bool* ← *coin-spmf*;
 S2 :: *bool* ← *coin-spmf*;
 S3 :: *bool* ← *coin-spmf*;
 S4 :: *bool* ← *coin-spmf*;
 S5 :: *bool* ← *coin-spmf*;
 a0 :: *bool* ← *coin-spmf*;
 a1 :: *bool* ← *coin-spmf*;
 a2 :: *bool* ← *coin-spmf*;
 a3 :: *bool* ← *coin-spmf*;
 let *a0'* = (*if* ((¬ *c0*) ∧ (¬ *c1*)) *then* (*S0* ⊕ *S2* ⊕ *out*) *else* *a0*);
 let *a1'* = (*if* ((¬ *c0*) ∧ *c1*) *then* (*S0* ⊕ *S3* ⊕ *out*) *else* *a1*);
 let *a2'* = (*if* (*c0* ∧ (¬ *c1*)) *then* (*S1* ⊕ *S4* ⊕ *out*) *else* *a2*);
 let *a3'* = (*if* (*c0* ∧ *c1*) *then* (*S1* ⊕ *S5* ⊕ *out*) *else* *a3*);
 a :: 'v-OT122 ← *S2-OT12* (*c0*::*bool*) (*if* *c0* *then* *S1* *else* *S0*);
 b :: 'v-OT122 ← *S2-OT12* (*c1*::*bool*) (*if* *c1* *then* *S3* *else* *S2*);
 c :: 'v-OT122 ← *S2-OT12* (*c1*::*bool*) (*if* *c1* *then* *S5* *else* *S4*);
 return-spmf ((*c0*,*c1*), (*a0'*,*a1'*,*a2'*,*a3'*), *a*,*b*,*c*)}

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

lemma *P2-OT-14-FT*: *R2-14 (m0,m1,m2,m3) (False,True) = funct-OT-14 (m0,m1,m2,m3)*
(False,True) ≫= (λ (out1, out2). S2-14 (False,True) out2)

including *monad-normalisation*

proof–

```

have R2-14 (m0,m1,m2,m3) (False,True) = do {
  S0 :: bool ← coin-spmf;
  S1 :: bool ← coin-spmf;
  S3 :: bool ← coin-spmf;
  S5 :: bool ← coin-spmf;
  a0 :: bool ← map-spmf (λ S2. S0 ⊕ S2 ⊕ m0) coin-spmf;
  let a1 = S0 ⊕ S3 ⊕ m1;
  a2 ← map-spmf (λ S4. S1 ⊕ S4 ⊕ m2) coin-spmf;
  let a3 = S1 ⊕ S5 ⊕ m3;
  a :: 'v-OT122 ← S2-OT12 False S0;
  b :: 'v-OT122 ← S2-OT12 True S3;

```



```

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

```

lemma *P2-OT-14-TT*: *R2-14* (*m0,m1,m2,m3*) (*True,True*) = *funct-OT-14* (*m0,m1,m2,m3*)

```

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

```

```

a0 :: bool ← coin-spmf;
a1 :: bool ← coin-spmf;
a2 ← coin-spmf;
let a3 = S1 ⊕ S5 ⊕ m3;
a :: 'v-OT122 ← S2-OT12 True S1;
b :: 'v-OT122 ← S2-OT12 True S3;
c :: 'v-OT122 ← S2-OT12 True S5;
return-spmf ((True, True), (a0, a1, a2, a3), a, b, c)
using coin-coin by simp
ultimately show ?thesis
by(simp add: funct-OT-14-def S2-14-def bind-spmf-const)
qed

```

lemma P2-OT-14-FF: R2-14 (m0, m1, m2, m3) (False, False) = funct-OT-14 (m0, m1, m2, m3) (False, False) \gg (λ (out1, out2). S2-14 (False, False) out2)

including monad-normalisation

proof –

```

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

```

```

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

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

```

```

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

```

qed

lemma *P2-sec-OT-14-split*: $R2-14 (m0, m1, m2, m3) (c0, c1) = \text{funct-OT-14} (m0, m1, m2, m3) (c0, c1) \gg (\lambda (out1, out2). S2-14 (c0, c1) out2)$
by(*cases c0*; *cases c1*; *auto simp add: P2-OT-14-FF P2-OT-14-TF P2-OT-14-FT P2-OT-14-TT*)

lemma *P2-sec-OT-14*: $R2-14 M C = \text{funct-OT-14} M C \gg (\lambda (out1, out2). S2-14 C out2)$
by(*metis P2-sec-OT-14-split surj-pair*)

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

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

lemma *OT-14-correct*: $OT-14.\text{correctness} M C$
unfolding *OT-14.correctness-def*
using *correctness-OT-14 by auto*

lemma *OT-14-P2-sec*: $OT-14.\text{perfect-sec-P2} m1 m2$
unfolding *OT-14.perfect-sec-P2-def*
using *P2-sec-OT-14 by blast*

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

end

locale *OT-14-asymp* = *sim-det-def* +
fixes $S1-OT12 :: \text{nat} \Rightarrow (\text{bool} \times \text{bool}) \Rightarrow \text{unit} \Rightarrow 'v\text{-OT121} \text{ spmf}$
and $R1-OT12 :: \text{nat} \Rightarrow (\text{bool} \times \text{bool}) \Rightarrow \text{bool} \Rightarrow 'v\text{-OT121} \text{ spmf}$
and $\text{adv-OT12} :: \text{nat} \Rightarrow \text{real}$
and $S2-OT12 :: \text{nat} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow 'v\text{-OT122} \text{ spmf}$
and $R2-OT12 :: \text{nat} \Rightarrow (\text{bool} \times \text{bool}) \Rightarrow \text{bool} \Rightarrow 'v\text{-OT122} \text{ spmf}$
and $\text{protocol-OT12} :: (\text{bool} \times \text{bool}) \Rightarrow \text{bool} \Rightarrow (\text{unit} \times \text{bool}) \text{ spmf}$
assumes $\text{ot14-base: } \bigwedge (n::\text{nat}). \text{ot14-base} (S1-OT12 n) (R1-12-OT n) (\text{adv-OT12} n) (S2-OT12 n) (R2-12OT n) (\text{protocol-OT12})$
begin

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

lemma *OT-14-P1-sec*: *OT-14.adv-P1* (*R1-12-0T* *n*) *n* *m1* *m2* *D* $\leq 3 * (\text{adv-OT12 } n)$

unfolding *OT-14.adv-P1-def* **using** *reduction-P1 surj-pair* **by** *metis*

theorem *OT-14-P1-asym-sec*: *negligible* ($\lambda n. \text{OT-14.adv-P1 } (R1-12-0T\ n)\ n\ m1\ m2\ D$) **if** *negligible* ($\lambda n. \text{adv-OT12 } n$)

proof –

have *adv-neg*: *negligible* ($\lambda n. 3 * \text{adv-OT12 } n$) **using** *that negligible-cmultI* **by** *simp*

have $|\text{OT-14.adv-P1 } (R1-12-0T\ n)\ n\ m1\ m2\ D| \leq |3 * (\text{adv-OT12 } n)|$ **for** *n*

proof –

have $|\text{OT-14.adv-P1 } (R1-12-0T\ n)\ n\ m1\ m2\ D| \leq 3 * \text{adv-OT12 } n$

using *OT-14.adv-P1-def* *OT-14-P1-sec* **by** *auto*

then show *?thesis*

by (*meson abs-ge-self order-trans*)

qed

thus *?thesis* **using** *OT-14-P1-sec negligible-le adv-neg*

by (*metis (no-types, lifting) negligible-absI*)

qed

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

end

end

2.7 1-out-of-4 OT to GMW

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

theory *GMW* **imports**

OT14

begin

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

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

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

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

type-synonym *msgs-14-OT* = (*bool* \times *bool* \times *bool* \times *bool*)

type-synonym *choice-14-OT* = (*bool* \times *bool*)

type-synonym *share-wire* = (*share-1* \times *share-2*)

locale *gmw-base* =
fixes *S1-14-OT* :: *msgs-14-OT* \Rightarrow *unit* \Rightarrow '*v-14-OT1* *spmf* — simulated view for party 1 of OT14
and *R1-14-OT* :: *msgs-14-OT* \Rightarrow *choice-14-OT* \Rightarrow '*v-14-OT1* *spmf* — real view for party 1 of OT14
and *S2-14-OT* :: *choice-14-OT* \Rightarrow *bool* \Rightarrow '*v-14-OT2* *spmf*
and *R2-14-OT* :: *msgs-14-OT* \Rightarrow *choice-14-OT* \Rightarrow '*v-14-OT2* *spmf*
and *protocol-14-OT* :: *msgs-14-OT* \Rightarrow *choice-14-OT* \Rightarrow (*unit* \times *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* \leq *adv-14-OT* — bound the advantage of party 1 in the 1-out-of-4 OT
and *P2-OT-12-inf-theoretic*: *sim-det-def.perfect-sec-P2* *R2-14-OT* *S2-14-OT* *funct-14-OT* *M C* — information theoretic security for party 2 in the 1-out-of-4 OT
and *correct-14*: *funct-OT-14* *msgs C* = *protocol-14-OT* *msgs C* — correctness of the 1-out-of-4 OT
and *lossless-R1-14-OT*: *lossless-spmf* (*R1-14-OT* (*m1,m2,m3,m4*) (*c0,c1*))
and *lossless-R2-14-OT*: *lossless-spmf* (*R2-14-OT* (*m1,m2,m3,m4*) (*c0,c1*))
and *lossless-S1-14-OT*: *lossless-spmf* (*S1-14-OT* (*m1,m2,m3,m4*) ())
and *lossless-S2-14-OT*: *lossless-spmf* (*S2-14-OT* (*c0,c1*) *b*)
and *lossless-protocol-14-OT*: *lossless-spmf* (*protocol-14-OT* *S C*)
and *lossless-funct-14-OT*: *lossless-spmf* (*funct-14-OT* *M C*)
begin

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

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

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

lemma *ass-adv-14-OT*: $|$ *spmf* (*bind-spmf* (*S1-14-OT* *msgs* ()) (λ *view*. (*D* *view*)))
True –
 $|$ *spmf* (*bind-spmf* (*R1-14-OT* *msgs* (*c0,c1*)) (λ *view*. (*D* *view*)))
True $|$ \leq *adv-14-OT*
(**is** ?*lhs* \leq *adv-14-OT*)
proof –
have *funct-OT-14* (*m0,m1,m2,m3*) (*c0*, *c1*) \ggg (λ (*o1*, *o2*). *S1-14-OT* (*m0,m1,m2,m3*)
() \ggg *D*) = *S1-14-OT* (*m0,m1,m2,m3*) () \ggg *D*


```

for  $m0\ m1\ m2\ m3$  by(simp add: funct-14)
hence funct: funct-OT-14 msgs (c0, c1)  $\ggg$  ( $\lambda(o1, o2). S1-14-OT\ msgs\ ()$ )  $\ggg$  D
= S1-14-OT msgs ()  $\ggg$  D
by (metis prod-cases4)
have  $?lhs = |spmf\ (bind\ -\ smpf\ (R1-14-OT\ msgs\ (c0,c1))\ (\lambda\ view.\ (D\ view)))$ 
True
- spmf\ (bind\ -\ smpf\ (S1-14-OT\ msgs\ ())\ (\lambda\ view.\ (D\ view)))\ True|
by linarith
hence  $\dots = |(spmf\ (R1-14-OT\ msgs\ (c0,c1))\ \ggg\ (\lambda\ view.\ D\ view))\ True$ 
- spmf\ (funct-OT-14\ msgs\ (c0,c1)\ \ggg\ (\lambda\ (o1, o2).\ S1-14-OT\ msgs\ o1
 $\ggg\ (\lambda\ view.\ D\ view)))\ True|$ 
by(simp add: funct)
thus ?thesis using P1-OT-14-adv-bound sim-det-def.adv-P1-def
by (simp add: OT-14-sim.adv-P1-def abs-minus-commute)
qed

```

The sharing scheme

```

definition share :: bool  $\Rightarrow$  share-wire smpf
where share x = do {
   $a_1 \leftarrow coin\ -\ smpf;$ 
   $let\ b_1 = x \oplus a_1;$ 
  return-smpf (a1, b1)}

```

```

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

```

```

definition reconstruct :: (share-1  $\times$  share-2)  $\Rightarrow$  bool smpf
where reconstruct shares = do {
   $let\ (a,b) = shares;$ 
  return-smpf (a  $\oplus$  b)}

```

```

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

```

```

lemma reconstruct-share : (bind-smpf (share x) reconstruct) = (return-smpf x)
proof -

```

```

  have  $y = (y = x) = x$  for  $y$  by auto
  thus ?thesis
  by(auto simp add: share-def reconstruct-def bind-smpf-const eq-commute)
qed

```

```

lemma (reconstruct (s1,s2)  $\ggg$  ( $\lambda\ rec.\ share\ rec\ \ggg$  ( $\lambda\ shares.\ reconstruct\ shares$ )))
= return-smpf (s1  $\oplus$  s2)
apply(simp add: reconstruct-share reconstruct-def share-def)
apply(cases s1; cases s2) by(auto simp add: bind-smpf-const)

```

```

definition xor-evaluate :: bool  $\Rightarrow$  bool  $\Rightarrow$  bool smpf
where xor-evaluate A B = return-smpf (A  $\oplus$  B)

```

definition *xor-funct* :: *share-wire* \Rightarrow *share-wire* \Rightarrow (*bool* \times *bool*) *spmf*
where *xor-funct* *A B* = *do* {
 let (*a1*, *b1*) = *A*;
 let (*a2*, *b2*) = *B*;
 return-spmf (*a1* \oplus *a2*, *b1* \oplus *b2*)}

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

definition *xor-protocol* :: *share-wire* \Rightarrow *share-wire* \Rightarrow (*bool* \times *bool*) *spmf*
where *xor-protocol* *A B* = *do* {
 let (*a1*, *b1*) = *A*;
 let (*a2*, *b2*) = *B*;
 return-spmf (*a1* \oplus *a2*, *b1* \oplus *b2*)}

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

lemma *share-xor-reconstruct*:
shows *share* *x* \gg (λ *w1*. *share* *y* \gg (λ *w2*. *xor-protocol* *w1* *w2*
 \gg (λ (*a*, *b*). *reconstruct* (*a*, *b*)))) = *xor-evaluate* *x y*

proof–

have (*ya* = (\neg *yb*)) = ((*x* = (\neg *ya*)) = (*y* = (\neg *yb*))) = (*x* = (\neg *y*)) **for** *ya yb*
by *auto*

then show *?thesis*

by(*simp add: share-def xor-protocol-def reconstruct-def xor-evaluate-def bind-spmf-const*)

qed

definition *R1-xor* :: (*bool* \times *bool*) \Rightarrow (*bool* \times *bool*) \Rightarrow (*bool* \times *bool*) *spmf*
where *R1-xor* *A B* = *return-spmf* *A*

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

definition *S1-xor* :: (*bool* \times *bool*) \Rightarrow *bool* \Rightarrow (*bool* \times *bool*) *spmf*
where *S1-xor* *A out* = *return-spmf* *A*

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

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

definition *R2-xor* :: (*bool* \times *bool*) \Rightarrow (*bool* \times *bool*) \Rightarrow (*bool* \times *bool*) *spmf*
where *R2-xor* *A B* = *return-spmf* *B*

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

definition $S2\text{-xor} :: (\text{bool} \times \text{bool}) \Rightarrow \text{bool} \Rightarrow (\text{bool} \times \text{bool}) \text{ spmf}$
where $S2\text{-xor } B \text{ out} = \text{return-spmf } B$

lemma $\text{lossless-}S2\text{-xor}$: $\text{lossless-spmf } (S2\text{-xor } A \text{ out})$
by($\text{simp add: } S2\text{-xor-def}$)

lemma $P2\text{-xor-inf-th}$: $R2\text{-xor } A \ B = \text{xor-funct } A \ B \ggg (\lambda (\text{out1}, \text{out2}). S2\text{-xor } B \text{ out2})$
by($\text{simp add: } R2\text{-xor-def xor-funct-def } S2\text{-xor-def split-def}$)

sublocale xor-sim-det : $\text{sim-det-def } R1\text{-xor } S1\text{-xor } R2\text{-xor } S2\text{-xor } \text{xor-funct } \text{xor-protocol}$

unfolding sim-det-def-def
by($\text{simp add: } \text{lossless-}R1\text{-xor } \text{lossless-}S1\text{-xor } \text{lossless-}R2\text{-xor } \text{lossless-}S2\text{-xor } \text{lossless-xor-funct}$)

lemma $\text{xor-sim-det.perfect-sec-P1 } m1 \ m2$
by($\text{simp add: } \text{xor-sim-det.perfect-sec-P1-def } P1\text{-xor-inf-th}$)

lemma $\text{xor-sim-det.perfect-sec-P2 } m1 \ m2$
by($\text{simp add: } \text{xor-sim-det.perfect-sec-P2-def } P2\text{-xor-inf-th}$)

definition $\text{and-funct} :: (\text{share-1} \times \text{share-2}) \Rightarrow (\text{share-1} \times \text{share-2}) \Rightarrow \text{share-wire spmf}$
where $\text{and-funct } A \ B = \text{do } \{$
 $\text{let } (a1, a2) = A;$
 $\text{let } (b1, b2) = B;$
 $\sigma \leftarrow \text{coin-spmf};$
 $\text{return-spmf } (\sigma, \sigma \oplus ((a1 \oplus b1) \wedge (a2 \oplus b2)))\}$

lemma $\text{lossless-and-funct}$: $\text{lossless-spmf } (\text{and-funct } A \ B)$
by($\text{simp add: } \text{and-funct-def split-def}$)

definition $\text{and-evaluate} :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool spmf}$
where $\text{and-evaluate } A \ B = \text{return-spmf } (A \wedge B)$

definition $\text{and-protocol} :: \text{share-wire} \Rightarrow \text{share-wire} \Rightarrow \text{share-wire spmf}$
where $\text{and-protocol } A \ B = \text{do } \{$
 $\text{let } (a1, b1) = A;$
 $\text{let } (a2, b2) = B;$
 $\sigma \leftarrow \text{coin-spmf};$
 $\text{let } s0 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (b1 \oplus \text{False}));$
 $\text{let } s1 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (b1 \oplus \text{True}));$
 $\text{let } s2 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (b1 \oplus \text{False}));$
 $\text{let } s3 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (b1 \oplus \text{True}));$
 $(-, s) \leftarrow \text{protocol-14-OT } (s0, s1, s2, s3) (a2, b2);$
 $\text{return-spmf } (\sigma, s)\}$

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

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

lemma *share-and-reconstruct:*

shows *share x \gg (λ (a1,a2). share y \gg (λ (b1,b2).*

and-protocol (a1,b1) (a2,b2) \gg (λ (a, b). reconstruct (a, b)))) =

and-evaluate x y

proof –

have (*yc = (\neg (if x = (\neg ya) then if snd (snd (ya, x = (\neg ya)), snd (yb, y = (\neg yb))) then yc*

= (fst (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb))) \vee snd (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb))))

else yc = (fst (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb))) \vee \neg snd (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb))))

else if snd (snd (ya, x = (\neg ya)), snd (yb, y = (\neg yb))) then yc = (fst (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb)))

\rightarrow snd (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb)))

else yc = (fst (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb)))

\rightarrow \neg snd (fst (ya, x = (\neg ya)), fst (yb, y = (\neg yb)))

yb)))))) = (x \wedge y)

for *yc yb ya by auto*

then show *?thesis*

by(*auto simp add: share-def reconstruct-def and-protocol-def and-evaluate-def split-def correct-14[symmetric] funct-14 bind-spmf-const Let-def*)

qed

definition *and-R1 :: (share-1 \times share-1) \Rightarrow (share-2 \times share-2) \Rightarrow (((share-1 \times share-1) \times bool \times 'v-14-OT1) \times (share-1 \times share-2)) spmf*

where *and-R1 A B = do {*

let (a1, a2) = A;

let (b1,b2) = B;

$\sigma \leftarrow$ coin-spmf;

let s0 = $\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}));$

let s1 = $\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}));$

let s2 = $\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}));$

let s3 = $\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True}));$

V \leftarrow R1-14-OT (s0,s1,s2,s3) (b1,b2);

(-, s) \leftarrow protocol-14-OT (s0,s1,s2,s3) (b1,b2);

return-spmf (((a1,a2), σ , V), (σ , s))}

lemma *lossless-and-R1: lossless-spmf (and-R1 A B)*

apply(*simp add: and-R1-def split-def lossless-R1-14-OT lossless-protocol-14-OT Let-def*)

by (*metis prod.collapse lossless-R1-14-OT*)

definition $S1\text{-and} :: (\text{share-1} \times \text{share-1}) \Rightarrow \text{bool} \Rightarrow (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v\text{-14-OT1})) \text{ spmf}$

where $S1\text{-and} A \sigma = \text{do} \{$
 $\text{let } (a1, a2) = A;$
 $\text{let } s0 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}));$
 $\text{let } s1 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}));$
 $\text{let } s2 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}));$
 $\text{let } s3 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True}));$
 $V \leftarrow S1\text{-14-OT } (s0, s1, s2, s3) ();$
 $\text{return-spmf } ((a1, a2), \sigma, V)\}$

definition $\text{out1} :: (\text{share-1} \times \text{share-1}) \Rightarrow (\text{share-2} \times \text{share-2}) \Rightarrow \text{bool} \Rightarrow (\text{share-1} \times \text{share-2}) \text{ spmf}$

where $\text{out1} A B \sigma = \text{do} \{$
 $\text{let } (a1, a2) = A;$
 $\text{let } (b1, b2) = B;$
 $\text{return-spmf } (\sigma, \sigma \oplus ((a1 \oplus b1) \wedge (a2 \oplus b2)))\}$

definition $S1\text{-and}' :: (\text{share-1} \times \text{share-1}) \Rightarrow (\text{share-2} \times \text{share-2}) \Rightarrow \text{bool} \Rightarrow (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v\text{-14-OT1}) \times (\text{share-1} \times \text{share-2})) \text{ spmf}$

where $S1\text{-and}' A B \sigma = \text{do} \{$
 $\text{let } (a1, a2) = A;$
 $\text{let } (b1, b2) = B;$
 $\text{let } s0 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}));$
 $\text{let } s1 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}));$
 $\text{let } s2 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}));$
 $\text{let } s3 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True}));$
 $V \leftarrow S1\text{-14-OT } (s0, s1, s2, s3) ();$
 $\text{return-spmf } (((a1, a2), \sigma, V), (\sigma, \sigma \oplus ((a1 \oplus b1) \wedge (a2 \oplus b2))))\}$

lemma $\text{sec-ex-P1-and}:$

shows $\exists (A :: 'v\text{-14-OT1} \Rightarrow \text{bool} \Rightarrow \text{bool} \text{ spmf}).$

$| \text{spmf } ((\text{and-funct } (a1, a2) (b1, b2)) \gg (\lambda (s1, s2). (S1\text{-and}' (a1, a2) (b1, b2) s1))$

$\gg (D :: (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v\text{-14-OT1}) \times (\text{share-1} \times \text{share-2})) \Rightarrow \text{bool} \text{ spmf})) | \text{True} - \text{spmf } ((\text{and-R1 } (a1, a2) (b1, b2)) \gg D) \text{True} | =$

$| \text{spmf } (\text{coin-spmf} \gg (\lambda \sigma. S1\text{-14-OT } ((\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}))), (\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True})))) ()$

$\gg (\lambda \text{view}. A \text{view } \sigma)) \text{True}$

$- \text{spmf } (\text{coin-spmf} \gg (\lambda \sigma. R1\text{-14-OT } ((\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}))), (\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True})))) (b1, b2)$

$\gg (\lambda \text{view}. A \text{view } \sigma)) \text{True} |$

including $\text{monad-normalisation}$

proof –

define A' **where** $A' == \lambda \text{view } \sigma. (D (((a1, a2), \sigma, \text{view}), (\sigma, \sigma \oplus ((a1 \oplus b1) \wedge (a2 \oplus b2))))))$

have $| \text{spmf } ((\text{and-funct } (a1, a2) (b1, b2)) \gg (\lambda (s1, s2). (S1\text{-and}' (a1, a2) (b1, b2) s1))$

$(b1, b2) s1$
 $\gg (D :: (((bool \times bool) \times bool \times 'v-14-OT1) \times (share-1 \times share-2)) \Rightarrow$
 $bool\ spmf))\ True -$
 $spmf ((and-R1 (a1, a2) (b1, b2)) \gg (D :: (((bool \times bool) \times bool \times$
 $'v-14-OT1) \times (bool \times bool)) \Rightarrow bool\ spmf))\ True| =$
 $|spmf (coin-spmf \gg (\lambda \sigma :: bool. S1-14-OT ((\sigma \oplus ((a1 \oplus False) \wedge$
 $(a2 \oplus False))), (\sigma \oplus ((a1 \oplus False) \wedge (a2 \oplus True))), (\sigma \oplus ((a1 \oplus True) \wedge (a2 \oplus$
 $False))), (\sigma \oplus ((a1 \oplus True) \wedge (a2 \oplus True))))\ ()$
 $\gg (\lambda view. A' view \sigma))\ True - spmf (coin-spmf \gg (\lambda \sigma. R1-14-OT$
 $((\sigma \oplus ((a1 \oplus False) \wedge (a2 \oplus False))), (\sigma \oplus ((a1 \oplus False) \wedge (a2 \oplus True))), (\sigma \oplus$
 $((a1 \oplus True) \wedge (a2 \oplus False))), (\sigma \oplus ((a1 \oplus True) \wedge (a2 \oplus True))))\ (b1, b2)$
 $\gg (\lambda view. A' view \sigma))\ True|$
by(*auto simp add: S1-and'-def A'-def and-funct-def and-R1-def Let-def split-def*
correct-14[symmetric] funct-14; cases a1; cases a2; cases b1; cases b2; auto)
then show *?thesis by auto*
qed

lemma *bound-14-OT*:

$|spmf (coin-spmf \gg (\lambda \sigma. S1-14-OT ((\sigma \oplus ((a1 \oplus False) \wedge (a2 \oplus False))), (\sigma$
 $\oplus ((a1 \oplus False) \wedge (a2 \oplus True))), (\sigma \oplus ((a1 \oplus True) \wedge (a2 \oplus False))), (\sigma \oplus ((a1$
 $\oplus True) \wedge (a2 \oplus True))))\ ()$
 $\gg (\lambda view. (A :: 'v-14-OT1 \Rightarrow bool \Rightarrow bool\ spmf)\ view\ \sigma))\ True - spmf$
 $(coin-spmf \gg (\lambda \sigma. R1-14-OT ((\sigma \oplus ((a1 \oplus False) \wedge (a2 \oplus False))), (\sigma \oplus ((a1$
 $\oplus False) \wedge (a2 \oplus True))), (\sigma \oplus ((a1 \oplus True) \wedge (a2 \oplus False))), (\sigma \oplus ((a1 \oplus$
 $True) \wedge (a2 \oplus True))))\ (b1, b2)$
 $\gg (\lambda view. A\ view\ \sigma))\ True| \leq adv-14-OT$
(is *?lhs* $\leq adv-14-OT$)

proof –

have *int1*: *integrable (measure-spmf coin-spmf) ($\lambda x. spmf (S1-14-OT (x \oplus (a1$*
 $\oplus False \wedge a2 \oplus False), x \oplus (a1 \oplus False \wedge a2 \oplus True), x \oplus (a1 \oplus True \wedge a2 \oplus$
 $False), x \oplus (a1 \oplus True \wedge a2 \oplus True))\ () \gg (\lambda view. A\ view\ x))\ True$
and *int2*: *integrable (measure-spmf coin-spmf) ($\lambda x. spmf (R1-14-OT (x \oplus (a1$*
 $\oplus False \wedge a2 \oplus False), x \oplus (a1 \oplus False \wedge a2 \oplus True), x \oplus (a1 \oplus True \wedge a2 \oplus$
 $False), x \oplus (a1 \oplus True \wedge a2 \oplus True))\ (b1, b2) \gg (\lambda view. A\ view\ x))\ True$
by(*rule measure-spmf.integrable-const-bound[where B=1]; simp add: pmf-le-1*) +
have *?lhs* = $|LINT\ x|measure-spmf\ coin-spmf.$
 $spmf (S1-14-OT (x \oplus (a1 \oplus False \wedge a2 \oplus False), x \oplus (a1 \oplus False \wedge a2$
 $\oplus True), x \oplus (a1 \oplus True \wedge a2 \oplus False), x \oplus (a1 \oplus True \wedge a2 \oplus True))\ () \gg$
 $(\lambda view. A\ view\ x))\ True -$
 $spmf (R1-14-OT (x \oplus (a1 \oplus False \wedge a2 \oplus False), x \oplus (a1 \oplus False \wedge a2 \oplus$
 $True), x \oplus (a1 \oplus True \wedge a2 \oplus False), x \oplus (a1 \oplus True \wedge a2 \oplus True))\ (b1, b2)$
 $\gg (\lambda view. A\ view\ x))\ True|$
apply(*subst (1 2) spmf-bind*) **using** *int1 int2 by simp*
also have $\dots \leq LINT\ x|measure-spmf\ coin-spmf. |spmf (S1-14-OT (x = (a1 \longrightarrow$
 $\neg a2), x = (a1 \longrightarrow a2), x = (a1 \vee \neg a2), x = (a1 \vee a2))\ () \gg (\lambda view. A\ view$
 $x))\ True$
 $- spmf (R1-14-OT (x = (a1 \longrightarrow \neg a2), x = (a1 \longrightarrow a2), x = (a1$
 $\vee \neg a2), x = (a1 \vee a2))\ (b1, b2) \gg (\lambda view. A\ view\ x))\ True|$
by(*rule integral-abs-bound[THEN order-trans]; simp add: split-beta*)

ultimately have $?lhs \leq LINT x | \text{measure-spmf coin-spmf} . | \text{spmf } (S1-14-OT (x = (a1 \longrightarrow \neg a2), x = (a1 \longrightarrow a2), x = (a1 \vee \neg a2), x = (a1 \vee a2))) () \ggg (\lambda \text{view. } A \text{ view } x) \text{ True}$
 $- \text{spmf } (R1-14-OT (x = (a1 \longrightarrow \neg a2), x = (a1 \longrightarrow a2), x = (a1 \vee \neg a2), x = (a1 \vee a2))) (b1, b2) \ggg (\lambda \text{view. } A \text{ view } x) \text{ True}$
by simp
also have $LINT x | \text{measure-spmf coin-spmf} . | \text{spmf } (S1-14-OT (x = (a1 \longrightarrow \neg a2), x = (a1 \longrightarrow a2), x = (a1 \vee \neg a2), x = (a1 \vee a2))) () \ggg (\lambda \text{view. } A \text{ view } x) \text{ True}$
 $- \text{spmf } (R1-14-OT (x = (a1 \longrightarrow \neg a2), x = (a1 \longrightarrow a2), x = (a1 \vee \neg a2), x = (a1 \vee a2))) (b1, b2) \ggg (\lambda \text{view. } A \text{ view } x) \text{ True} \leq \text{adv-14-OT}$
apply(rule integral-mono[THEN order-trans])
apply(rule measure-spmf.integrable-const-bound[where B=2])
apply clarsimp
apply(rule abs-triangle-ineq4[THEN order-trans])
apply(cases a1) **apply**(cases a2)
subgoal for M
using pmf-le-1[of R1-14-OT ($\neg M, M, M, M$) (b1,b2) $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
 pmf-le-1 [of S1-14-OT ($\neg M, M, M, M$) () $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
by simp
subgoal for M
using pmf-le-1[of R1-14-OT ($M, \neg M, M, M$) (b1,b2) $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
 pmf-le-1 [of S1-14-OT ($M, \neg M, M, M$) () $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
by simp
apply(cases a2) **apply**(auto)
subgoal for M
using pmf-le-1[of R1-14-OT ($M, M, \neg M, M$) (b1,b2) $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
 pmf-le-1 [of S1-14-OT ($M, M, \neg M, M$) () $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
by(simp)
subgoal for M
using pmf-le-1[of R1-14-OT ($M, M, M, \neg M$) (b1,b2) $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
 pmf-le-1 [of S1-14-OT ($M, M, M, \neg M$) () $\ggg (\lambda \text{view. } A \text{ view } M) \text{ Some True}$]
by(simp)
using ass-adv-14-OT **by fast**
ultimately show ?thesis **by simp**
qed

lemma security-and-P1:

shows $| \text{spmf } ((\text{and-funct } (a1, a2) (b1,b2))) \ggg (\lambda (s1, s2). (S1\text{-and}' (a1,a2) (b1,b2) s1))$
 $\ggg (D :: (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v\text{-14-OT1}) \times (\text{share-1} \times \text{share-2}))$

$\Rightarrow \text{bool spmf})) \text{ True} -$
 $\text{spmf } ((\text{and-R1 } (a1, a2) (b1, b2)) \gg D) \text{ True} | \leq \text{adv-14-OT}$

proof –

obtain $A :: 'v-14-OT1 \Rightarrow \text{bool} \Rightarrow \text{bool spmf}$ **where** $A:$
 $| \text{spmf } ((\text{and-funct } (a1, a2) (b1, b2)) \gg (\lambda (s1, s2). (\text{S1-and}' (a1, a2) (b1, b2) s1) \gg D)) \text{ True} - \text{spmf } ((\text{and-R1 } (a1, a2) (b1, b2)) \gg D) \text{ True} | =$
 $| \text{spmf } (\text{coin-spmf } \gg (\lambda \sigma. \text{S1-14-OT } ((\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}))),$
 $(\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}))), (\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus$
 $((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True})))) ()$
 $\gg (\lambda \text{view. } A \text{ view } \sigma)) \text{ True} - \text{spmf } (\text{coin-spmf}$
 $\gg (\lambda \sigma. \text{R1-14-OT } ((\sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus ((a1 \oplus$
 $\text{False}) \wedge (a2 \oplus \text{True}))), (\sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}))), (\sigma \oplus ((a1 \oplus \text{True})$
 $\wedge (a2 \oplus \text{True})))) (b1, b2)$
 $\gg (\lambda \text{view. } A \text{ view } \sigma)) \text{ True} |$
using *sec-ex-P1-and* **by** *blast*
then show *?thesis using bound-14-OT*[of $a1 a2 A b1 b2$] **by** *metis*
qed

lemma *security-and-P1'*:

shows $| \text{spmf } ((\text{and-R1 } (a1, a2) (b1, b2)) \gg D) \text{ True} -$
 $\text{spmf } ((\text{and-funct } (a1, a2) (b1, b2)) \gg (\lambda (s1, s2). (\text{S1-and}' (a1, a2)$
 $(b1, b2) s1)$
 $\gg (D :: (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v-14-OT1) \times (\text{share-1} \times \text{share-2})))$
 $\Rightarrow \text{bool spmf})) \text{ True} | \leq \text{adv-14-OT}$

proof –

have $| \text{spmf } ((\text{and-R1 } (a1, a2) (b1, b2)) \gg D) \text{ True} -$
 $\text{spmf } ((\text{and-funct } (a1, a2) (b1, b2)) \gg (\lambda (s1, s2). (\text{S1-and}' (a1, a2)$
 $(b1, b2) s1)$
 $\gg (D :: (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v-14-OT1) \times (\text{share-1} \times \text{share-2})))$
 $\Rightarrow \text{bool spmf})) \text{ True} | =$
 $| \text{spmf } ((\text{and-funct } (a1, a2) (b1, b2)) \gg (\lambda (s1, s2). (\text{S1-and}' (a1, a2)$
 $(b1, b2) s1)$
 $\gg (D :: (((\text{bool} \times \text{bool}) \times \text{bool} \times 'v-14-OT1) \times (\text{share-1} \times \text{share-2})))$
 $\Rightarrow \text{bool spmf})) \text{ True} -$
 $\text{spmf } ((\text{and-R1 } (a1, a2) (b1, b2)) \gg D) \text{ True} |$ **using** *abs-minus-commute*
by *blast*
thus *?thesis using security-and-P1* **by** *simp*
qed

definition *and-R2* $:: (\text{share-1} \times \text{share-2}) \Rightarrow (\text{share-2} \times \text{share-1}) \Rightarrow (((\text{bool} \times$
 $\text{bool}) \times 'v-14-OT2) \times (\text{share-1} \times \text{share-2})) \text{ spmf}$

where *and-R2* $A B = \text{do } \{$
 $\text{let } (a1, a2) = A;$
 $\text{let } (b1, b2) = B;$
 $\sigma \leftarrow \text{coin-spmf};$
 $\text{let } s0 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{False}));$
 $\text{let } s1 = \sigma \oplus ((a1 \oplus \text{False}) \wedge (a2 \oplus \text{True}));$
 $\text{let } s2 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{False}));$
 $\text{let } s3 = \sigma \oplus ((a1 \oplus \text{True}) \wedge (a2 \oplus \text{True}));$

$(-, s) \leftarrow \text{protocol-14-OT } (s0, s1, s2, s3) B;$
 $V \leftarrow \text{R2-14-OT } (s0, s1, s2, s3) B;$
 $\text{return-spmf } ((B, V), (\sigma, s))$

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

definition *S2-and* :: $(\text{share-1} \times \text{share-2}) \Rightarrow \text{bool} \Rightarrow (((\text{bool} \times \text{bool}) \times 'v\text{-14-OT2}))$
 spmf
where *S2-and* B s2 = do {
 $\text{let } (a2, b2) = B;$
 $V :: 'v\text{-14-OT2} \leftarrow \text{S2-14-OT } (a2, b2) s2;$
 $\text{return-spmf } ((B, V))$ }

definition *out2* :: $(\text{share-1} \times \text{share-2}) \Rightarrow (\text{share-1} \times \text{share-2}) \Rightarrow \text{bool} \Rightarrow (\text{share-1} \times \text{share-2})$ spmf
where *out2* B A s2 = do {
 $\text{let } (a1, b1) = A;$
 $\text{let } (a2, b2) = B;$
 $\text{let } s1 = s2 \oplus ((a1 \oplus a2) \wedge (b1 \oplus b2));$
 $\text{return-spmf } (s1, s2)$ }

definition *S2-and'* :: $(\text{share-1} \times \text{share-2}) \Rightarrow (\text{share-1} \times \text{share-2}) \Rightarrow \text{bool} \Rightarrow (((\text{bool} \times \text{bool}) \times 'v\text{-14-OT2}) \times (\text{share-1} \times \text{share-2}))$ spmf
where *S2-and'* B A s2 = do {
 $\text{let } (a1, a2) = A;$
 $\text{let } (b1, b2) = B;$
 $V :: 'v\text{-14-OT2} \leftarrow \text{S2-14-OT } B s2;$
 $\text{let } s1 = s2 \oplus ((a1 \oplus b1) \wedge (a2 \oplus b2));$
 $\text{return-spmf } ((B, V), s1, s2)$ }

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

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

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

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

proof –

have *and-R2 (a1, b1) (a2, b2) = and-funct (a1, b1) (a2, b2) \ggg ($\lambda(s1, s2).$ and-secret-sharing.Ideal2 (a2, b2) (a1, b1) s2)*

```

for a1 a2 b1 b2
apply(auto simp add: split-def inf-th-14-OT-P4 S2-and'-def and-R2-def and-funct-def
Let-def correct-14[symmetric] and-secret-sharing.Ideal2-def S2-and-def out2-def)
apply(simp only: funct-14)
apply auto
by(cases b1; cases b2; cases a1; cases a2; auto)
thus ?thesis
by(simp add: and-secret-sharing.perfect-sec-P2-def;metis prod.collapse)
qed

```

```

lemma and-P1-security: and-secret-sharing.adv-P1 m1 m2 D ≤ adv-14-OT
proof –
have |spmf (and-R1 (a1, a2) (b1, b2) ≫≧ D) True –
spmf (and-funct (a1, a2) (b1, b2) ≫≧ (λ(s1, s2).
and-secret-sharing.Ideal1 (a1, a2) (b1, b2) s1 ≫≧ D)) True|
≤ adv-14-OT for a1 a2 b1 b2
using security-and-P1' ideal-S1-and prod.collapse by simp
thus ?thesis
by(simp add: and-secret-sharing.adv-P1-def;metis prod.collapse)
qed

```

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

```

lemma share-reconstruct-xor: share x ≫≧ (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 –
have (((ya = (x = ya) = (yb = (y = (¬ yb)))))) = (x = (¬ y)) for ya yb by
auto
thus ?thesis
by(simp add: xor-protocol-def share-def reconstruct-def xor-evaluate-def bind-spmf-const)
qed

```

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

```

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

```

```

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

```

end

```

locale gmw-asym =
fixes S1-14-OT :: nat ⇒ msgs-14-OT ⇒ unit ⇒ 'v-14-OT1 spmf
and R1-14-OT :: nat ⇒ msgs-14-OT ⇒ choice-14-OT ⇒ 'v-14-OT1 spmf

```

```

and S2-14-OT :: nat ⇒ choice-14-OT ⇒ bool ⇒ 'v-14-OT2 spmf
and R2-14-OT :: nat ⇒ msgs-14-OT ⇒ choice-14-OT ⇒ 'v-14-OT2 spmf
and protocol-14-OT :: nat ⇒ msgs-14-OT ⇒ choice-14-OT ⇒ (unit × bool)
spmf
and adv-14-OT :: nat ⇒ real
assumes gmw-base: ∧ (n::nat). gmw-base (S1-14-OT n) (R1-14-OT n) (S2-14-OT
n) (R2-14-OT n) (protocol-14-OT n) (adv-14-OT n)
begin

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

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

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

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

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

```

end

end

2.8 Secure multiplication protocol

theory *Secure-Multiplication* **imports**

CryptHOL.Cyclic-Group-SPMF

Uniform-Sampling

Semi-Honest-Def

begin

locale *secure-mult* =

fixes *q* :: nat

assumes *q-gt-0*: *q* > 0

and *prime* *q*

begin

type-synonym *real-view* = $\text{nat} \Rightarrow \text{nat} \Rightarrow ((\text{nat} \times \text{nat} \times \text{nat} \times \text{nat}) \times \text{nat} \times \text{nat}) \text{ spmf}$
type-synonym *sim* = $\text{nat} \Rightarrow \text{nat} \Rightarrow ((\text{nat} \times \text{nat} \times \text{nat} \times \text{nat}) \times \text{nat} \times \text{nat}) \text{ spmf}$

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

definition *funct* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ spmf}$
where *funct* *x y* = *do* {
s \leftarrow *sample-uniform* *q*;
return-spmf (*s*, (*x***y* + (*q* - *s*)) *mod* *q*)}

definition *TI* :: $((\text{nat} \times \text{nat}) \times (\text{nat} \times \text{nat})) \text{ spmf}$
where *TI* = *do* {
a \leftarrow *sample-uniform* *q*;
b \leftarrow *sample-uniform* *q*;
r \leftarrow *sample-uniform* *q*;
return-spmf ((*a*, *r*), (*b*, ((*a***b* + (*q* - *r*)) *mod* *q*)))}

definition *out* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ spmf}$
where *out* *x y* = *do* {
(*c1*,*d1*),(*c2*,*d2*) \leftarrow *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*) \leftarrow *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* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat} \times \text{nat} \times \text{nat}) \text{ spmf}$
where *S1* *x s1* = *do* {
a :: *nat* \leftarrow *sample-uniform* *q*;
e1 \leftarrow *sample-uniform* *q*;
let *d1* = (*x***e1* + (*q* - *s1*)) *mod* *q*;
return-spmf (*x*, *a*, *d1*, *e1*)}

definition *Out1* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ spmf}$
where *Out1* *x y s1* = *do* {
let *s2* = (*x***y* + (*q* - *s1*)) *mod* *q*;
return-spmf (*s1*,*s2*)}

definition $R2 :: \text{real-view}$

where $R2\ x\ y = \text{do}$ {
 $((c1, d1), (c2, d2)) \leftarrow TI$;
 $\text{let } e2 = (x + c1) \bmod q$;
 $\text{let } e1 = (y + (q - c2)) \bmod q$;
 $\text{let } s1 = (x * e1 + (q - d1)) \bmod q$;
 $\text{let } s2 = (e2 * c2 + (q - d2)) \bmod q$;
 $\text{return-spmf } ((y, c2, d2, e2), s1, s2)$ }

definition $S2 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat} \times \text{nat} \times \text{nat}) \text{ spmf}$

where $S2\ y\ s2 = \text{do}$ {
 $b \leftarrow \text{sample-uniform } q$;
 $e2 \leftarrow \text{sample-uniform } q$;
 $\text{let } d2 = (e2 * b + (q - s2)) \bmod q$;
 $\text{return-spmf } (y, b, d2, e2)$ }

definition $Out2 :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times \text{nat}) \text{ spmf}$

where $Out2\ y\ x\ s2 = \text{do}$ {
 $\text{let } s1 = (x * y + (q - s2)) \bmod q$;
 $\text{return-spmf } (s1, s2)$ }

definition $Ideal2 :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow ((\text{nat} \times \text{nat} \times \text{nat} \times \text{nat}) \times (\text{nat} \times \text{nat})) \text{ spmf}$

where $Ideal2\ y\ x\ out2 = \text{do}$ {
 $\text{view2} :: (\text{nat} \times \text{nat} \times \text{nat} \times \text{nat}) \leftarrow S2\ y\ out2$;
 $out2 \leftarrow Out2\ y\ x\ out2$;
 $\text{return-spmf } (\text{view2}, out2)$ }

sublocale $\text{sim-non-det-def}: \text{sim-non-det-def } R1\ S1\ Out1\ R2\ S2\ Out2\ \text{funct} .$

lemma minus-mod :

assumes $a > b$

shows $[a - b \bmod q = a - b] \pmod{q}$

by ($\text{metis } \text{cong-diff-nat } \text{cong-def } \text{le-trans } \text{less-or-eq-imp-le } \text{assms } \text{mod-less-eq-dividend } \text{mod-mod-trivial}$)

lemma $q\text{-cong}: [a = q + a] \pmod{q}$

by ($\text{simp } \text{add: } \text{cong-def}$)

lemma $q\text{-cong-reverse}: [q + a = a] \pmod{q}$

by ($\text{simp } \text{add: } \text{cong-def}$)

lemma $qq\text{-cong}: [a = q * q + a] \pmod{q}$

by ($\text{simp } \text{add: } \text{cong-def}$)

lemma $\text{minus-q-mult-cancel}$:

assumes $[a = e + b - q * c - d] \pmod{q}$

and $e + b - d > 0$

and $e + b - q * c - d > 0$

shows $[a = e + b - d] \pmod{q}$
proof –
have $a \pmod{q} = (e + b - q * c - d) \pmod{q}$
using *assms(1) cong-def* **by** *blast*
then have $a \pmod{q} = (e + b - d) \pmod{q}$
by (*metis (no-types) add-cancel-left-left assms(3) diff-commute diff-is-0-eq'*
linordered-semidom-class.add-diff-inverse mod-add-left-eq mod-mult-self1-is-0 nat-less-le)
then show *?thesis*
using *cong-def* **by** *blast*
qed

lemma *s1-s2*:

assumes $x < q \ a < q \ b < q$ **and** $r : r < q \ y < q$
shows $((x + a) \pmod{q} * b + q - (a * b + q - r) \pmod{q}) \pmod{q} =$
 $(x * y + q - (x * ((y + q - b) \pmod{q}) + q - r) \pmod{q}) \pmod{q}$
proof –
have $s : (x * y + (q - (x * ((y + (q - b)) \pmod{q}) + (q - r)) \pmod{q})) \pmod{q}$
 $= ((x + a) \pmod{q} * b + (q - (a * b + (q - r)) \pmod{q})) \pmod{q}$
proof –
have $lhs : (x * y + (q - (x * ((y + (q - b)) \pmod{q}) + (q - r)) \pmod{q})) \pmod{q}$
 $= (x * b + r) \pmod{q}$
proof –
let $?h = (x * y + (q - (x * ((y + (q - b)) \pmod{q}) + (q - r)) \pmod{q})) \pmod{q}$
have $[?h = x * y + q - (x * ((y + (q - b)) \pmod{q}) + (q - r)) \pmod{q}] \pmod{q}$
 $q)$
by (*simp add: assms(1) cong-def q-gt-0*)
then have $[?h = x * y + q - (x * (y + (q - b)) + (q - r)) \pmod{q}] \pmod{q}$
by (*metis mod-add-left-eq mod-mult-right-eq*)
then have $no-qq : [?h = x * y + q - (x * y + x * (q - b) + (q - r)) \pmod{q}] \pmod{q}$
 $q]$ (*mod q*)
by (*metis distrib-left*)
then have $[?h = q * q + x * y + q - (x * y + x * (q - b) + (q - r)) \pmod{q}] \pmod{q}$
 $q]$ (*mod q*)
proof –
have $[x * y + q - (x * y + x * (q - b) + (q - r)) \pmod{q} = q * q + x * y$
 $+ q - (x * y + x * (q - b) + (q - r)) \pmod{q}] \pmod{q}$
by (*smt qq-cong add.assoc cong-diff-nat cong-def le-add2 le-trans mod-le-divisor*
q-gt-0)
then show *?thesis* **using** *cong-trans no-qq* **by** *blast*
qed
then have $mod : [?h = q + q * q + x * y + q - (x * y + x * (q - b) + (q -$
 $r)) \pmod{q}] \pmod{q}$
by (*smt Nat.add-diff-assoc cong-def add.assoc add.commute le-add2 le-trans*
mod-add-self2 mod-le-divisor q-gt-0)
then have $[?h = q + q * q + x * y + q - (x * y + x * (q - b) + (q - r))] \pmod{q}$
 (\pmod{q})
proof –
have $1 : q \geq q - b$ **using** *assms* **by** *simp*
then have $q * q \geq x * (q - b) \ q \geq q - r$ **using** 1 *assms*

```

    apply (auto simp add: mult-strict-mono)
    by (simp add: mult-le-mono)
  then have  $q + q*q + x * y + q > x * y + x * (q - b) + (q - r)$ 
    using assms(5) by linarith
  then have  $[q + q*q + x * y + q - (x * y + x * (q - b) + (q - r)) \bmod$ 
 $q = q + q*q + x * y + q - (x * y + x * (q - b) + (q - r))] \bmod q$ 
    using minus-mod by blast
  then show ?thesis using mod using cong-trans by blast
qed
  then have  $[?h = q + q*q + x * y + q - (x * y + (x * q - x*b) + (q -$ 
 $r))] \bmod q$ 
    by (simp add: right-diff-distrib')
  then have  $[?h = q + q*q + x * y + q - x * y - (x * q - x*b) - (q - r)]$ 
 $(\bmod q)$ 
    by simp
  then have mod':  $[?h = q + q*q + q - (x * q - x*b) - (q - r)] \bmod q$ 
    by (simp add: add commute)
  then have neg:  $[?h = q + q*q + q - x * q + x*b - (q - r)] \bmod q$ 
  proof-
    have  $[q + q*q + q - (x * q - x*b) - (q - r) = q + q*q + q - x * q +$ 
 $x*b - (q - r)] \bmod q$ 
      proof (cases  $x = 0$ )
        case True
          then show ?thesis by simp
        next
          case False
            have  $x * q - x*b > 0$  using False assms by simp
            also have  $q + q*q + q - x * q > 0$ 
              by (metis assms(1) add commute diff-mult-distrib2 less-Suc-eq mult commute
 $mult-Suc-right$  nat-0-less-mult-iff q-gt-0 zero-less-diff)
            ultimately show ?thesis by simp
      qed
    then show ?thesis using mod' cong-trans by blast
  qed
  then have  $[?h = q + q*q + q + x*b - (q - r)] \bmod q$ 
  proof-
    have  $[q + q*q + q - x * q + x*b - (q - r) = q + q*q + q + x*b - (q$ 
 $- r)] \bmod q$ 
      proof (cases  $x = 0$ )
        case True
          then show ?thesis by simp
        next
          case False
            have  $q*q > x*q$ 
              using False assms
              by (simp add: mult-strict-mono)
            then have 1:  $q + q*q + q - x * q + x*b - (q - r) > 0$ 
              by linarith
            then have 2:  $q + q*q + q + x*b - (q - r) > 0$  by simp

```

```

then show ?thesis
  by (smt 1 2 Nat.add-diff-assoc2 ⟨x * q < q * q⟩ add-cancel-left-left
add-diff-inverse-nat
  le-add1 le-add2 le-trans less-imp-add-positive less-numeral-extra(3)
minus-mod
  minus-q-mult-cancel mod-if mult.commute q-gt-0)
qed
then show ?thesis using cong-trans neg by blast
qed
then have [?h = q + q*q + q + x*b - q + r] (mod q)
  by (metis r(1) Nat.add-diff-assoc2 Nat.diff-diff-right le-add2 less-imp-le-nat
semiring-normalization-rules(23))
then have [?h = q + q*q + q + x*b + r] (mod q)
  apply(simp add: cong-def)
  by (metis (no-types, lifting) add.assoc add.commute add-diff-cancel-right'
diff-is-0-eq' mod-if mod-le-divisor q-gt-0)
then have [?h = x*b + r] (mod q)
  apply(simp add: cong-def)
  by (metis mod-add-cong mod-add-self1 mod-mult-self1)
then show ?thesis by (simp add: cong-def assms)
qed
also have rhs: ((x + a) mod q * b + (q - (a * b + (q - r)) mod q)) mod q
= (x*b + r) mod q
proof-
  let ?h = ((x + a) mod q * b + (q - (a * b + (q - r)) mod q)) mod q
  have [?h = (x + a) mod q * b + q - (a * b + (q - r)) mod q] (mod q)
    by (simp add: q-gt-0 assms(1) cong-def)
  then have [?h = (x + a) * b + q - (a * b + (q - r)) mod q] (mod q)
    by (smt Nat.add-diff-assoc cong-def mod-add-cong mod-le-divisor mod-mult-left-eq
q-gt-0 assms)
  then have [?h = x*b + a*b + q - (a * b + (q - r)) mod q] (mod q)
    by(metis distrib-right)
  then have mod: [?h = q + x*b + a*b + q - (a * b + (q - r)) mod q] (mod
q)
    by (smt Nat.add-diff-assoc cong-def add.assoc add.commute le-add2 le-trans
mod-add-self2 mod-le-divisor q-gt-0)
  then have [?h = q + x*b + a*b + q - (a * b + (q - r))] (mod q) using
q-cong assms(1)
proof-
  have ge: q + x*b + a*b + q > (a * b + (q - r)) using assms by simp
  then have [ q + x*b + a*b + q - (a * b + (q - r)) mod q = q + x*b +
a*b + q - (a * b + (q - r))] (mod q)
    using Divides.mod-less-eq-dividend cong-diff-nat cong-def le-trans less-not-refl2
less-or-eq-imp-le q-gt-0 minus-mod by presburger
  then show ?thesis using mod cong-trans by blast
qed
then have [?h = q + x*b + q - (q - r)] (mod q)
  by (simp add: add.commute)
then have [?h = q + x*b + q - q + r] (mod q)

```


by (*metis Nat.add-diff-assoc2 Nat.diff-diff-right r(1) le-add2 less-imp-le-nat*)
 then have [$?h = q + x*b + r \pmod q$] by *simp*
 then have [$?h = q + (x*b + r) \pmod q$]
 using *add.assoc* by *metis*
 then have [$?h = x*b + r \pmod q$]
 using *cong-def q-cong-reverse* by *metis*
 then show *?thesis* by (*simp add: cong-def assms(1)*)
 qed
 ultimately show *?thesis* by *simp*
 qed
 have *lhs*: $((x + a) \bmod q * b + q - (a * b + q - r) \bmod q) \bmod q = ((x + a) \bmod q * b + (q - (a * b + (q - r)) \bmod q)) \bmod q$
 using *assms* by *simp*
 have *rhs*: $(x * y + q - (x * ((y + q - b) \bmod q) + q - r) \bmod q) \bmod q = (x * y + (q - (x * ((y + (q - b)) \bmod q) + (q - r)) \bmod q)) \bmod q$
 using *assms* by *simp*
 have $((x + a) \bmod q * b + (q - (a * b + (q - r)) \bmod q)) \bmod q = (x * y + (q - (x * ((y + (q - b)) \bmod q) + (q - r)) \bmod q)) \bmod q$
 using *assms s[symmetric]* by *blast*
 then show *?thesis* using *lhs rhs*
 by *metis*
 qed

lemma *s1-s2-P2*:

assumes $x < q \quad xa < q \quad xb < q \quad xc < q \quad y < q$
 shows $((y, xa, (xb * xa + q - xc) \bmod q, (x + xb) \bmod q), (x * ((y + q - xa) \bmod q) + q - xc) \bmod q, ((x + xb) \bmod q * xa + q - (xb * xa + q - xc) \bmod q) \bmod q) =$
 $((y, xa, (xb * xa + q - xc) \bmod q, (x + xb) \bmod q), (x * ((y + q - xa) \bmod q) + q - xc) \bmod q, (x * y + q - (x * ((y + q - xa) \bmod q) + q - xc) \bmod q) \bmod q)$
 using *assms s1-s2* by *metis*

lemma *c1*:

assumes $e2 = (x + c1) \bmod q$
 and $x < q \quad c1 < q$
 shows $c1 = (e2 + q - x) \bmod q$
 proof-
 have $[e2 + q = x + c1] \pmod q$ by (*simp add: assms cong-def*)
 then have $[e2 + q - x = c1] \pmod q$
 proof-
 have $e2 + q \geq x$ using *assms* by *simp*
 then show *?thesis*
 by (*metis* $\langle [e2 + q = x + c1] \pmod q \rangle$ *cong-add-lcancel-nat le-add-diff-inverse*)
 qed
 then show *?thesis* by (*simp add: cong-def assms*)
 qed

lemma *c1-P2*:

assumes $xb < q \quad xa < q \quad xc < q \quad x < q$
shows $((y, xa, (xb * xa + q - xc) \text{ mod } q, (x + xb) \text{ mod } q), (x * ((y + q - xa) \text{ mod } q) + q - xc) \text{ mod } q, (x * y + q - (x * ((y + q - xa) \text{ mod } q) + q - xc) \text{ mod } q) \text{ mod } q) =$
 $((y, xa, (((x + xb) \text{ mod } q + q - x) \text{ mod } q * xa + q - xc) \text{ mod } q, (x + xb) \text{ mod } q), (x * ((y + q - xa) \text{ mod } q) + q - xc) \text{ mod } q, (x * y + q - (x * ((y + q - xa) \text{ mod } q) + q - xc) \text{ mod } q) \text{ mod } q)$
proof –
have $(xb * xa + q - xc) \text{ mod } q = (((x + xb) \text{ mod } q + q - x) \text{ mod } q * xa + q - xc) \text{ mod } q$
using *assms c1* **by** *simp*
then show *?thesis*
using *assms* **by** *metis*
qed

lemma *minus-mod-cancel*:

assumes $a - b > 0 \quad a - b \text{ mod } q > 0$
shows $[a - b + c = a - b \text{ mod } q + c] \text{ (mod } q)$
proof –
have $(b - b \text{ mod } q + (a - b)) \text{ mod } q = (0 + (a - b)) \text{ mod } q$
using *cong-def mod-add-cong neq0-conv q-gt-0*
by (*simp add: minus-mod-eq-mult-div*)
then show *?thesis*
by (*metis (no-types) Divides.mod-less-eq-dividend Nat.add-diff-assoc2 add-diff-inverse-nat assms(1) cong-def diff-is-0-eq' less-or-eq-imp-le mod-add-cong neq0-conv*)
qed

lemma *d2*:

assumes $d2: d2 = (((e2 + q - x) \text{ mod } q) * b + (q - r)) \text{ mod } q$
and $s1: s1 = (x * ((y + (q - b)) \text{ mod } q) + (q - r)) \text{ mod } q$
and $s2: s2 = (x * y + (q - s1)) \text{ mod } q$
and $x: x < q$
and $y: y < q$
and $r: r < q$
and $b: b < q$
and $e2: e2 < q$
shows $d2 = (e2 * b + (q - s2)) \text{ mod } q$
proof –
have *s1-le-q: s1 < q*
using *s1 q-gt-0* **by** *simp*
have *s2-le-q: s2 < q*
using *s2 q-gt-0* **by** *simp*
have $xb: (x * b) \text{ mod } q = (s2 + (q - r)) \text{ mod } q$
proof –
have $s1 = (x * (y + (q - b)) + (q - r)) \text{ mod } q$ **using** *s1 b*
by (*metis mod-add-left-eq mod-mult-right-eq*)
then have *s1-dist: s1 = (x * y + x * (q - b) + (q - r)) mod q*
by (*metis distrib-left*)
then have $s1 = (x * y + x * q - x * b + (q - r)) \text{ mod } q$

```

    using distrib-left b diff-mult-distrib2 by auto
  then have [s1 = x*y + x*q - x*b + (q - r)] (mod q)
    by(simp add: cong-def)
  then have [s1 + x * b = x*y + x*q - x*b + x*b + (q - r)] (mod q)
    by (metis add.commute add.left-commute cong-add-lcancel-nat)
  then have [s1 + x*b = x*y + x*q + (q - r)] (mod q)
    using b by (simp add: algebra-simps)
    (metis add-diff-inverse-nat diff-diff-left diff-mult-distrib2 less-imp-add-positive
mult.commute not-add-less1 zero-less-diff)
  then have s1-xb: [s1 + x*b = q + x*y + x*q + (q - r)] (mod q)
    by (smt mod-add-cong mod-add-self1 cong-def)
  then have [x*b = q + x*y + x*q + (q - r) - s1] (mod q)
  proof-
    have q + x*y + x*q + (q - r) - s1 > 0 using s1-le-q by simp
    then show ?thesis
    by (metis add-diff-inverse-nat less-numeral-extra(3) s1-xb cong-add-lcancel-nat
nat-diff-split)
  qed
  then have [x*b = x*y + x*q + (q - r) + q - s1] (mod q)
    by (metis add.assoc add.commute)
  then have [x*b = x*y + (q - r) + q - s1] (mod q)
    by (smt Nat.add-diff-assoc cong-def less-imp-le-nat mod-mult-self1 s1-le-q
semiring-normalization-rules(23))
  then have (x*b) mod q = (x*y + (q - r) + q - s1) mod q
    by(simp add: cong-def)
  then have (x*b) mod q = (x*y + (q - r) + (q - s1)) mod q
    using add.assoc s1-le-q by auto
  then have (x*b) mod q = (x*y + (q - s1) + (q - r)) mod q
    using add.commute by presburger
  then show ?thesis using s2 by presburger
  qed
  have d2 = (((e2 + q - x) mod q)*b + (q - r)) mod q
    using d2 by simp
  then have d2 = (((e2 + q - x))*b + (q - r)) mod q
    using mod-add-cong mod-mult-left-eq by blast
  then have d2 = (e2*b + q*b - x*b + (q - r)) mod q
    by (simp add: distrib-right diff-mult-distrib)
  then have a: [d2 = e2*b + q*b - x*b + (q - r)] (mod q)
    by(simp add: cong-def)
  then have b:[d2 = q + q + e2*b + q*b - x*b + (q - r)] (mod q)
  proof-
    have [e2*b + q*b - x*b + (q - r) = q + q + e2*b + q*b - x*b + (q - r)]
(mod q)
    by (smt assms Nat.add-diff-assoc add.commute cong-def less-imp-le-nat mod-add-self2

    mult.commute nat-mult-le-cancel-disj semiring-normalization-rules(23))
  then show ?thesis using cong-trans a by blast
  qed
  then have [d2 = q + q + e2*b + q*b - (x*b) mod q + (q - r)] (mod q)

```

```

proof-
  have  $[q + q + e2*b + q*b - (x*b) + (q - r) = q + q + e2*b + q*b - (x*b)$ 
   $\text{mod } q + (q - r)] \text{ (mod } q)$ 
  proof(cases  $b = 0$ )
    case True
      then show ?thesis by simp
    next
      case False
        have  $q*b - (x*b) > 0$ 
          using False x by simp
        then have 1:  $q + q + e2*b + q*b - (x*b) > 0$  by linarith
        then have 2:  $q + q + e2*b + q*b - (x*b) \text{ mod } q > 0$ 
          by (simp add: q-gt-0 trans-less-add1)
        then show ?thesis using 1 2 minus-mod-cancel by simp
      qed
    then show ?thesis using cong-trans b by blast
  qed
  then have c:  $[d2 = q + q + e2*b + q*b - (s2 + (q - r)) \text{ mod } q + (q - r)]$ 
   $\text{(mod } q)$ 
    using xb by simp
    then have  $[d2 = q + q + e2*b + q*b - (s2 + (q - r)) + (q - r)] \text{ (mod } q)$ 
    proof-
      have  $[q + q + e2*b + q*b - (s2 + (q - r)) \text{ mod } q + (q - r) = q + q +$ 
       $e2*b + q*b - (s2 + (q - r)) + (q - r)] \text{ (mod } q)$ 
      proof-
        have  $q + q + e2*b + q*b - (s2 + (q - r)) \text{ mod } q > 0$ 
          by (metis diff-is-0-eq gr0I le-less-trans mod-less-divisor not-add-less1 q-gt-0
semiring-normalization-rules(23) trans-less-add2)
        moreover have  $q + q + e2*b + q*b - (s2 + (q - r)) > 0$ 
          using s2-le-q by simp
        ultimately show ?thesis
          using minus-mod-cancel cong-sym by blast
      qed
    then show ?thesis using cong-trans c by blast
  qed
  then have d:  $[d2 = q + q + e2*b + q*b - s2 - (q - r) + (q - r)] \text{ (mod } q)$ 
by simp
  then have  $[d2 = q + q + e2*b + q*b - s2] \text{ (mod } q)$ 
  proof-
    have  $q + q + e2*b + q*b - s2 - (q - r) > 0$ 
      using s2-le-q by simp
    then show ?thesis using d cong-trans by simp
  qed
  then have  $[d2 = q + q + e2*b - s2] \text{ (mod } q)$ 
    by (smt Nat.add-diff-assoc2 cong-def less-imp-le-nat mod-mult-self1 mult.commute
s2-le-q semiring-normalization-rules(23) trans-less-add2)
  then have  $[d2 = q + e2*b + q - s2] \text{ (mod } q)$ 
    by(simp add: add.commute add.assoc)
  then have  $[d2 = e2*b + q - s2] \text{ (mod } q)$ 

```

by (*smt Nat.add-diff-assoc2 add.commute cong-def less-imp-le-nat mod-add-self2 s2-le-q trans-less-add2*)
then have $[d2 = e2*b + (q - s2)] \text{ (mod } q)$
by (*simp add: less-imp-le-nat s2-le-q*)
then show *?thesis* **by** (*simp add: cong-def d2*)
qed

lemma *d2-P2*:

assumes *x: x < q and y: y < q and r: b < q and b: e2 < q and e2: r < q*
shows $((y, b, ((e2 + q - x) \text{ mod } q * b + q - r) \text{ mod } q, e2), (x * ((y + q - b) \text{ mod } q) + q - r) \text{ mod } q, (x * y + q - (x * ((y + q - b) \text{ mod } q) + q - r) \text{ mod } q) \text{ mod } q) =$
 $((y, b, (e2 * b + q - (x * y + q - (x * ((y + q - b) \text{ mod } q) + q - r) \text{ mod } q) \text{ mod } q) \text{ mod } q, e2), (x * ((y + q - b) \text{ mod } q) + q - r) \text{ mod } q,$
 $(x * y + q - (x * ((y + q - b) \text{ mod } q) + q - r) \text{ mod } q) \text{ mod } q)$

proof –

have $((e2 + q - x) \text{ mod } q * b + q - r) \text{ mod } q = (e2 * b + q - (x * y + q - (x * ((y + q - b) \text{ mod } q) + q - r) \text{ mod } q) \text{ mod } q) \text{ mod } q$
(is ?lhs = ?rhs)

proof –

have *d2*: $((e2 + q - x) \text{ mod } q) * b + (q - r) \text{ mod } q = (e2 * b + (q - ((x * y + (q - ((x * ((y + (q - b) \text{ mod } q) + (q - r) \text{ mod } q))) \text{ mod } q))) \text{ mod } q)$
using *assms d2 by blast*
have *?lhs* = $((e2 + q - x) \text{ mod } q) * b + (q - r) \text{ mod } q$
using *assms by simp*
also have *?rhs* = $(e2 * b + (q - ((x * y + (q - ((x * ((y + (q - b) \text{ mod } q) + (q - r) \text{ mod } q))) \text{ mod } q))) \text{ mod } q)$
using *assms by simp*
ultimately show *?thesis* **using** *assms d2 by metis*

qed

then show *?thesis* **using** *assms by metis*

qed

lemma *s1*:

assumes *s2: s2 = (x*y + q - s1) mod q*

and *x: x < q*

and *y: y < q*

and *s1: s1 < q*

shows *s1 = (x*y + q - s2) mod q*

proof –

have *s2-le-q:s2 < q* **using** *s2 q-gt-0 by simp*

have $[s2 = x*y + q - s1] \text{ (mod } q)$ **by** (*simp add: cong-def s2*)

then have $[s2 = x*y + q - s1] \text{ (mod } q)$ **using** *add.assoc*

by (*simp add: less-imp-le-nat s1*)

then have *s1-s2: [s2 + s1 = x*y + q] (mod q)*

by (*metis (mono-tags, lifting) cong-def le-add2 le-add-diff-inverse2 le-trans mod-add-left-eq order.strict-implies-order s1*)

then have $[s1 = x*y + q - s2] \text{ (mod } q)$

proof –

```

have  $x*y + q - s2 > 0$  using s2-le-q by simp
then show ?thesis
by (metis s1-s2 add-diff-cancel-left' cong-diff-nat cong-def le-add1 less-imp-le-nat
zero-less-diff)
qed
then show ?thesis by (simp add: cong-def s1)
qed

```

lemma *s1-P2*:

```

assumes  $x: x < q$ 
and  $y: y < q$ 
and  $b < q$ 
and  $e2 < q$ 
and  $r < q$ 
and  $s1 < q$ 
shows  $((y, b, (e2 * b + q - (x * y + q - r) \bmod q) \bmod q, e2), r, (x * y + q - r) \bmod q) =$ 
 $((y, b, (e2 * b + q - (x * y + q - r) \bmod q) \bmod q, e2), (x * y + q - (x * y + q - r) \bmod q) \bmod q, (x * y + q - r) \bmod q)$ 
proof -
have  $s1 = (x*y + q - ((x*y + q - s1) \bmod q)) \bmod q$ 
using assms secure-mult.s1 secure-mult-axioms by blast
then show ?thesis using assms s1 by blast
qed

```

theorem *P2-security*:

```

assumes  $x < q$   $y < q$ 
shows sim-non-det-def.perfect-sec-P2  $x$   $y$ 
including monad-normalisation
proof -
have  $((\text{funct } x \ y) \gg (\lambda (s1', s2'). (\text{sim-non-det-def.Ideal2 } y \ x \ s2')) = R2 \ x \ y)$ 
proof -
have  $R2 \ x \ y = \text{do } \{$ 
 $a :: \text{nat} \leftarrow \text{sample-uniform } q;$ 
 $b :: \text{nat} \leftarrow \text{sample-uniform } q;$ 
 $r :: \text{nat} \leftarrow \text{sample-uniform } q;$ 
 $\text{let } c1 = a;$ 
 $\text{let } d1 = r;$ 
 $\text{let } c2 = b;$ 
 $\text{let } d2 = ((a*b + (q - r)) \bmod q);$ 
 $\text{let } e2 = (x + c1) \bmod q;$ 
 $\text{let } e1 = (y + (q - c2)) \bmod q;$ 
 $\text{let } s1 = (x*e1 + (q - r)) \bmod q;$ 
 $\text{let } s2 = (e2 * c2 + (q - d2)) \bmod q;$ 
 $\text{return-spmf } ((y, c2, d2, e2), s1, s2)\}$ 
by (simp add: R2-def TI-def Let-def)
also have  $\dots = \text{do } \{$ 
 $a :: \text{nat} \leftarrow \text{sample-uniform } q;$ 
 $b :: \text{nat} \leftarrow \text{sample-uniform } q;$ 

```

```

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

```

```

    e2 ← sample-uniform q;
    s1 ← map-spmf (λ r. (x*((y + (q - b)) mod q) + (q - r)) mod q)
(sample-uniform q);
    let s2 = (x*y + (q - s1)) mod q;
    let d2 = (e2*b + (q - s2)) mod q;
    return-spmf ((y, b, d2, e2), s1, s2)}
  by(simp add: bind-map-spmf o-def Let-def)
also have ... = do {
  b :: nat ← sample-uniform q;
  e2 ← sample-uniform q;
  s1 ← sample-uniform q;
  let s2 = (x*y + (q - s1)) mod q;
  let d2 = (e2*b + (q - s2)) mod q;
  return-spmf ((y, b, d2, e2), s1, s2)}
  by(simp add: samp-uni-minus-one-time-pad)
also have ... = do {
  b :: nat ← sample-uniform q;
  e2 ← sample-uniform q;
  s1 ← sample-uniform q;
  let s2 = (x*y + (q - s1)) mod q;
  let d2 = (e2*b + (q - s2)) mod q;
  return-spmf ((y, b, d2, e2), (x*y + (q - s2)) mod q, s2)}
  by(simp add: s1-P2 assms Let-def cong: bind-spmf-cong-simp)
ultimately show ?thesis by(simp add: funct-def Let-def sim-non-det-def.Ideal2-def
Out2-def S2-def R2-def)
qed
then show ?thesis by(simp add: sim-non-det-def.perfect-sec-P2-def)
qed

```

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

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

lemma r:
assumes $e1: e1 = (y + (q - b)) \bmod q$
and $s1: s1 = (x*((y + (q - b)) \bmod q) + (q - r)) \bmod q$
and $b: b < q$
and $x: x < q$
and $y: y < q$
and $r: r < q$
shows $r = (x*e1 + (q - s1)) \bmod q$


```

    (is ?lhs = ?rhs)
  proof -
    have s1 = (x*((y + (q - b))) + (q - r)) mod q using s1 b
      by (metis mod-add-left-eq mod-mult-right-eq)
    then have s1-dist: s1 = (x*y + x*(q - b) + (q - r)) mod q
      by (metis distrib-left)
    then have ?rhs = (x*((y + (q - b)) mod q) + (q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
      using e1 by simp
    then have ?rhs = (x*((y + (q - b))) + (q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
      by (metis mod-add-left-eq mod-mult-right-eq)
    then have ?rhs = (x*y + x*(q - b) + (q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
      by (metis distrib-left)
    then have a: ?rhs = (x*y + x*q - x*b + (q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
      using distrib-left b diff-mult-distrib2 by auto
    then have b: ?rhs = (x*y + x*q - x*b + (q*q + q*q + q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
      proof -
        have (x*y + x*q - x*b + (q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
          = (x*y + x*q - x*b + (q*q + q*q + q - (x*y + x*(q - b) + (q - r)) mod q)) mod q
          mod q
        proof -
          have f1:  $\forall n na nb nc nd. (n::nat) \text{ mod } na \neq nb \text{ mod } na \vee nc \text{ mod } na \neq nd \text{ mod } na \vee (n + nc) \text{ mod } na = (nb + nd) \text{ mod } na$ 
            by (meson mod-add-cong)
          then have (q - (x * y + x*(q - b) + (q - r)) mod q) mod q = (q * q + q * q + q - (x * y + x*(q - b) + (q - r)) mod q) mod q
            by (metis Nat.diff-add-assoc mod-le-divisor q-gt-0 mod-mult-self4)
          then show ?thesis
            using f1 by blast
        qed
        then show ?thesis using a by simp
      qed
    then have ?rhs = (x*y + x*q - x*b + (q*q + q*q + q - (x*y + x*(q - b) + (q - r)))) mod q
      proof -
        have (x*y + x*q - x*b + (q*q + q*q + q - (x*y + x*(q - b) + (q - r)) mod q)) mod q =
          (x*y + x*q - x*b + (q*q + q*q + q - (x*y + x*(q - b) + (q - r)))) mod q
        proof (cases x = 0)
          case True
            then show ?thesis
              by (metis (no-types, lifting) assms(2) assms(4) True Nat.add-diff-assoc add.left-neutral cong-def diff-le-self minus-mod mult-is-0 not-gr-zero zero-eq-add-iff-both-eq-0)
        qed
      qed
  
```

```

zero-less-diff)
next
  case False
  have qb:  $q - b \leq q$ 
    using b by simp
  then have qb':  $x*(q - b) < q*q$ 
  using x by (metis mult-less-le-imp-less nat-0-less-mult-iff nat-less-le neq0-conv)

  have a:  $x*y + x*(q - b) > 0$ 
    using False assms by simp
  have 1:  $q*q > x*y$ 
    using False by (simp add: mult-strict-mono q-gt-0 x y)
  have 2:  $q*q > x*q$  using False
    by (simp add: mult-strict-mono q-gt-0 x y)
  have b:  $(q*q + q*q + q - (x*y + x*(q - b) + (q - r))) > 0$ 
    using 1 qb' by simp
  then show ?thesis using a b mod-minus[of x*y + x*q x*b q*q + q*q + q
x*y + x*(q - b) + (q - r)]
    by (smt add.left-neutral cong-def grOI minus-mod zero-less-diff)
  qed
  then show ?thesis using b by simp
  qed
  then have d: ?rhs =  $(x*y + x*q - x*b + (q*q + q*q + q - x*y - x*(q - b)$ 
 $- (q - r))) \bmod q$ 
    by simp
  then have e: ?rhs =  $(x*y + x*q - x*b + q*q + q*q + q - x*y - x*(q - b)$ 
 $- (q - r)) \bmod q$ 
  proof(cases x = 0)
    case True
    then show ?thesis using True d by simp
  next
  case False
  have qb:  $q - b \leq q$  using b by simp
  then have qb':  $x*(q - b) < q*q$ 
  using x by (metis mult-less-le-imp-less nat-0-less-mult-iff nat-less-le neq0-conv)

  have a:  $x*y + x*(q - b) > 0$  using False assms by simp
  have 1:  $q*q > x*y$  using False
    by (simp add: mult-strict-mono q-gt-0 x y)
  have 2:  $q*q > x*q$  using False
    by (simp add: mult-strict-mono q-gt-0 x y)
  have b:  $q*q + q*q + q - x*y - x*(q - b) - (q - r) > 0$  using 1 qb' by
simp
  then show ?thesis using b d
    by (smt Nat.add-diff-assoc add.assoc diff-diff-left less-imp-le-nat zero-less-diff)
  qed
  then have ?rhs =  $(x*q - x*b + q*q + q*q + q - x*(q - b) - (q - r)) \bmod q$ 
  proof-
    have  $(x*y + x*q - x*b + q*q + q*q + q - x*y - x*(q - b) - (q - r)) \bmod$ 

```

```

q = (x*q - x*b + q*q + q*q + q - x*(q - b) - (q - r)) mod q
proof -
  have 1: q + n - b = q - b + n for n
    by (simp add: assms(3) less-imp-le)
  have 2: (c::nat) * b + (c * a + n) = c * (b + a) + n
    for n a b c by (simp add: distrib-left)
  have (c::nat) + (b + a) - (n + a) = c + b - n for n a b c
    by auto
  then have (q + (q * q + (q * q + x * (q + y - b)))) - (q - r + x * (y +
(q - b)))) mod q = (q + (q * q + (q * q + x * (q - b)))) - (q - r + x * (q -
b))) mod q
    by (metis (no-types) add commute 1 2)
  then show ?thesis
    by (simp add: add commute diff-mult-distrib2 distrib-left)
qed
then show ?thesis using e by simp
qed
then have ?rhs = (x*(q - b) + q*q + q*q + q - x*(q - b) - (q - r)) mod q
  by(metis diff-mult-distrib2)
then have ?rhs = (q*q + q*q + q - (q - r)) mod q
  using assms(6) by simp
then have ?rhs = (q*q + q*q + q - q + r) mod q
  using assms(6) by(simp add: Nat.diff-add-assoc2 less-imp-le)
then have ?rhs = (q*q + q*q + r) mod q
  by simp
then have ?rhs = r mod q
  by (metis add commute distrib-right mod-mult-self1)
then show ?thesis using assms(6) by simp
qed

```

lemma r-P2:

assumes b: $b < q$ **and** x: $x < q$ **and** y: $y < q$ **and** r: $r < q$

shows

$((x, a, r, (y + q - b) \bmod q), (x * ((y + q - b) \bmod q) + q - r) \bmod q, (x * y + q - (x * ((y + q - b) \bmod q) + q - r) \bmod q) \bmod q) =$
 $((x, a, (x * ((y + q - b) \bmod q) + q - (x * ((y + q - b) \bmod q) + q - r) \bmod q) \bmod q, (y + q - b) \bmod q), (x * ((y + q - b) \bmod q) + q - r) \bmod q,$
 $(x * y + q - (x * ((y + q - b) \bmod q) + q - r) \bmod q) \bmod q)$

proof -

have $(x * ((y + q - b) \bmod q) + q - (x * ((y + q - b) \bmod q) + q - r) \bmod q) \bmod q = r$
(is ?lhs = ?rhs)

proof -

have 1: $r = (x * ((y + (q - b)) \bmod q) + (q - ((x * ((y + (q - b)) \bmod q) + (q - r)) \bmod q))) \bmod q$

using assms secure-mult.r secure-mult-axioms **by** blast

also have ?rhs = $(x * ((y + (q - b)) \bmod q) + (q - ((x * ((y + (q - b)) \bmod q) + (q - r)) \bmod q))) \bmod q$ **using** assms 1 **by** blast

ultimately show *?thesis using assms 1 by simp*
qed
then show *?thesis using assms by simp*
qed

theorem *P1-security:*

assumes $x < q \ y < q$
shows *sim-non-det-def.perfect-sec-P1 x y*
including *monad-normalisation*

proof –

have $(\text{funct } x \ y) \gg (\lambda (s1', s2'). (\text{sim-non-det-def.Ideal1 } x \ y \ s1')) = R1 \ x \ y$

proof –

have $R1 \ x \ y = \text{do } \{$

$a :: \text{nat} \leftarrow \text{sample-uniform } q;$

$b :: \text{nat} \leftarrow \text{sample-uniform } q;$

$r :: \text{nat} \leftarrow \text{sample-uniform } q;$

$\text{let } c1 = a;$

$\text{let } d1 = r;$

$\text{let } c2 = b;$

$\text{let } d2 = ((a*b + (q - r)) \text{ mod } q);$

$\text{let } e2 = (x + c1) \text{ mod } q;$

$\text{let } e1 = (y + (q - c2)) \text{ mod } q;$

$\text{let } s1 = (x*e1 + (q - d1)) \text{ mod } q;$

$\text{let } s2 = (e2 * c2 + (q - d2)) \text{ mod } q;$

$\text{return-spmf } ((x, c1, d1, e1), s1, s2)\}$

by(*simp add: R1-def TI-def Let-def*)

also have $\dots = \text{do } \{$

$a :: \text{nat} \leftarrow \text{sample-uniform } q;$

$b :: \text{nat} \leftarrow \text{sample-uniform } q;$

$r :: \text{nat} \leftarrow \text{sample-uniform } q;$

$\text{let } c1 = a;$

$\text{let } c2 = b;$

$\text{let } e1 = (y + (q - b)) \text{ mod } q;$

$\text{let } s1 = (x*((y + (q - b)) \text{ mod } q) + (q - r)) \text{ mod } q;$

$\text{let } d1 = (x*e1 + (q - s1)) \text{ mod } q;$

$\text{return-spmf } ((x, c1, d1, e1), s1, (x*y + (q - s1)) \text{ mod } q)\}$

by(*simp add: Let-def assms s1-s2-P1 r-P2 cong: bind-spmf-cong-simp*)

also have $\dots = \text{do } \{$

$a :: \text{nat} \leftarrow \text{sample-uniform } q;$

$b :: \text{nat} \leftarrow \text{sample-uniform } q;$

$\text{let } c1 = a;$

$\text{let } c2 = b;$

$\text{let } e1 = (y + (q - b)) \text{ mod } q;$

$s1 \leftarrow \text{map-spmf } (\lambda r. (x*((y + (q - b)) \text{ mod } q) + (q - r)) \text{ mod } q) (\text{sample-uniform } q);$

$\text{let } d1 = (x*e1 + (q - s1)) \text{ mod } q;$

$\text{return-spmf } ((x, c1, d1, e1), s1, (x*y + (q - s1)) \text{ mod } q)\}$

by(*simp add: bind-map-spmf Let-def o-def*)

also have $\dots = \text{do } \{$

```

a :: nat ← sample-uniform q;
b :: nat ← sample-uniform q;
let c1 = a;
let c2 = b;
let e1 = (y + (q - b)) mod q;
s1 ← sample-uniform q;
let d1 = (x*e1 + (q - s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x*y + (q - s1)) mod q)}
  by(simp add: samp-uni-minus-one-time-pad)
also have ... = do {
a :: nat ← sample-uniform q;
let c1 = a;
e1 ← map-spmf (λb. (y + (q - b)) mod q) (sample-uniform q);
s1 ← sample-uniform q;
let d1 = (x*e1 + (q - s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x*y + (q - s1)) mod q)}
  by(simp add: bind-map-spmf Let-def o-def)
also have ... = do {
a :: nat ← sample-uniform q;
let c1 = a;
e1 ← sample-uniform q;
s1 ← sample-uniform q;
let d1 = (x*e1 + (q - s1)) mod q;
return-spmf ((x, c1, d1, e1), s1, (x*y + (q - s1)) mod q)}
  by(simp add: samp-uni-minus-one-time-pad)
ultimately show ?thesis by(simp add: funct-def sim-non-det-def.Ideal1-def
Let-def R1-def TI-def Out1-def S1-def)
qed
thus ?thesis by(simp add: sim-non-det-def.perfect-sec-P1-def)
qed

end

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

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

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

theorem P2-secure:
  assumes x < q n y < q n
  shows sim-non-det-def.perfect-sec-P2 n x y

```

by (*metis P2-security assms*)

end

end

2.9 DHH Extension

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

theory *DH-Ext* **imports**

Game-Based-Crypto.Diffie-Hellman

Cyclic-Group-Ext

begin

context *ddh* **begin**

definition *DDH0* :: 'grp adversary \Rightarrow bool spmf

where *DDH0* $\mathcal{A} = do \{$

$s \leftarrow sample\text{-uniform} (order \mathcal{G});$

$r \leftarrow sample\text{-uniform} (order \mathcal{G});$

$let h = \mathbf{g} [\wedge] s;$

$\mathcal{A} h (\mathbf{g} [\wedge] r) (h [\wedge] r)\}$

definition *DDH1* :: 'grp adversary \Rightarrow bool spmf

where *DDH1* $\mathcal{A} = do \{$

$s \leftarrow sample\text{-uniform} (order \mathcal{G});$

$r \leftarrow sample\text{-uniform} (order \mathcal{G});$

$let h = \mathbf{g} [\wedge] s;$

$\mathcal{A} h (\mathbf{g} [\wedge] r) ((h [\wedge] r) \otimes \mathbf{g})\}$

definition *DDH-advantage* :: 'grp adversary \Rightarrow real

where *DDH-advantage* $\mathcal{A} = |spmf (DDH0 \mathcal{A}) True - spmf (DDH1 \mathcal{A}) True|$

definition *DDH-A'* :: 'grp adversary \Rightarrow 'grp \Rightarrow 'grp \Rightarrow 'grp \Rightarrow bool spmf

where *DDH-A'* $D\text{-ddh } a b c = D\text{-ddh } a b (c \otimes \mathbf{g})$

end

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

begin

lemma *DDH0-eq-ddh-0*: *ddh.DDH0* $\mathcal{G} \mathcal{A} = ddh.ddh\text{-}0 \mathcal{G} \mathcal{A}$

by(*simp add: ddh.DDH0-def Let-def monoid.nat-pow-pow ddh.ddh-0-def*)

lemma *DDH-bound1*: $|spmf (ddh.DDH0 \mathcal{G} \mathcal{A}) True - spmf (ddh.DDH1 \mathcal{G} \mathcal{A}) True|$

$$\leq |spmf (ddh.ddh\text{-}0 \mathcal{G} \mathcal{A}) True - spmf (ddh.ddh\text{-}1 \mathcal{G} \mathcal{A}) True| \\ + |spmf (ddh.ddh\text{-}1 \mathcal{G} \mathcal{A}) True - spmf (ddh.DDH1 \mathcal{G} \mathcal{A}) True|$$

$True|$
by (*simp add: abs-diff-triangle-ineq2 DDH0-eq-ddh-0*)

lemma *DDH-bound2*:

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

lemma *rewrite*:

shows $(sample-uniform (order \mathcal{G}) \gg (\lambda x. sample-uniform (order \mathcal{G})$
 $\gg (\lambda y. sample-uniform (order \mathcal{G}) \gg (\lambda z. \mathcal{A} (\mathbf{g} [\uparrow] x) (\mathbf{g} [\uparrow] y) (\mathbf{g} [\uparrow] z$
 $\otimes \mathbf{g}))))$
 $= (sample-uniform (order \mathcal{G}) \gg (\lambda x. sample-uniform (order \mathcal{G})$
 $\gg (\lambda y. sample-uniform (order \mathcal{G}) \gg (\lambda z. \mathcal{A} (\mathbf{g} [\uparrow] x) (\mathbf{g} [\uparrow] y) (\mathbf{g}$
 $[\uparrow] z))))$
(is ?lhs = ?rhs)

proof –

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

qed

lemma *DDH-A'-bound*: $ddh.advantage \mathcal{G} (ddh.DDH-A' \mathcal{G} \mathcal{A}) = |spmf (ddh.ddh-1$
 $\mathcal{G} \mathcal{A}) True - spmf (ddh.DDH1 \mathcal{G} \mathcal{A}) True|$

unfolding *ddh.advantage-def ddh.ddh-1-def ddh.DDH1-def Let-def ddh.DDH-A'-def*
 $ddh.ddh-0-def$

by (*simp add: rewrite abs-minus-commute nat-pow-pow*)

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

using *DDH-bound2 DDH-A'-bound DDH-advantage-def by simp*

end

end

3 Malicious Security

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

3.1 Malicious Security Definitions

theory *Malicious-Defs* **imports**

CryptHOL.CryptHOL

begin

type-synonym ('in1', 'aux', 'P1-S1-aux') *P1-ideal-adv1* = 'in1' \Rightarrow 'aux' \Rightarrow ('in1' \times 'P1-S1-aux') *spmf*

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

type-synonym ('in1', 'aux', 'out1', 'P1-S1-aux', 'adv-out1') *P1-ideal-adv* = ('in1', 'aux', 'P1-S1-aux') *P1-ideal-adv1* \times ('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' \Rightarrow 'in1' \Rightarrow 'aux' \Rightarrow ('in1' \times 'P1-S1-aux') *spmf*

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

type-synonym ('in2', 'aux', 'P2-S2-aux') *P2-ideal-adv1* = 'in2' \Rightarrow 'aux' \Rightarrow ('in2' \times 'P2-S2-aux') *spmf*

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

type-synonym ('in2', 'aux', 'out2', 'P2-S2-aux', 'adv-out2') *P2-ideal-adv*
 = ('in2', 'aux', 'P2-S2-aux') *P2-ideal-adv1*
 \times ('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' \Rightarrow 'in2' \Rightarrow 'aux' \Rightarrow ('in2' \times 'P2-S2-aux') *spmf*

type-synonym ('P2-real-adv', 'in2', 'aux', 'out2', 'P2-S2-aux', 'adv-out2') P2-sim2
= 'P2-real-adv' \Rightarrow 'in2' \Rightarrow 'aux' \Rightarrow 'out2'
 \Rightarrow 'P2-S2-aux' \Rightarrow '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
 \times ('P2-real-adv', 'in2', 'aux', 'out2', 'P2-S2-aux', 'adv-out2')
P2-sim2)

locale *malicious-base* =
fixes *funct* :: 'in1 \Rightarrow 'in2 \Rightarrow ('out1 \times 'out2) *spmf* — the functionality
and *protocol* :: 'in1 \Rightarrow 'in2 \Rightarrow ('out1 \times '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 \Rightarrow 'in2 \Rightarrow 'aux' \Rightarrow 'P1-real-adv' \Rightarrow ('adv-out1 \times '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 \Rightarrow 'in2 \Rightarrow 'aux' \Rightarrow 'P2-real-adv' \Rightarrow ('out1 \times 'adv-out2)
spmf — real view for party 2, the adversary totally controls party 2
begin

definition *correct* m1 m2 \longleftrightarrow (protocol m1 m2 = *funct* m1 m2)

abbreviation *trusted-party* x y \equiv *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 auxiliary 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 \Rightarrow 'in2 \Rightarrow 'aux' \Rightarrow ('in1', 'aux', 'out1', 'P1-S1-aux',
'adv-out1') P1-ideal-adv \Rightarrow ('adv-out1' \times 'out2) *spmf*
where *ideal-game-1* x y z A = do {
let (A1,A2) = A;
(x', aux-out) \leftarrow A1 x z;
(out1, out2) \leftarrow *trusted-party* x' y;
out1' :: 'adv-out1' \leftarrow A2 x' z out1 aux-out;
return-*spmf* (out1', out2)}

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

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

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

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

definition $adv-P1\ x\ y\ z\ S\ \mathcal{A}\ (D :: ('adv-out1 \times 'out2) \Rightarrow \text{bool spmf}) =$
 $| \text{ spmf } (P1-real-view\ x\ y\ z\ \mathcal{A} \ggg (\lambda\ view.\ D\ view))\ \text{True}$
 $- \text{ spmf } (ideal-view-1\ x\ y\ z\ S\ \mathcal{A} \ggg (\lambda\ view.\ D\ view))\ \text{True} |$

definition $ideal-game-2 :: 'in1 \Rightarrow 'in2 \Rightarrow 'aux \Rightarrow ('in2, 'aux, 'out2, 'P2-S2-aux, 'adv-out2) P2-ideal-adv \Rightarrow ('out1 \times 'adv-out2) \text{ spmf}$
where $ideal-game-2\ x\ y\ z\ A = \text{do } \{$
 $\text{let } (A1, A2) = A;$
 $(y', aux-out) \leftarrow A1\ y\ z;$
 $(out1, out2) \leftarrow \text{trusted-party}\ x\ y';$
 $out2' :: 'adv-out2 \leftarrow A2\ y'\ z\ out2\ aux-out;$
 $\text{return-spmf } (out1, out2') \}$

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

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

definition $adv-P2\ x\ y\ z\ S\ \mathcal{A}\ (D :: ('out1 \times 'adv-out2) \Rightarrow \text{bool spmf}) =$
 $| \text{ spmf } (P2-real-view\ x\ y\ z\ \mathcal{A} \ggg (\lambda\ view.\ D\ view))\ \text{True}$
 $- \text{ spmf } (ideal-view-2\ x\ y\ z\ S\ \mathcal{A} \ggg (\lambda\ view.\ D\ view))\ \text{True} |$

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' ⇒ 'aux ⇒ (('grp' × 'grp' × 'grp') × 'state) spmf*

type-synonym (*'grp', 'state*) *adv-2-P1* = *'grp' ⇒ '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 × ('adv-out2', 'state) adv-3-P2*)

locale *ot-base* =

fixes *G* :: *'grp cyclic-group (structure)*

assumes *finite-group: finite (carrier G)*

and *order-gt-0: order G > 0*

and *prime-order: prime (order G)*

begin

lemma *prime-field: a < (order G) ⇒ a ≠ 0 ⇒ coprime a (order G)*

by (*metis dvd-imp-le neq0-conv not-le prime-imp-coprime prime-order coprime-commute*)

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

fun *funct-DH-ZK* :: (*'grp × 'grp × 'grp*) ⇒ (*(('grp × 'grp × 'grp) × nat) ⇒ (bool × unit) spmf*

where *funct-DH-ZK (h,a,b) ((h',a',b'),r) = return-spmf (a = **g** [∗] r ∧ b = h*

$$[\ulcorner] r \wedge (h, a, b) = (h', a', b'), (())$$

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') ←  $\mathcal{A}2$  h0 h1 a b0 b1 M s;
adv-out :: 'adv-out1 ←  $\mathcal{A}3$  s';
return-spmf (adv-out, (if  $\sigma$  then (z1  $\otimes$  (inv w1 [ $\uparrow$ ]  $\alpha1$ )) else (z0  $\otimes$  (inv w0 [ $\uparrow$ ]  $\alpha0$ ))))}

```

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

definition $P1-S1 :: ('aux, 'grp, 'adv-out1, 'state) \text{adv-mal-}P1 \Rightarrow ('grp \times 'grp) \Rightarrow 'aux \Rightarrow (('grp \times 'grp) \times 'state) \text{spmfm}$

```

where  $P1-S1 \mathcal{A} M z = \text{do}$  {
  let ( $\mathcal{A}1, \mathcal{A}2, \mathcal{A}3$ ) =  $\mathcal{A}$ ;
  r ← sample-uniform (order  $\mathcal{G}$ );
   $\alpha0$  ← sample-uniform (order  $\mathcal{G}$ );
   $\alpha1$  ← sample-uniform (order  $\mathcal{G}$ );
  let h0 =  $\mathbf{g}$  [ $\uparrow$ ]  $\alpha0$ ;
  let h1 =  $\mathbf{g}$  [ $\uparrow$ ]  $\alpha1$ ;
  let a =  $\mathbf{g}$  [ $\uparrow$ ] r;
  let b0 = h0 [ $\uparrow$ ] r;
  let b1 = h1 [ $\uparrow$ ] r  $\otimes$   $\mathbf{g}$ ;
  ((in1 :: 'grp, in2 :: 'grp, in3 :: 'grp), s) ←  $\mathcal{A}1$  M h0 h1 a b0 b1 z;
  let (h,a,b) = (h0  $\otimes$  inv h1, a, b0  $\otimes$  inv b1);
  - :: unit ← assert-spmf ((in1, in2, in3) = (h,a,b));
  (((w0,z0),(w1,z1)),s') ←  $\mathcal{A}2$  h0 h1 a b0 b1 M s;
  let x0 = (z0  $\otimes$  (inv w0 [ $\uparrow$ ]  $\alpha0$ ));
  let x1 = (z1  $\otimes$  (inv w1 [ $\uparrow$ ]  $\alpha1$ ));
  return-spmf ((x0,x1), s')}

```

definition $P1-S2 :: ('aux, 'grp, 'adv-out1, 'state) \text{adv-mal-}P1 \Rightarrow ('grp \times 'grp) \Rightarrow 'aux \Rightarrow \text{unit} \Rightarrow 'state \Rightarrow 'adv-out1 \text{spmfm}$

```

where  $P1-S2 \mathcal{A} M z \text{out1 } s' = \text{do}$  {
  let ( $\mathcal{A}1, \mathcal{A}2, \mathcal{A}3$ ) =  $\mathcal{A}$ ;
   $\mathcal{A}3$  s'}

```

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

definition $P1\text{-ideal-model} :: ('grp \times 'grp) \Rightarrow \text{bool} \Rightarrow 'aux \Rightarrow ('aux, 'grp, 'adv-out1, 'state) \text{adv-mal-}P1 \Rightarrow ('adv-out1 \times 'grp) \text{spmfm}$

```

where  $P1\text{-ideal-model } M \sigma z \mathcal{A} = \text{do}$  {
  let ( $\mathcal{A}1, \mathcal{A}2, \mathcal{A}3$ ) =  $\mathcal{A}$ ;
  r ← sample-uniform (order  $\mathcal{G}$ );
   $\alpha0$  ← sample-uniform (order  $\mathcal{G}$ );
   $\alpha1$  ← sample-uniform (order  $\mathcal{G}$ );
  let h0 =  $\mathbf{g}$  [ $\uparrow$ ]  $\alpha0$ ;
  let h1 =  $\mathbf{g}$  [ $\uparrow$ ]  $\alpha1$ ;
  let a =  $\mathbf{g}$  [ $\uparrow$ ] r;
  let b0 = h0 [ $\uparrow$ ] r;
  let b1 = h1 [ $\uparrow$ ] r  $\otimes$   $\mathbf{g}$ ;
  ((in1 :: 'grp, in2 :: 'grp, in3 :: 'grp), s) ←  $\mathcal{A}1$  M h0 h1 a b0 b1 z;
  let (h,a,b) = (h0  $\otimes$  inv h1, a, b0  $\otimes$  inv b1);

```

```

- :: unit ← assert-spmf ((in1, in2, in3) = (h,a,b));
(((w0,z0),(w1,z1)),s') ←  $\mathcal{A}2$  h0 h1 a b0 b1 M s;
let x0' = z0 ⊗ inv w0 [∇] α0;
let x1' = z1 ⊗ inv w1 [∇] α1;
(-, f-out2) ← funct-OT-12 (x0', x1') σ;
adv-out :: 'adv-out1 ←  $\mathcal{A}3$  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  $\mathcal{A}$  z
= |spmf ((P1-real-model M σ z  $\mathcal{A}$ ) ≫ (λ view. D view)) True
  - spmf ((P1-ideal-model M σ z  $\mathcal{A}$ ) ≫ (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 \mathcal{A} = do {

```

  let (x0,x1) = M;
  let ( $\mathcal{A}1$ ,  $\mathcal{A}2$ ,  $\mathcal{A}3$ ) =  $\mathcal{A}$ ;
  ((h0,h1,a,b0,b1),s) ←  $\mathcal{A}1$  σ z;
  - :: unit ← assert-spmf (h0 ∈ carrier  $\mathcal{G}$  ∧ h1 ∈ carrier  $\mathcal{G}$  ∧ a ∈ carrier  $\mathcal{G}$  ∧
b0 ∈ carrier  $\mathcal{G}$  ∧ b1 ∈ carrier  $\mathcal{G}$ );
  (((in1, in2, in3 :: 'grp), r),s') ←  $\mathcal{A}2$  (h0,h1,a,b0,b1) s;
  let (h,a,b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
  (out-zk-funct, -) ← funct-DH-ZK (h,a,b) ((in1, in2, in3), r);
  - :: unit ← assert-spmf out-zk-funct;
  u0 ← sample-uniform (order  $\mathcal{G}$ );
  u1 ← sample-uniform (order  $\mathcal{G}$ );
  v0 ← sample-uniform (order  $\mathcal{G}$ );
  v1 ← sample-uniform (order  $\mathcal{G}$ );
  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 ←  $\mathcal{A}3$  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 \mathcal{A} σ z = do {

```

  let ( $\mathcal{A}1$ ,  $\mathcal{A}2$ ,  $\mathcal{A}3$ ) =  $\mathcal{A}$ ;
  ((h0,h1,a,b0,b1),s) ←  $\mathcal{A}1$  σ z;
  - :: unit ← assert-spmf (h0 ∈ carrier  $\mathcal{G}$  ∧ h1 ∈ carrier  $\mathcal{G}$  ∧ a ∈ carrier  $\mathcal{G}$  ∧
b0 ∈ carrier  $\mathcal{G}$  ∧ b1 ∈ carrier  $\mathcal{G}$ );
  (((in1, in2, in3 :: 'grp), r),s') ←  $\mathcal{A}2$  (h0,h1,a,b0,b1) s;

```

```

let (h,a,b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
(out-zk-funct, -) ← funct-DH-ZK (h,a,b) ((in1, in2, in3), r);
- :: unit ← assert-spmf out-zk-funct;
let l = b0 ⊗ (inv (h0 [∧] r));
return-spmf ((if l = 1 then False else True), (h0,h1,a,b0,b1), s')

```

definition $P2-S2 :: ('aux, 'grp, 'adv-out2, 'state) \text{adv-mal-}P2 \Rightarrow \text{bool} \Rightarrow 'aux \Rightarrow 'grp \Rightarrow ((('grp \times 'grp \times 'grp \times 'grp \times 'grp) \times 'state) \Rightarrow 'adv-out2 \text{ spmf})$
where $P2-S2 \mathcal{A} \sigma' z x\sigma \text{aux-out} = \text{do} \{$
 $\text{let } (\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A};$
 $\text{let } ((h0,h1,a,b0,b1),s) = \text{aux-out};$
 $u0 \leftarrow \text{sample-uniform (order } \mathcal{G});$
 $v0 \leftarrow \text{sample-uniform (order } \mathcal{G});$
 $u1 \leftarrow \text{sample-uniform (order } \mathcal{G});$
 $v1 \leftarrow \text{sample-uniform (order } \mathcal{G});$
 $\text{let } w0 = a [\wedge] u0 \otimes \mathbf{g} [\wedge] v0;$
 $\text{let } w1 = a [\wedge] u1 \otimes \mathbf{g} [\wedge] v1;$
 $\text{let } z0 = b0 [\wedge] u0 \otimes h0 [\wedge] v0 \otimes (\text{if } \sigma' \text{ then } \mathbf{1} \text{ else } x\sigma);$
 $\text{let } z1 = (b1 \otimes \text{inv } \mathbf{g}) [\wedge] u1 \otimes h1 [\wedge] v1 \otimes (\text{if } \sigma' \text{ then } x\sigma \text{ else } \mathbf{1});$
 $\text{let } e0 = (w0, z0);$
 $\text{let } e1 = (w1, z1);$
 $\mathcal{A}3 \ e0 \ e1 \ s\}$

sublocale $\text{mal-def} : \text{malicious-base funct-OT-12 protocol-ot } P1-S1 \ P1-S2 \ P1\text{-real-model } P2-S1 \ P2-S2 \ P2\text{-real-model} .$

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

lemma $P1\text{-ideal-ideal-eq}:$

shows $\text{mal-def.ideal-view-1 } x \ y \ z \ (P1-S1, P1-S2) \ \mathcal{A} = P1\text{-ideal-model } x \ y \ z \ \mathcal{A}$
including $\text{monad-normalisation}$
unfolding $\text{mal-def.ideal-view-1-def mal-def.ideal-game-1-def } P1\text{-ideal-model-def } P1-S1\text{-def } P1-S2\text{-def } \text{Let-def split-def}$
by($\text{simp add: split-def}$)

lemma $P1\text{-advantages-eq}:$

shows $\text{mal-def.adv-P1 } x \ y \ z \ (P1-S1, P1-S2) \ \mathcal{A} \ D = P1\text{-adv-real-ideal-model } D \ x \ y \ \mathcal{A} \ z$
by($\text{simp add: mal-def.adv-P1-def } P1\text{-adv-real-ideal-model-def } P1\text{-ideal-ideal-eq}$)

fun $P1\text{-DDH-mal-adv-}\sigma\text{-false} :: ('grp \times 'grp) \Rightarrow 'aux \Rightarrow ('aux, 'grp, 'adv-out1, 'state) \text{adv-mal-}P1 \Rightarrow ((('adv-out1 \times 'grp) \Rightarrow \text{bool spmf}) \Rightarrow 'grp \text{ddh.adversary})$

where $P1\text{-DDH-mal-adv-}\sigma\text{-false } M \ z \ \mathcal{A} \ D \ h \ a \ t = \text{do} \{$
 $\text{let } (\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A};$
 $\alpha0 \leftarrow \text{sample-uniform (order } \mathcal{G});$
 $\text{let } h0 = \mathbf{g} [\wedge] \alpha0;$
 $\text{let } h1 = h;$
 $\text{let } b0 = a [\wedge] \alpha0;$
 $\text{let } b1 = t;$

```

((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';
}

```


$\text{return-spmf } ((), \text{out})\}$

definition $P2\text{-ideal-model-end} :: ('grp \times 'grp) \Rightarrow 'grp \Rightarrow (('grp \times 'grp \times 'grp \times 'grp \times 'grp) \times 'state)$
 $\Rightarrow ('grp, 'adv\text{-out}2, 'state) \text{adv-3-P2} \Rightarrow (\text{unit} \times 'adv\text{-out}2) \text{spmf}$

where $P2\text{-ideal-model-end } M \text{ l bs } \mathcal{A}3 = \text{do } \{$
 $\text{let } (x0, x1) = M;$
 $\text{let } ((h0, h1, a, b0, b1), s) = \text{bs};$
 $\text{let } \sigma' = (\text{if } l = \mathbf{1} \text{ then False else True});$
 $(- :: \text{unit}, x\sigma) \leftarrow \text{funct-OT-12 } (x0, x1) \sigma';$
 $u0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $v0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $u1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $v1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $\text{let } w0 = a \text{ [}\uparrow\text{]} u0 \otimes \mathbf{g} \text{ [}\uparrow\text{]} v0;$
 $\text{let } w1 = a \text{ [}\uparrow\text{]} u1 \otimes \mathbf{g} \text{ [}\uparrow\text{]} v1;$
 $\text{let } z0 = b0 \text{ [}\uparrow\text{]} u0 \otimes h0 \text{ [}\uparrow\text{]} v0 \otimes (\text{if } \sigma' \text{ then } \mathbf{1} \text{ else } x\sigma);$
 $\text{let } z1 = (b1 \otimes \text{inv } \mathbf{g}) \text{ [}\uparrow\text{]} u1 \otimes h1 \text{ [}\uparrow\text{]} v1 \otimes (\text{if } \sigma' \text{ then } x\sigma \text{ else } \mathbf{1});$
 $\text{let } e0 = (w0, z0);$
 $\text{let } e1 = (w1, z1);$
 $\text{out} \leftarrow \mathcal{A}3 \text{ e0 e1 s};$
 $\text{return-spmf } ((), \text{out})\}$

definition $P2\text{-ideal-model}' :: ('grp \times 'grp) \Rightarrow \text{bool} \Rightarrow 'aux \Rightarrow ('aux, 'grp, 'adv\text{-out}2, 'state) \text{adv-mal-P2} \Rightarrow (\text{unit} \times 'adv\text{-out}2) \text{spmf}$

where $P2\text{-ideal-model}' M \sigma z \mathcal{A} = \text{do } \{$
 $\text{let } (x0, x1) = M;$
 $\text{let } (\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A};$
 $((h0, h1, a, b0, b1), s) \leftarrow \mathcal{A}1 \sigma z;$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (h0 \in \text{carrier } \mathcal{G} \wedge h1 \in \text{carrier } \mathcal{G} \wedge a \in \text{carrier } \mathcal{G} \wedge b0 \in \text{carrier } \mathcal{G} \wedge b1 \in \text{carrier } \mathcal{G});$
 $((\text{in}1, \text{in}2, \text{in}3 :: 'grp), r), s') \leftarrow \mathcal{A}2 (h0, h1, a, b0, b1) s;$
 $\text{let } (h, a, b) = (h0 \otimes \text{inv } h1, a, b0 \otimes \text{inv } b1);$
 $(\text{out-zk-funct}, -) \leftarrow \text{funct-DH-ZK } (h, a, b) ((\text{in}1, \text{in}2, \text{in}3), r);$
 $- :: \text{unit} \leftarrow \text{assert-spmf } \text{out-zk-funct};$
 $\text{let } l = b0 \otimes (\text{inv } (h0 \text{ [}\uparrow\text{]} r));$
 $P2\text{-ideal-model-end } (x0, x1) l ((h0, h1, a, b0, b1), s') \mathcal{A}3\}$

lemma $P2\text{-ideal-model-rewrite: } P2\text{-ideal-model } M \sigma z \mathcal{A} = P2\text{-ideal-model}' M \sigma z \mathcal{A}$

by($\text{simp add: } P2\text{-ideal-model}'\text{-def } P2\text{-ideal-model}\text{-def } P2\text{-ideal-model-end}\text{-def } \text{Let}\text{-def } \text{split}\text{-def}$)

definition $P2\text{-real-model-end} :: ('grp \times 'grp) \Rightarrow (('grp \times 'grp \times 'grp \times 'grp \times 'grp) \times 'state)$
 $\Rightarrow ('grp, 'adv\text{-out}2, 'state) \text{adv-3-P2} \Rightarrow (\text{unit} \times 'adv\text{-out}2) \text{spmf}$

where $P2\text{-real-model-end } M \text{ bs } \mathcal{A}3 = \text{do } \{$

```

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
by(simp add: *P2-real-model'-def P2-real-model-def P2-real-model-end-def split-def*)

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

end

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

lemma *P1-assert-correct1*:
shows ((g [∧] (α0::nat)) [∧] (r::nat) ⊗ g ⊗ inv ((g [∧] (α1::nat)) [∧] r ⊗ g)
 = (g [∧] α0 ⊗ inv (g [∧] α1)) [∧] r)
 (is ?lhs = ?rhs)
proof –
have *in-carrier1*: (g [∧] α1) [∧] r ∈ carrier G **by** *simp*

have *in-carrier2*: $\text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r) \in \text{carrier } \mathcal{G}$ **by** *simp*
have *1*: $?lhs = (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes \mathbf{g} \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r) \otimes \text{inv } \mathbf{g}$
using *cyclic-group-assoc nat-pow-pow inverse-split in-carrier1* **by** *simp*
also have *2*: $\dots = (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes (\mathbf{g} \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r)) \otimes \text{inv } \mathbf{g}$
using *cyclic-group-assoc in-carrier2* **by** *simp*
also have $\dots = (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes (\text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r) \otimes \mathbf{g}) \otimes \text{inv } \mathbf{g}$
using *in-carrier2 cyclic-group-commute* **by** *simp*
also have *3*: $\dots = (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r) \otimes (\mathbf{g} \otimes \text{inv } \mathbf{g})$
using *cyclic-group-assoc in-carrier2* **by** *simp*
also have $\dots = (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r)$ **by** *simp*
also have $\dots = (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1)) [\uparrow] r$
using *inverse-pow-pow* **by** *simp*
ultimately show *?thesis*
by (*simp add: cyclic-group-commute pow-mult-distrib*)
qed

lemma *P1-assert-correct2*:
shows $(\mathbf{g} [\uparrow] (\alpha 0::\text{nat})) [\uparrow] (r::\text{nat}) \otimes \text{inv } ((\mathbf{g} [\uparrow] (\alpha 1::\text{nat})) [\uparrow] r) = (\mathbf{g} [\uparrow] \alpha 0$
 $\otimes \text{inv } (\mathbf{g} [\uparrow] \alpha 1)) [\uparrow] r$
(is *?lhs = ?rhs***)**
proof–
have *in-carrier2*: $\mathbf{g} [\uparrow] \alpha 1 \in \text{carrier } \mathcal{G}$ **by** *simp*
hence *?lhs* = $(\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1)) [\uparrow] r$
using *inverse-pow-pow* **by** *simp*
thus *?thesis*
by (*simp add: cyclic-group-commute monoid-comm-monoidI pow-mult-distrib*)
qed

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

lemma *P1-real-ddh0-σ-false*:
assumes $\sigma = \text{False}$
shows $((P1\text{-real-model } M \sigma z \mathcal{A}) \gg (\lambda \text{ view. } D \text{ view})) = (\text{ddh.DDH0 } (P1\text{-DDH-mal-adv-}\sigma\text{-false } M z \mathcal{A} D))$
including *monad-normalisation*
proof–
have
 $(in2 = \mathbf{g} [\uparrow] (r::\text{nat}) \wedge in3 = in1 [\uparrow] r \wedge in1 = \mathbf{g} [\uparrow] (\alpha 0::\text{nat}) \otimes \text{inv } (\mathbf{g} [\uparrow] (\alpha 1::\text{nat})))$
 $\wedge in2 = \mathbf{g} [\uparrow] r \wedge in3 = (\mathbf{g} [\uparrow] r) [\uparrow] \alpha 0 \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r))$
 $\longleftrightarrow (in1 = \mathbf{g} [\uparrow] \alpha 0 \otimes \text{inv } (\mathbf{g} [\uparrow] \alpha 1) \wedge in2 = \mathbf{g} [\uparrow] r \wedge in3$
 $= (\mathbf{g} [\uparrow] r) [\uparrow] \alpha 0 \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r))$
for *in1 in2 in3 r α0 α1*
by (*auto simp add: P1-assert-correct2 power-swap*)
moreover have $((P1\text{-real-model } M \sigma z \mathcal{A}) \gg (\lambda \text{ view. } D \text{ view})) = \text{do } \{$
 $\text{let } (\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A};$
 $r \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$
 $\alpha 0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

```

α1 ← sample-uniform (order G);
let h0 = g [∧] α0;
let h1 = g [∧] α1;
let a = g [∧] r;
let b0 = (g [∧] r) [∧] α0;
let b1 = h1 [∧] r;
((in1, in2, in3), s) ← A1 M h0 h1 a b0 b1 z;
let (h, a, b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
(b :: bool, - :: unit) ← funct-DH-ZK (in1, in2, in3) ((h, a, b), r);
- :: unit ← assert-spmf (b);
(((w0, z0), (w1, z1)), s') ← A2 h0 h1 a b0 b1 M s;
adv-out ← A3 s';
D (adv-out, ((z0 ⊗ (inv w0 [∧] α0))))}
by(simp add: P1-real-model-def assms split-def Let-def power-swap)
ultimately show ?thesis
by(simp add: P1-real-model-def ddh.DDH0-def Let-def)
qed

lemma P1-ideal-ddh1-σ-false:
  assumes σ = False
  shows ((P1-ideal-model M σ z A) ≫= (λ view. D view)) = (ddh.DDH1 (P1-DDH-mal-adv-σ-false
M z A D))
  including monad-normalisation
proof-
  have ((P1-ideal-model M σ z A) ≫= (λ view. D view)) = do {
    let (A1, A2, A3) = A;
    r ← sample-uniform (order G);
    α0 ← sample-uniform (order G);
    α1 ← sample-uniform (order G);
    let h0 = g [∧] α0;
    let h1 = g [∧] α1;
    let a = g [∧] r;
    let b0 = (g [∧] r) [∧] α0;
    let b1 = h1 [∧] r ⊗ g;
    ((in1, in2, in3), s) ← A1 M h0 h1 a b0 b1 z;
    let (h, a, b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
    - :: unit ← assert-spmf ((in1, in2, in3) = (h, a, b));
    (((w0, z0), (w1, z1)), s') ← A2 h0 h1 a b0 b1 M s;
    let x0 = (z0 ⊗ (inv w0 [∧] α0));
    let x1 = (z1 ⊗ (inv w1 [∧] α1));
    (-, f-out2) ← funct-OT-12 (x0, x1) σ;
    adv-out ← A3 s';
    D (adv-out, f-out2)}
  by(simp add: P1-ideal-model-def assms split-def Let-def power-swap)
  thus ?thesis
  by(auto simp add: P1-ideal-model-def ddh.DDH1-def funct-OT-12-def Let-def
  assms )
qed

```

lemma *P1-real-ddh1-σ-true:*

assumes $\sigma = \text{True}$

shows $((P1\text{-real-model } M \ \sigma \ z \ \mathcal{A}) \gg (\lambda \text{ view. } D \text{ view})) = (\text{ddh.DDH1 } (P1\text{-DDH-mal-adv-}\sigma\text{-true } M \ z \ \mathcal{A} \ D))$

including *monad-normalisation*

proof –

have $(in2 = \mathbf{g} [\uparrow] (r::\text{nat}) \wedge in3 = in1 [\uparrow] r \wedge in1 = \mathbf{g} [\uparrow] (\alpha0::\text{nat}) \otimes \text{inv } (\mathbf{g} [\uparrow] (\alpha1::\text{nat})))$

$\wedge in2 = \mathbf{g} [\uparrow] r \wedge in3 = (\mathbf{g} [\uparrow] r) [\uparrow] \alpha0 \otimes \mathbf{g} \otimes \text{inv } ((\mathbf{g} [\uparrow] \alpha1) [\uparrow] r \otimes \mathbf{g}))$

$\longleftrightarrow (in1 = \mathbf{g} [\uparrow] \alpha0 \otimes \text{inv } (\mathbf{g} [\uparrow] \alpha1) \wedge in2 = \mathbf{g} [\uparrow] r$

$\wedge in3 = (\mathbf{g} [\uparrow] \alpha0) [\uparrow] r \otimes \mathbf{g} \otimes \text{inv } ((\mathbf{g} [\uparrow] r) [\uparrow] \alpha1 \otimes \mathbf{g}))$

for $in1 \ in2 \ in3 \ r \ \alpha0 \ \alpha1$

by (*auto simp add: P1-assert-correct1 power-swap*)

moreover have $((P1\text{-real-model } M \ \sigma \ z \ \mathcal{A}) \gg (\lambda \text{ view. } D \text{ view})) = \text{do } \{$

$\text{let } (\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A};$

$r \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

$\alpha0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

$\alpha1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

$\text{let } h0 = \mathbf{g} [\uparrow] \alpha0;$

$\text{let } h1 = \mathbf{g} [\uparrow] \alpha1;$

$\text{let } a = \mathbf{g} [\uparrow] r;$

$\text{let } b0 = ((\mathbf{g} [\uparrow] r) [\uparrow] \alpha0) \otimes \mathbf{g};$

$\text{let } b1 = (h1 [\uparrow] r) \otimes \mathbf{g};$

$((in1, in2, in3), s) \leftarrow \mathcal{A}1 \ M \ h0 \ h1 \ a \ b0 \ b1 \ z;$

$\text{let } (h, a, b) = (h0 \otimes \text{inv } h1, a, b0 \otimes \text{inv } b1);$

$(b :: \text{bool}, - :: \text{unit}) \leftarrow \text{funct-DH-ZK } (in1, in2, in3) ((h, a, b), r);$

$- :: \text{unit} \leftarrow \text{assert-spmf } (b);$

$((w0, z0), (w1, z1), s') \leftarrow \mathcal{A}2 \ h0 \ h1 \ a \ b0 \ b1 \ M \ s;$

$\text{adv-out} \leftarrow \mathcal{A}3 \ s';$

$D (\text{adv-out}, ((z1 \otimes (\text{inv } w1 [\uparrow] \alpha1))))\}$

by (*simp add: P1-real-model-def assms split-def Let-def power-swap*)

ultimately show *?thesis*

by (*simp add: Let-def P1-real-model-def ddh.DDH1-def assms power-swap*)

qed

lemma *P1-ideal-ddh0-σ-true:*

assumes $\sigma = \text{True}$

shows $((P1\text{-ideal-model } M \ \sigma \ z \ \mathcal{A}) \gg (\lambda \text{ view. } D \text{ view})) = (\text{ddh.DDH0 } (P1\text{-DDH-mal-adv-}\sigma\text{-true } M \ z \ \mathcal{A} \ D))$

including *monad-normalisation*

proof –

have $((P1\text{-ideal-model } M \ \sigma \ z \ \mathcal{A}) \gg (\lambda \text{ view. } D \text{ view})) = \text{do } \{$

$\text{let } (\mathcal{A}1, \mathcal{A}2, \mathcal{A}3) = \mathcal{A};$

$r \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

$\alpha0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

$\alpha1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$

$\text{let } h0 = \mathbf{g} [\uparrow] \alpha0;$

$\text{let } h1 = \mathbf{g} [\uparrow] \alpha1;$

$\text{let } a = \mathbf{g} [\uparrow] r;$

```

let b0 = (g [⌈] r) [⌋] α0;
let b1 = h1 [⌈] r ⊗ g;
((in1, in2, in3), s) ← A1 M h0 h1 a b0 b1 z;
let (h, a, b) = (h0 ⊗ inv h1, a, b0 ⊗ inv b1);
- :: unit ← assert-spmf ((in1, in2, in3) = (h, a, b));
((w0, z0), (w1, z1), s') ← A2 h0 h1 a b0 b1 M s;
let x0 = (z0 ⊗ (inv w0 [⌈] α0));
let x1 = (z1 ⊗ (inv w1 [⌈] α1));
(-, f-out2) ← funct-OT-12 (x0, x1) σ;
adv-out ← A3 s';
D (adv-out, f-out2)}
by(simp add: P1-ideal-model-def assms Let-def split-def power-swap)
thus ?thesis
by(simp add: split-def Let-def P1-ideal-model-def ddh.DDH0-def assms funct-OT-12-def
power-swap)
qed

```

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

```

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

```

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

```

assumes σ = False
shows mal-def.adv-P1 M σ z (P1-S1, P1-S2) A D
  ≤ ddh.advantage (P1-DDH-mal-adv-σ-false M z A D)
  + ddh.advantage (ddh.DDH-A' (P1-DDH-mal-adv-σ-false M z A D))
using P1-real-ideal-DDH-advantage-false ddh.DDH-advantage-bound assms by
metis

```

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

```

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

```

lemma *P1-real-ideal-DDH-advantage-true-bound:*

```

assumes σ = True
shows mal-def.adv-P1 M σ z (P1-S1, P1-S2) A D
  ≤ ddh.advantage (P1-DDH-mal-adv-σ-true M z A D)
  + ddh.advantage (ddh.DDH-A' (P1-DDH-mal-adv-σ-true M z A D))
using P1-real-ideal-DDH-advantage-true ddh.DDH-advantage-bound assms by
metis

```

lemma *P2-output-rewrite*:

assumes $s < \text{order } \mathcal{G}$

shows $(\mathbf{g} [\wedge] (r * u1 + v1), \mathbf{g} [\wedge] (r * \alpha * u1 + v1 * \alpha) \otimes \text{inv } \mathbf{g} [\wedge] u1)$
 $= (\mathbf{g} [\wedge] (r * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G}),$

$\mathbf{g} [\wedge] (r * \alpha * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $\otimes \text{inv } \mathbf{g} [\wedge] ((s + u1) \text{ mod } \text{order } \mathcal{G} + (\text{order } \mathcal{G} - s)))$

proof–

have $\mathbf{g} [\wedge] (r * u1 + v1) = \mathbf{g} [\wedge] (r * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G})$

proof–

have $[(r * u1 + v1) = (r * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G})] \text{ (mod } (\text{order } \mathcal{G}))$

proof–

have $[(r * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G}) = r * (s + u1) + (r * \text{order } \mathcal{G} - r * s + v1)] \text{ (mod } (\text{order } \mathcal{G}))$

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

hence $[(r * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G}) = r * s + r * u1 + r * \text{order } \mathcal{G} - r * s + v1] \text{ (mod } (\text{order } \mathcal{G}))$

by (*metis* (*no-types*, *lifting*) *Nat.add-diff-assoc add.assoc assms distrib-left less-or-eq-imp-le mult-le-mono*)

hence $[(r * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G}) = r * u1 + r * \text{order } \mathcal{G} + v1] \text{ (mod } (\text{order } \mathcal{G}))$ **by** *simp*

thus *?thesis*

by (*simp add: cong-def semiring-normalization-rules(23)*)

qed

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

qed

moreover have $\mathbf{g} [\wedge] (r * \alpha * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G} * \alpha)$

$\otimes \text{inv } \mathbf{g} [\wedge] ((s + u1) \text{ mod } \text{order } \mathcal{G} + (\text{order } \mathcal{G} - s))$
 $= \mathbf{g} [\wedge] (r * \alpha * u1 + v1 * \alpha) \otimes \text{inv } \mathbf{g} [\wedge] u1$

proof–

have $\mathbf{g} [\wedge] (r * \alpha * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G} * \alpha) = \mathbf{g} [\wedge] (r * \alpha * u1 + v1 * \alpha)$

proof–

have $[(r * \alpha * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G} * \alpha) = r * \alpha * u1 + v1 * \alpha] \text{ (mod } (\text{order } \mathcal{G}))$

proof–

have $[(r * \alpha * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G} * \alpha)$

$= r * \alpha * (s + u1) + (r * \text{order } \mathcal{G} - r * s + v1) * \alpha] \text{ (mod } (\text{order } \mathcal{G}))$

using *cong-def mod-add-cong mod-mult-left-eq mod-mult-right-eq by blast*

hence $[(r * \alpha * ((s + u1) \text{ mod } \text{order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod } \text{order } \mathcal{G} * \alpha)$

$$= r * \alpha * s + r * \alpha * u1 + (r * \text{order } \mathcal{G} - r * s + v1) * \alpha] \text{ (mod (order } \mathcal{G}))$$
by (*simp add: distrib-left*)
hence $[(r * \alpha * ((s + u1) \text{ mod order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod order } \mathcal{G} * \alpha)$

$$= r * \alpha * s + r * \alpha * u1 + r * \text{order } \mathcal{G} * \alpha - r * s * \alpha + v1 * \alpha]$$
(mod (order } \mathcal{G})) **using** *distrib-right assms*
by (*smt Groups.mult-ac(3) order-gt-0 Nat.add-diff-assoc2 add.commute diff-mult-distrib2 mult.commute mult-strict-mono order.strict-implies-order semiring-normalization-rules(25) zero-order(1)*)
hence $[(r * \alpha * ((s + u1) \text{ mod order } \mathcal{G}) + (r * \text{order } \mathcal{G} - r * s + v1) \text{ mod order } \mathcal{G} * \alpha)$

$$= r * \alpha * u1 + r * \text{order } \mathcal{G} * \alpha + v1 * \alpha] \text{ (mod (order } \mathcal{G}))$$
 by *simp*
thus *?thesis*
by (*simp add: cong-def semiring-normalization-rules(16) semiring-normalization-rules(23)*)
qed
thus *?thesis using finite-group pow-generator-eq-iff-cong by blast*
qed
also have $\text{inv } \mathbf{g} [\wedge] ((s + u1) \text{ mod order } \mathcal{G} + (\text{order } \mathcal{G} - s)) = \text{inv } \mathbf{g} [\wedge] u1$
proof–
have $[(s + u1) \text{ mod order } \mathcal{G} + (\text{order } \mathcal{G} - s) = u1] \text{ (mod (order } \mathcal{G}))$
proof–
have $[(s + u1) \text{ mod order } \mathcal{G} + (\text{order } \mathcal{G} - s) = s + u1 + (\text{order } \mathcal{G} - s)]$
(mod (order } \mathcal{G}))
by (*simp add: add.commute mod-add-right-eq cong-def*)
hence $[(s + u1) \text{ mod order } \mathcal{G} + (\text{order } \mathcal{G} - s) = u1 + \text{order } \mathcal{G}] \text{ (mod (order } \mathcal{G}))$
using *assms by simp*
thus *?thesis by (simp add: cong-def)*
qed
hence $\mathbf{g} [\wedge] ((s + u1) \text{ mod order } \mathcal{G} + (\text{order } \mathcal{G} - s)) = \mathbf{g} [\wedge] u1$
using *finite-group pow-generator-eq-iff-cong by blast*
thus *?thesis*
by (*metis generator-closed inverse-pow-pow*)
qed
ultimately show *?thesis by argo*
qed
ultimately show *?thesis by simp*
qed

lemma *P2-inv-g-rewrite:*

assumes $s < \text{order } \mathcal{G}$
shows $(\text{inv } \mathbf{g} [\wedge] (u1' + (\text{order } \mathcal{G} - s))) = \mathbf{g} [\wedge] s \otimes \text{inv } (\mathbf{g} [\wedge] u1')$
proof–
have *power-commute-rewrite:* $\mathbf{g} [\wedge] (((\text{order } \mathcal{G} - s) + u1') \text{ mod order } \mathcal{G}) = \mathbf{g} [\wedge] (u1' + (\text{order } \mathcal{G} - s))$
using *add.commute pow-generator-mod by metis*
have $(\text{order } \mathcal{G} - s + u1') \text{ mod order } \mathcal{G} = (\text{int } (\text{order } \mathcal{G}) - \text{int } s + \text{int } u1') \text{ mod order } \mathcal{G}$

by (metis of-nat-add of-nat-diff order.strict-implies-order zmod-int assms(1))
 also have ... = $(- \text{int } s + \text{int } u1') \text{ mod order } \mathcal{G}$
 by (metis (full-types) add commute minus-mod-self1 mod-add-right-eq)
 ultimately have $(\text{order } \mathcal{G} - s + u1') \text{ mod order } \mathcal{G} = (- \text{int } s \text{ mod } (\text{order } \mathcal{G}) + \text{int } u1' \text{ mod } (\text{order } \mathcal{G})) \text{ mod order } \mathcal{G}$
 by presburger
 hence $\mathbf{g} [\uparrow] (((\text{order } \mathcal{G} - s) + u1') \text{ mod order } \mathcal{G})$
 $= \mathbf{g} [\uparrow] ((- \text{int } s \text{ mod } (\text{order } \mathcal{G}) + \text{int } u1' \text{ mod } (\text{order } \mathcal{G})) \text{ mod order } \mathcal{G})$
 by (metis int-pow-int)
 hence $\mathbf{g} [\uparrow] (u1' + (\text{order } \mathcal{G} - s))$
 $= \mathbf{g} [\uparrow] ((- \text{int } s \text{ mod } (\text{order } \mathcal{G}) + \text{int } u1' \text{ mod } (\text{order } \mathcal{G})) \text{ mod order } \mathcal{G})$
 using power-commute-rewrite by argo
 also have ...
 $= \mathbf{g} [\uparrow] (- \text{int } s \text{ mod } (\text{order } \mathcal{G}) + \text{int } u1' \text{ mod } (\text{order } \mathcal{G}))$
 using pow-generator-mod-int by blast
 also have ... = $\mathbf{g} [\uparrow] (- \text{int } s \text{ mod } (\text{order } \mathcal{G})) \otimes \mathbf{g} [\uparrow] (\text{int } u1' \text{ mod } (\text{order } \mathcal{G}))$
 by (simp add: int-pow-mult)
 also have ... = $\mathbf{g} [\uparrow] (- \text{int } s) \otimes \mathbf{g} [\uparrow] (\text{int } u1')$
 using pow-generator-mod-int by simp
 ultimately have $\text{inv } (\mathbf{g} [\uparrow] (u1' + (\text{order } \mathcal{G} - s))) = \text{inv } (\mathbf{g} [\uparrow] (- \text{int } s) \otimes \mathbf{g} [\uparrow] (\text{int } u1'))$ by simp
 hence $\text{inv } (\mathbf{g} [\uparrow] ((u1' + (\text{order } \mathcal{G} - s)) \text{ mod } (\text{order } \mathcal{G}))) = \text{inv } (\mathbf{g} [\uparrow] (- \text{int } s) \otimes \text{inv } (\mathbf{g} [\uparrow] (\text{int } u1')))$
 using pow-generator-mod
 by (simp add: inverse-split)
 also have ... = $\mathbf{g} [\uparrow] (\text{int } s) \otimes \text{inv } (\mathbf{g} [\uparrow] (\text{int } u1'))$
 by (simp add: int-pow-neg)
 also have ... = $\mathbf{g} [\uparrow] s \otimes \text{inv } (\mathbf{g} [\uparrow] u1')$
 by (simp add: int-pow-int)
 ultimately show ?thesis
 by (simp add: inverse-pow-pow pow-generator-mod)
 qed

lemma P2-inv-g-s-rewrite:

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

lemma *P2-e0-rewrite:*

assumes $s < \text{order } \mathcal{G}$

shows $(\mathbf{g} [\wedge] (r * x + xa), \mathbf{g} [\wedge] (r * \alpha * x + xa * \alpha) \otimes \mathbf{g} [\wedge] x) =$
 $(\mathbf{g} [\wedge] (r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G}),$

$\mathbf{g} [\wedge] (r * \alpha * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G} * \alpha)$
 $\otimes \mathbf{g} [\wedge] ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G} + s))$

proof –

have $\mathbf{g} [\wedge] (r * x + xa) = \mathbf{g} [\wedge] (r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G})$

proof –

have $[(r * x + xa) = (r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G})] (\bmod \text{order } \mathcal{G})$

proof –

have $[(r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G}) = (r * ((\text{order } \mathcal{G} - s + x) + (r * s + xa))] (\bmod \text{order } \mathcal{G})$

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

hence $[(r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G}) = r * (\text{order } \mathcal{G} - s) + r * x + r * s + xa] (\bmod \text{order } \mathcal{G})$

by (*simp add: add.assoc distrib-left*)

hence $[(r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G}) = r * x + r * s + r * (\text{order } \mathcal{G} - s) + xa] (\bmod \text{order } \mathcal{G})$

by (*metis add.assoc add.commute*)

hence $[(r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G}) = r * x + r * s + r * \text{order } \mathcal{G} - r * s + xa] (\bmod \text{order } \mathcal{G})$

proof –

have $[(xa + r * s) \bmod \text{order } \mathcal{G} + r * ((x + (\text{order } \mathcal{G} - s)) \bmod \text{order } \mathcal{G}) = xa + r * (s + x + (\text{order } \mathcal{G} - s))] (\bmod \text{order } \mathcal{G})$

by (*metis (no-types) <[r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G} = r * x + r * s + r * (\text{order } \mathcal{G} - s) + xa] (\bmod \text{order } \mathcal{G})>*, *add.commute distrib-left*)

then show *?thesis*

by (*simp add: assms add.commute distrib-left order.strict-implies-order*)

qed

hence $[(r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G}) = r * x + xa] (\bmod \text{order } \mathcal{G})$

proof –

have $[(xa + r * s) \bmod \text{order } \mathcal{G} + r * ((x + (\text{order } \mathcal{G} - s)) \bmod \text{order } \mathcal{G}) = xa + (r * x + r * \text{order } \mathcal{G})] (\bmod \text{order } \mathcal{G})$

by (*metis (no-types) <[r * ((\text{order } \mathcal{G} - s + x) \bmod \text{order } \mathcal{G}) + (r * s + xa) \bmod \text{order } \mathcal{G} = r * x + r * s + r * \text{order } \mathcal{G} - r * s + xa] (\bmod \text{order } \mathcal{G})>*, *add.commute add.left-commute add-diff-cancel-left'*)

then show *?thesis*

by (*metis (no-types) add.commute cong-add-lcancel-nat cong-def distrib-left mod-add-self2 mod-mult-right-eq*)

qed

then show *?thesis using cong-def by metis*

qed
then show *?thesis using finite-group pow-generator-eq-iff-cong by blast*
qed
moreover have $\mathbf{g} [\uparrow] (r * \alpha * x + xa * \alpha) \otimes \mathbf{g} [\uparrow] x =$
 $\mathbf{g} [\uparrow] (r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $\otimes \mathbf{g} [\uparrow] ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G} + s)$
proof-
have $\mathbf{g} [\uparrow] (r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= \mathbf{g} [\uparrow] (r * \alpha * x + xa * \alpha)$
proof-
have $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha) = r * \alpha * x + xa * \alpha] (\text{mod } \text{order } \mathcal{G})$
proof-
have $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= (r * \alpha * ((\text{order } \mathcal{G} - s) + x) + (r * s + xa) * \alpha)] (\text{mod } \text{order } \mathcal{G})$
by *(metis (no-types, lifting) cong-add cong-def mod-mult-left-eq mod-mult-right-eq)*

hence $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= r * \alpha * (\text{order } \mathcal{G} - s) + r * \alpha * x + r * s * \alpha + xa * \alpha] (\text{mod } \text{order } \mathcal{G})$
by *(simp add: add.assoc distrib-left distrib-right)*
hence $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= r * \alpha * x + r * s * \alpha + r * \alpha * (\text{order } \mathcal{G} - s) + xa * \alpha] (\text{mod } \text{order } \mathcal{G})$
by *(simp add: add.commute add.left-commute)*
hence $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= r * \alpha * x + r * s * \alpha + r * \alpha * \text{order } \mathcal{G} - r * \alpha * s + xa * \alpha]$
 $(\text{mod } \text{order } \mathcal{G})$
proof -
have $\forall n \ na. \neg (n::\text{nat}) \leq na \vee n + (na - n) = na$
by *(meson ordered-cancel-comm-monoid-diff-class.add-diff-inverse)*
then have $r * \alpha * s + r * \alpha * (\text{order } \mathcal{G} - s) = r * \alpha * \text{order } \mathcal{G}$
by *(metis add-mult-distrib2 assms less-or-eq-imp-le)*
then have $r * \alpha * x + r * s * \alpha + r * \alpha * \text{order } \mathcal{G} = r * \alpha * s + r * \alpha$
 $* (\text{order } \mathcal{G} - s) + (r * \alpha * x + r * s * \alpha)$
by *presburger*
then have $f1: r * \alpha * x + r * s * \alpha + r * \alpha * \text{order } \mathcal{G} - r * \alpha * s = r$
 $* \alpha * s + r * \alpha * (\text{order } \mathcal{G} - s) - r * \alpha * s + (r * \alpha * x + r * s * \alpha)$
by *simp*
have $r * \alpha * s + r * \alpha * (\text{order } \mathcal{G} - s) = r * \alpha * (\text{order } \mathcal{G} - s) + r * \alpha$
 $* s$
by *presburger*
then have $r * \alpha * x + r * s * \alpha + r * \alpha * \text{order } \mathcal{G} - r * \alpha * s = r * \alpha$

$* x + r * s * \alpha + r * \alpha * (\text{order } \mathcal{G} - s)$
using *f1 diff-add-inverse2* **by** *presburger*
then show *?thesis*
using $\langle [r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha = r * \alpha * x + r * s * \alpha + r * \alpha * (\text{order } \mathcal{G} - s) + xa * \alpha] (\text{mod } \text{order } \mathcal{G}) \rangle$ **by** *presburger*
qed
hence $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= r * \alpha * x + r * \alpha * s + r * \alpha * \text{order } \mathcal{G} - r * \alpha * s + xa * \alpha]$
 $(\text{mod } \text{order } \mathcal{G})$
using *add.commute add.assoc* **by** *force*
hence $[(r * \alpha * ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G}) + (r * s + xa) \text{ mod } \text{order } \mathcal{G} * \alpha)$
 $= r * \alpha * x + r * \alpha * \text{order } \mathcal{G} + xa * \alpha] (\text{mod } \text{order } \mathcal{G})$ **by** *simp*
thus *?thesis using cong-def semiring-normalization-rules(23)*
by $(\text{simp add: } \langle \wedge c b a. [b = c] (\text{mod } a) = (b \text{ mod } a = c \text{ mod } a) \rangle \langle \wedge c b a. a + b + c = a + c + b \rangle)$
qed
thus *?thesis using finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
also have $\mathbf{g} [\wedge] ((\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G} + s) = \mathbf{g} [\wedge] x$
proof-
have $[(\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G} + s = x] (\text{mod } \text{order } \mathcal{G})$
proof-
have $[(\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G} + s = (\text{order } \mathcal{G} - s + x + s)] (\text{mod } \text{order } \mathcal{G})$
by $(\text{simp add: } \text{add.commute } \text{cong-def } \text{mod-add-right-eq})$
hence $[(\text{order } \mathcal{G} - s + x) \text{ mod } \text{order } \mathcal{G} + s = (\text{order } \mathcal{G} + x)] (\text{mod } \text{order } \mathcal{G})$
qed
using *assms* **by** *auto*
thus *?thesis*
by $(\text{simp add: } \text{cong-def})$
qed
thus *?thesis using finite-group pow-generator-eq-iff-cong* **by** *blast*
qed
ultimately show *?thesis* **by** *argo*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *P2-case-l-new-1-gt-e0-rewrite:*

assumes $s < \text{order } \mathcal{G}$
shows $(\mathbf{g} [\wedge] (r * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})) + x) \text{ mod } \text{order } \mathcal{G})$
 $+ (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})) + xa) \text{ mod } \text{order } \mathcal{G}),$
 $\mathbf{g} [\wedge] (r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})) + x) \text{ mod } \text{order } \mathcal{G}))$

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

$$\mathbf{g} [\wedge] (t * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t \text{ (order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G})$$

$$+ s * (\text{nat } ((\text{fst } (\text{bezw } t \text{ (order } \mathcal{G}))) \text{ mod order } \mathcal{G})))) = (\mathbf{g} [\wedge] (r * x + xa), \mathbf{g} [\wedge] (r * \alpha * x + xa * \alpha) \otimes \mathbf{g} [\wedge] (t * x))$$

proof –

have $\mathbf{g} [\wedge] ((r::\text{nat}) * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t \text{ (order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G})$

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

$$= \mathbf{g} [\wedge] (r * x + xa)$$

proof (cases $r = 0$)

case *True*
then show *?thesis*
by (*simp add: pow-generator-mod*)
next
case *False*
have $[(r::\text{nat}) * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t \text{ (order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G})$

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

proof –

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

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

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

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

proof –

have $\text{order } \mathcal{G} \neq 0$
using *order-gt-0* **by** *simp*
then show *?thesis*
using *cong-add cong-def mod-mult-right-eq*
by (*smt mod-mod-trivial*)
qed
hence $[r * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t \text{ (order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G})$

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

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

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

proof –

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

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

```

order  $\mathcal{G}$ )) + r * x] (mod order  $\mathcal{G}$ )
  by (simp add: cong-def distrib-left mod-mult-right-eq)
  then show ?thesis
    using assms cong-add gr-implies-not0 by fastforce
  qed
  hence [r * ((order  $\mathcal{G}$  * order  $\mathcal{G}$  - s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order
 $\mathcal{G}$ )) + x) mod order  $\mathcal{G}$ )
    + (r * s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ )) + xa) mod
order  $\mathcal{G}$ 
      = r * order  $\mathcal{G}$  * order  $\mathcal{G}$  - r * s * (nat ((fst (bezw t (order
 $\mathcal{G}$ ))) mod order  $\mathcal{G}$ )) + r * x
      + r * s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ )) +
xa] (mod order  $\mathcal{G}$ )
    by (simp add: ab-semigroup-mult-class.mult-ac(1) right-diff-distrib' add.assoc)
  hence [r * ((order  $\mathcal{G}$  * order  $\mathcal{G}$  - s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order
 $\mathcal{G}$ )) + x) mod order  $\mathcal{G}$ )
    + (r * s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ )) + xa) mod
order  $\mathcal{G}$ 
      = r * order  $\mathcal{G}$  * order  $\mathcal{G}$  + r * x + xa] (mod order  $\mathcal{G}$ )
  proof-
  have r * order  $\mathcal{G}$  * order  $\mathcal{G}$  - r * s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod
order  $\mathcal{G}$ )) > 0
  proof-
  have order  $\mathcal{G}$  * order  $\mathcal{G}$  > s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order
 $\mathcal{G}$ ))
  proof-
  have (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ )) ≤ order  $\mathcal{G}$ 
  proof -
  have ∀ x0 x1. ((x0::int) mod x1 < x1) = (¬ x1 + - 1 * (x0 mod x1)
≤ 0)
    by linarith
  then have ¬ int (order  $\mathcal{G}$ ) + - 1 * (fst (bezw t (order  $\mathcal{G}$ ))) mod int
(order  $\mathcal{G}$ ) ≤ 0
    using of-nat-0-less-iff order-gt-0 by fastforce
  then show ?thesis
    by linarith
  qed
  thus ?thesis using assms
  proof -
  have ∀ n na. ¬ n ≤ na ∨ ¬ na * order  $\mathcal{G}$  < n * nat (fst (bezw t (order
 $\mathcal{G}$ ))) mod int (order  $\mathcal{G}$ ))
    by (meson <nat (fst (bezw (t::nat) (order  $\mathcal{G}$ ))) mod int (order  $\mathcal{G}$ )) ≤
order  $\mathcal{G}$  mult-le-mono not-le)
  then show ?thesis
    by (metis (no-types, opaque-lifting) <(s::nat) < order  $\mathcal{G}$ > mult-less-cancel2
nat-less-le not-le not-less-zero)
  qed
  qed
  thus ?thesis using False

```

by auto
qed
thus ?thesis
proof –
have $r * \text{order } \mathcal{G} * \text{order } \mathcal{G} + r * x + xa = r * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G}))) + (r * s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) + xa) + r * x$
using $\langle (0::\text{nat}) < (r::\text{nat}) * \text{order } \mathcal{G} * \text{order } \mathcal{G} - r * (s::\text{nat}) * \text{nat } (\text{fst } (\text{bezw } (t::\text{nat}) (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) \rangle$ *diff-mult-distrib2* **by force**
then show ?thesis
by (*metis (no-types)* $\langle [(r::\text{nat}) * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - (s::\text{nat}) * \text{nat } (\text{fst } (\text{bezw } (t::\text{nat}) (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) + (x::\text{nat})) \text{ mod order } \mathcal{G}) + (r * s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) + (xa::\text{nat})) \text{ mod order } \mathcal{G} = r * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G}))) + r * x + (r * s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) + xa)] \text{ mod order } \mathcal{G} \rangle$) *semiring-normalization-rules(23)*
qed
qed
hence $[r * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G}) + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G}$
 $= r * x + xa] \text{ mod order } \mathcal{G}$
by (*metis (no-types, lifting) mod-mult-self4 add.assoc mult.commute cong-def*)
thus ?thesis by blast
qed
then show ?thesis using finite-group pow-generator-eq-iff-cong by blast
qed
moreover have $\mathbf{g} [\hat{\cdot}] (r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G}) + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha) \otimes$
 $\mathbf{g} [\hat{\cdot}] (t * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G}) + s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})))$
 $= \mathbf{g} [\hat{\cdot}] (r * \alpha * x + xa * \alpha) \otimes \mathbf{g} [\hat{\cdot}] (t * x)$
proof–
have $\mathbf{g} [\hat{\cdot}] (r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G}) + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha)$
 $= \mathbf{g} [\hat{\cdot}] (r * \alpha * x + xa * \alpha)$
proof–
have $[r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G}) + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha$
 $= r * \alpha * x + xa * \alpha] \text{ mod order } \mathcal{G}$
proof–
have $[r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + x) \text{ mod order } \mathcal{G}) + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha$

$\mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha$
 $= r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G}))) + x) + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) +$
 $xa) * \alpha] (\text{mod order } \mathcal{G})$
proof –
show *?thesis*
by (*meson cong-def mod-add-cong mod-mult-left-eq mod-mult-right-eq*)
qed
hence *mod-eq*: $[r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha$
 $= r * \alpha * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G}))) + r * \alpha * x + r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))$
 $* \alpha + xa * \alpha] (\text{mod order } \mathcal{G})$
by (*simp add: distrib-left distrib-right add.assoc*)
hence *mod-eq'*: $[r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha$
 $= r * \alpha * \text{order } \mathcal{G} * \text{order } \mathcal{G} - r * \alpha * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod order } \mathcal{G}))) + r * \alpha * x + r * \alpha * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod$
 $\text{order } \mathcal{G})) + xa * \alpha] (\text{mod order } \mathcal{G})$
by (*simp add: semiring-normalization-rules(16) diff-mult-distrib2 semiring-normalization-rules(18)*)
hence $[r * \alpha * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod$
 $\text{order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + (r * s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))$
 $\text{mod order } \mathcal{G})) + xa) \text{ mod order } \mathcal{G} * \alpha$
 $= r * \alpha * \text{order } \mathcal{G} * \text{order } \mathcal{G} + r * \alpha * x + xa * \alpha] (\text{mod order } \mathcal{G})$
proof(*cases r * \alpha = 0*)
case *True*
then show *?thesis*
by (*metis mod-eq' diff-zero mult-0 plus-nat.add-0*)
next
case *False*
hence *bound*: $r * \alpha * \text{order } \mathcal{G} * \text{order } \mathcal{G} - r * \alpha * s * (\text{nat } ((\text{fst } (\text{bezw } t$
 $(\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) > 0$
proof–
have $s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) < \text{order } \mathcal{G} * \text{order}$
 \mathcal{G}
using *assms*
by (*simp add: mult-strict-mono nat-less-iff*)
thus *?thesis*
using *False by auto*
qed
thus *?thesis*
proof –
have $r * \alpha * \text{order } \mathcal{G} * \text{order } \mathcal{G} = r * \alpha * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * \text{nat}$
 $(\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G}))$
 $+ r * s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$
 $\text{mod int } (\text{order } \mathcal{G})) * \alpha$


```

      using bound diff-mult-distrib2 by force
      then have  $r * \alpha * \text{order } \mathcal{G} * \text{order } \mathcal{G} + r * \alpha * x = r * \alpha * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G})) + r * \alpha * x + r * s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G}) * \alpha$ 
      by presburger
      then show ?thesis
      using mod-eq by presburger
    qed
  qed
  thus ?thesis
  by (metis (mono-tags, lifting) add.assoc cong-def mod-mult-self3)
  qed
  then show ?thesis using finite-group pow-generator-eq-iff-cong by blast
  qed
  also have  $\mathbf{g} \ [\uparrow] (t * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})))$ 
    =  $\mathbf{g} \ [\uparrow] (t * x)$ 
  proof-
    have  $[t * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) = t * x] (\text{mod order } \mathcal{G})$ 
    proof-
      have  $[t * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) = (t * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) + x + s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})))] (\text{mod order } \mathcal{G})$ 
      using cong-def mod-add-left-eq mod-mult-cong by blast
      hence  $[t * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) + x) \text{ mod order } \mathcal{G} + s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G}))) = t * (\text{order } \mathcal{G} * \text{order } \mathcal{G} + x)] (\text{mod order } \mathcal{G})$ 
    proof-
      have  $\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) > 0$ 
      proof-
        have  $(\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod order } \mathcal{G})) \leq \text{order } \mathcal{G}$ 
        using nat-le-iff order.strict-implies-order order-gt-0
        by (simp add: order.strict-implies-order)
        thus ?thesis
        by (metis assms diff-mult-distrib le0 linorder-neqE-nat mult-strict-mono not-le zero-less-diff)
      qed
    thus ?thesis
    using  $\langle [(t::\text{nat}) * ((\text{order } \mathcal{G} * \text{order } \mathcal{G} - (s::\text{nat}) * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G}))) + (x::\text{nat})) \text{ mod order } \mathcal{G} + s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G})) = t * (\text{order } \mathcal{G} * \text{order } \mathcal{G} - s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G})) + x + s * \text{nat } (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G})) \rangle$  by auto

```

```

    qed
    thus ?thesis
    by (metis (no-types, opaque-lifting) add commute cong-def mod-mult-right-eq
mod-mult-self1)
    qed
    thus ?thesis using finite-group pow-generator-eq-iff-cong by blast
    qed
    ultimately show ?thesis by argo
    qed
    ultimately show ?thesis by simp
    qed

```

lemma *P2-case-l-neq-1-gt-x0-rewrite*:

```

    assumes  $t < \text{order } \mathcal{G}$ 
    and  $t \neq 0$ 
    shows  $\mathbf{g} \lceil \lceil (t * u0 + (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})))) = \mathbf{g} \lceil \lceil (t * u0) \otimes \mathbf{g} \lceil \lceil s$ 
    proof -
    from assms have gcd:  $\text{gcd } t (\text{order } \mathcal{G}) = 1$ 
    using prime-field coprime-imp-gcd-eq-1 by blast
    hence inverse-t:  $[s * (t * (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))) = s * 1] (\text{mod } \text{order } \mathcal{G})$ 
    by (metis Num.of-nat-simps(2) Num.of-nat-simps(5) cong-scalar-left order-gt-0 inverse)
    hence inverse-t':  $[t * u0 + s * (t * (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))) = t * u0 + s * 1] (\text{mod } \text{order } \mathcal{G})$ 
    using cong-add-lcancel by fastforce
    have eq:  $\text{int } (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})) = (\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G}$ 
    proof -
    have  $(\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G} \geq 0$  using order-gt-0 by simp
    hence  $(\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})) = (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))$ 
    mod order  $\mathcal{G}$  by linarith
    thus ?thesis by blast
    qed
    have  $[(t * (u0 + (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})))) = t * u0 + s] (\text{mod } \text{order } \mathcal{G})$ 
    proof -
    have  $[t * (u0 + (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})))) = t * u0 + t * (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G}))) (\text{mod } \text{order } \mathcal{G})$ 
    by (simp add: distrib-left)
    hence  $[t * (u0 + (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})))) = t * u0 + s * (t * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G}))) (\text{mod } \text{order } \mathcal{G})$ 
    by (simp add: ab-semigroup-mult-class.mult-ac(1) mult.left-commute)
    hence  $[t * (u0 + (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})))) = t * u0 + s * (t * ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})) (\text{mod } \text{order } \mathcal{G})$ 
    using eq
    by (simp add: distrib-left mult.commute semiring-normalization-rules(18))
    hence  $[t * (u0 + (s * (\text{nat } ((\text{fst } (\text{bezw } t (\text{order } \mathcal{G}))) \text{ mod } \text{order } \mathcal{G})))) = t * u0 + s * (t * (\text{fst } (\text{bezw } t (\text{order } \mathcal{G})))) (\text{mod } \text{order } \mathcal{G})$ 

```

by (*metis (no-types, opaque-lifting) cong-def mod-add-right-eq mod-mult-right-eq*)
 hence $[t * (u0 + (s * (nat ((fst (bezw t (order \mathcal{G}))) \text{ mod } order \mathcal{G})))) = t * u0$
 $+ s * 1] \text{ (mod } order \mathcal{G})$ using *inverse-t'*
 using *cong-trans cong-int-iff* by *blast*
 thus *?thesis* by *simp*
 qed
 hence $\mathbf{g} [\checkmark] (t * (u0 + (s * (nat ((fst (bezw t (order \mathcal{G}))) \text{ mod } order \mathcal{G})))) = \mathbf{g}$
 $[\checkmark] (t * u0 + s)$ using *finite-group pow-generator-eq-iff-cong* by *blast*
 thus *?thesis*
 by (*simp add: nat-pow-mult*)
 qed

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

lemma *P2-ideal-real-end-eq*:

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

 obtain $s :: nat$ where $s: \mathbf{g} [\checkmark] s = x1$ and *s-lt*: $s < order\ \mathcal{G}$
 by (*metis assms(3) mod-less-divisor generatorE order-gt-0 pow-generator-mod*)
 have $b1 \otimes inv\ \mathbf{g} = \mathbf{g} [\checkmark] (r * \alpha) \otimes inv\ \mathbf{g}$
 by (*metis \alpha b1-h1 generator-closed mult commute nat-pow-pow*)

```

have a-g-exp-rewrite: (g [⌈] (r::nat)) [⌈] u0 ⊗ g [⌈] v0 = g [⌈] (r * u0 + v0)
for u0 v0
by (simp add: nat-pow-mult nat-pow-pow)
have z1-rewrite: (b1 ⊗ inv g) [⌈] u1 ⊗ h1 [⌈] v1 ⊗ 1 = g [⌈] (r * α * u1 + v1
* α) ⊗ inv g [⌈] u1
for u1 v1 :: nat
by (smt α b1-h1 pow-mult-distrib cyclic-group-commute generator-closed inv-closed
m-assoc m-closed monoid-comm-monoidI mult.commute nat-pow-closed nat-pow-mult
nat-pow-pow r-one)
have z1-rewrite': g [⌈] (r * α * u1 + v1 * α) ⊗ g [⌈] s ⊗ inv g [⌈] u1 = (b1
⊗ inv g) [⌈] u1 ⊗ h1 [⌈] v1 ⊗ x1
for u1 v1
using assert-in-carrier cyclic-group-commute m-assoc s z1-rewrite by auto
have P2-ideal-model-end (x0,x1) (b0 ⊗ (inv (h0 [⌈] r))) ((h0,h1, g [⌈] (r::nat),b0,b1),s')
A3 = do {
  u0 ← sample-uniform (order G);
  v0 ← sample-uniform (order G);
  u1 ← sample-uniform (order G);
  v1 ← sample-uniform (order G);
  let w0 = (g [⌈] (r::nat)) [⌈] u0 ⊗ g [⌈] v0;
  let w1 = (g [⌈] (r::nat)) [⌈] u1 ⊗ g [⌈] v1;
  let z0 = b0 [⌈] u0 ⊗ h0 [⌈] v0 ⊗ x0;
  let z1 = (b1 ⊗ inv g) [⌈] u1 ⊗ h1 [⌈] v1 ⊗ 1;
  let e0 = (w0,z0);
  let e1 = (w1,z1);
  out ← A3 e0 e1 s';
  return-spmf ((), out)}
by(simp add: P2-ideal-model-end-def True funct-OT-12-def)
also have ... = do {
  u0 ← sample-uniform (order G);
  v0 ← sample-uniform (order G);
  u1 ← sample-uniform (order G);
  v1 ← sample-uniform (order G);
  let w0 = (g [⌈] (r::nat)) [⌈] u0 ⊗ g [⌈] v0;
  let w1 = (g [⌈] (r::nat)) [⌈] u1 ⊗ g [⌈] v1;
  let z0 = b0 [⌈] u0 ⊗ h0 [⌈] v0 ⊗ x0;
  let z1 = g [⌈] (r * α * u1 + v1 * α) ⊗ inv g [⌈] u1;
  let e0 = (w0,z0);
  let e1 = (w1,z1);
  out ← A3 e0 e1 s';
  return-spmf ((), out)}
by(simp add: z1-rewrite)
also have ... = do {
  u0 ← sample-uniform (order G);
  v0 ← sample-uniform (order G);
  u1 ← sample-uniform (order G);
  v1 ← sample-uniform (order G);
  let w0 = (g [⌈] (r::nat)) [⌈] u0 ⊗ g [⌈] v0;
  let w1 = g [⌈] (r * u1 + v1);

```

```

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

```

```

    return-spmf ((), out)}
  by(simp add: P2-inv-g-s-rewrite assms s-lt cong: bind-spmf-cong-simp)
also have ... = do {
  u0 ← sample-uniform (order  $\mathcal{G}$ );
  v0 ← sample-uniform (order  $\mathcal{G}$ );
  u1 ← sample-uniform (order  $\mathcal{G}$ );
  v1 ← sample-uniform (order  $\mathcal{G}$ );
  let w0 = (g [^] (r::nat)) [^] u0 ⊗ g [^] v0;
  let w1 = (g [^] (r::nat)) [^] u1 ⊗ g [^] v1;
  let z0 = b0 [^] u0 ⊗ h0 [^] v0 ⊗ x0;
  let z1 = (b1 ⊗ inv g) [^] u1 ⊗ h1 [^] v1 ⊗ x1;
  let e0 = (w0, z0);
  let e1 = (w1, z1);
  out ←  $\mathcal{A}3$  e0 e1 s';
  return-spmf ((), out)}
  by(simp add: a-g-exp-rewrite z1-rewrite^)
ultimately show ?thesis
  by(simp add: P2-real-model-end-def)
next
  obtain  $\alpha :: \text{nat}$  where  $\alpha: \mathbf{g} [^] \alpha = h0$ 
    using generatorE assms
    using assert-in-carrier by auto
  have w0-rewrite:  $\mathbf{g} [^] (r * u0 + v0) = (\mathbf{g} [^] (r::\text{nat})) [^] u0 \otimes \mathbf{g} [^] v0$ 
    for  $u0\ v0$ 
    by (simp add: nat-pow-mult nat-pow-pow)
  have order-gt-0: order  $\mathcal{G} > 0$  using order-gt-0 by simp
  obtain  $s :: \text{nat}$  where  $s: \mathbf{g} [^] s = x0$  and  $s\text{-lt}: s < \text{order } \mathcal{G}$ 
    by (metis mod-less-divisor generatorE order-gt-0 pow-generator-mod x0-in-carrier)
  case False — case 2
  hence l-neq-1:  $(b0 \otimes (\text{inv } (h0 [^] r))) \neq \mathbf{1}$  by auto
  then show ?thesis
  proof (cases  $(b0 \otimes (\text{inv } (h0 [^] r))) = \mathbf{g}$ )
    case True
      hence  $b0 = \mathbf{g} \otimes h0 [^] r$ 
        by (metis assert-in-carrier generator-closed inv-solve-right nat-pow-closed)
      hence  $b0 = \mathbf{g} \otimes \mathbf{g} [^] (r * \alpha)$ 
        by (metis  $\alpha$  generator-closed mult.commute nat-pow-pow)
      have z0-rewrite:  $b0 [^] u0 \otimes h0 [^] v0 \otimes \mathbf{1} = \mathbf{g} [^] (r * \alpha * u0 + v0 * \alpha) \otimes$ 
 $\mathbf{g} [^] u0$ 
        for  $u0\ v0 :: \text{nat}$ 
        by (smt  $\alpha \langle b0 = \mathbf{g} \otimes \mathbf{g} [^] (r * \alpha) \rangle$  pow-mult-distrib cyclic-group-commute generator-closed m-assoc monoid-comm-monoidI mult.commute nat-pow-closed nat-pow-mult nat-pow-pow r-one)
      have z0-rewrite':  $\mathbf{g} [^] (r * \alpha * u0 + v0 * \alpha) \otimes \mathbf{g} [^] (u0 + s) = \mathbf{g} [^] (r * \alpha * u0 + v0 * \alpha) \otimes \mathbf{g} [^] u0 \otimes \mathbf{g} [^] s$ 
        for  $u0\ v0$ 
        by (simp add: add.assoc nat-pow-mult)
      have z0-rewrite'':  $\mathbf{g} [^] (r * \alpha * u0 + v0 * \alpha) \otimes \mathbf{g} [^] u0 \otimes x0 = b0 [^] u0$ 
 $\otimes h0 [^] v0 \otimes x0$ 

```

```

    for u0 v0 using z0-rewrite
    using assert-in-carrier by auto
  have P2-ideal-model-end (x0,x1) (b0 ⊗ (inv (h0 [⌈ r])) ((h0,h1,g [⌈ (r::nat),b0,b1),s')
  A3 = do {
    u0 ← sample-uniform (order G);
    v0 ← sample-uniform (order G);
    u1 ← sample-uniform (order G);
    v1 ← sample-uniform (order G);
    let w0 = (g [⌈ (r::nat)) [⌈ u0 ⊗ g [⌈ v0;
    let w1 = (g [⌈ (r::nat)) [⌈ u1 ⊗ g [⌈ v1;
    let z0 = b0 [⌈ u0 ⊗ h0 [⌈ v0 ⊗ 1;
    let z1 = (b1 ⊗ inv g) [⌈ u1 ⊗ h1 [⌈ v1 ⊗ x1;
    let e0 = (w0,z0);
    let e1 = (w1,z1);
    out ← A3 e0 e1 s';
    return-spmf ((), out)}
    apply(simp add: P2-ideal-model-end-def True funct-OT-12-def)
    using order-gt-0 order-gt-1-gen-not-1 True l-neq-1 by auto
  also have ... = do {
    u0 ← sample-uniform (order G);
    v0 ← sample-uniform (order G);
    u1 ← sample-uniform (order G);
    v1 ← sample-uniform (order G);
    let w0 = g [⌈ (r * u0 + v0);
    let w1 = (g [⌈ (r::nat)) [⌈ u1 ⊗ g [⌈ v1;
    let z0 = g [⌈ (r * α * u0 + v0 * α) ⊗ g [⌈ u0;
    let z1 = (b1 ⊗ inv g) [⌈ u1 ⊗ h1 [⌈ v1 ⊗ x1;
    let e0 = (w0,z0);
    let e1 = (w1,z1);
    out ← A3 e0 e1 s';
    return-spmf ((), out)}
    by(simp add: z0-rewrite w0-rewrite)
  also have ... = do {
    u0 ← map-spmf (λ u0. ((order G - s) + u0) mod (order G)) (sample-uniform
    (order G));
    v0 ← map-spmf (λ v0. (r * s + v0) mod (order G)) (sample-uniform (order
    G));
    u1 ← sample-uniform (order G);
    v1 ← sample-uniform (order G);
    let w0 = g [⌈ (r * u0 + v0);
    let w1 = (g [⌈ (r::nat)) [⌈ u1 ⊗ g [⌈ v1;
    let z0 = g [⌈ (r * α * u0 + v0 * α) ⊗ g [⌈ (u0 + s);
    let z1 = (b1 ⊗ inv g) [⌈ u1 ⊗ h1 [⌈ v1 ⊗ x1;
    let e0 = (w0,z0);
    let e1 = (w1,z1);
    out ← A3 e0 e1 s';
    return-spmf ((), out)}
    apply(simp add: bind-map-spmf o-def Let-def cong: bind-spmf-cong-simp)
    using P2-e0-rewrite assms s-lt assms by presburger

```

```

also have ... = do {
  u0 ← map-spmf (λ u0. ((order  $\mathcal{G}$  - s) + u0) mod (order  $\mathcal{G}$ )) (sample-uniform
(order  $\mathcal{G}$ ));
  v0 ← map-spmf (λ v0. (r * s + v0) mod (order  $\mathcal{G}$ )) (sample-uniform (order
 $\mathcal{G}$ ));
  u1 ← sample-uniform (order  $\mathcal{G}$ );
  v1 ← sample-uniform (order  $\mathcal{G}$ );
  let w0 = g [↑] (r * u0 + v0);
  let w1 = (g [↑] (r::nat)) [↑] u1 ⊗ g [↑] v1;
  let z0 = g [↑] (r * α * u0 + v0 * α) ⊗ g [↑] u0 ⊗ x0;
  let z1 = (b1 ⊗ inv g) [↑] u1 ⊗ h1 [↑] v1 ⊗ x1;
  let e0 = (w0, z0);
  let e1 = (w1, z1);
  out ←  $\mathcal{A}3$  e0 e1 s';
  return-spmf ((), out)}
by(simp add: z0-rewrite' s)
also have ... = do {
  u0 ← map-spmf (λ u0. ((order  $\mathcal{G}$  - s) + u0) mod (order  $\mathcal{G}$ )) (sample-uniform
(order  $\mathcal{G}$ ));
  v0 ← map-spmf (λ v0. (r * s + v0) mod (order  $\mathcal{G}$ )) (sample-uniform (order
 $\mathcal{G}$ ));
  u1 ← sample-uniform (order  $\mathcal{G}$ );
  v1 ← sample-uniform (order  $\mathcal{G}$ );
  let w0 = (g [↑] (r::nat)) [↑] u0 ⊗ g [↑] v0;
  let w1 = (g [↑] (r::nat)) [↑] u1 ⊗ g [↑] v1;
  let z0 = b0 [↑] u0 ⊗ h0 [↑] v0 ⊗ x0;
  let z1 = (b1 ⊗ inv g) [↑] u1 ⊗ h1 [↑] v1 ⊗ x1;
  let e0 = (w0, z0);
  let e1 = (w1, z1);
  out ←  $\mathcal{A}3$  e0 e1 s';
  return-spmf ((), out)}
by(simp add: w0-rewrite z0-rewrite'')
also have ... = do {
  u0 ← sample-uniform (order  $\mathcal{G}$ );
  v0 ← sample-uniform (order  $\mathcal{G}$ );
  u1 ← sample-uniform (order  $\mathcal{G}$ );
  v1 ← sample-uniform (order  $\mathcal{G}$ );
  let w0 = (g [↑] (r::nat)) [↑] u0 ⊗ g [↑] v0;
  let w1 = (g [↑] (r::nat)) [↑] u1 ⊗ g [↑] v1;
  let z0 = b0 [↑] u0 ⊗ h0 [↑] v0 ⊗ x0;
  let z1 = (b1 ⊗ inv g) [↑] u1 ⊗ h1 [↑] v1 ⊗ x1;
  let e0 = (w0, z0);
  let e1 = (w1, z1);
  out ←  $\mathcal{A}3$  e0 e1 s';
  return-spmf ((), out)}
by(simp add: samp-uni-plus-one-time-pad)
ultimately show ?thesis
by(simp add: P2-real-model-end-def)
next

```



```

case False — case 3
have b0-l:  $b0 = (b0 \otimes (\text{inv } (h0 \ [\ ] r))) \otimes h0 \ [\ ] r$ 
  by (simp add: assert-in-carrier m-assoc)
have b0-g-r:  $b0 = (b0 \otimes (\text{inv } (h0 \ [\ ] r))) \otimes \mathbf{g} \ [\ ] (r * \alpha)$ 
  by (metis  $\alpha$  b0-l generator-closed mult.commute nat-pow-pow)
obtain  $t :: \text{nat}$  where  $t: \mathbf{g} \ [\ ] t = (b0 \otimes (\text{inv } (h0 \ [\ ] r)))$  and t-lt-order-g:  $t < \text{order } \mathcal{G}$ 
  by (metis (full-types) mod-less-divisor order-gt-0 pow-generator-mod
    assert-in-carrier cyclic-group.generatorE cyclic-group-axioms
    inv-closed m-closed nat-pow-closed)
with l-neq-1 have t-neq-0:  $t \neq 0$  using l-neq-1-exp-neq-0 by simp
have z0-rewrite:  $b0 \ [\ ] u0 \otimes h0 \ [\ ] v0 \otimes \mathbf{1} = \mathbf{g} \ [\ ] (r * \alpha * u0 + v0 * \alpha) \otimes$ 
 $((b0 \otimes (\text{inv } (h0 \ [\ ] r)))) \ [\ ] u0$ 
  for  $u0 \ v0$ 
proof—
  from b0-l have  $b0 \ [\ ] u0 \otimes h0 \ [\ ] v0 = ((b0 \otimes (\text{inv } (h0 \ [\ ] r))) \otimes h0 \ [\ ]$ 
 $r) \ [\ ] u0 \otimes h0 \ [\ ] v0$  by simp
  hence  $b0 \ [\ ] u0 \otimes h0 \ [\ ] v0 = ((b0 \otimes (\text{inv } (h0 \ [\ ] r)))) \ [\ ] u0 \otimes (h0 \ [\ ] r)$ 
 $[\ ] u0 \otimes h0 \ [\ ] v0$ 
  by (simp add: assert-in-carrier pow-mult-distrib cyclic-group-commute
monoid-comm-monoidI)
  hence  $b0 \ [\ ] u0 \otimes h0 \ [\ ] v0 = ((\mathbf{g} \ [\ ] \alpha) \ [\ ] r) \ [\ ] u0 \otimes (\mathbf{g} \ [\ ] \alpha) \ [\ ] v0 \otimes$ 
 $((b0 \otimes (\text{inv } (h0 \ [\ ] r)))) \ [\ ] u0$ 
  using cyclic-group-assoc cyclic-group-commute assert-in-carrier  $\alpha$  by simp
  hence  $b0 \ [\ ] u0 \otimes h0 \ [\ ] v0 = \mathbf{g} \ [\ ] (r * \alpha * u0 + v0 * \alpha) \otimes ((b0 \otimes (\text{inv } (h0 \ [\ ] r)))) \ [\ ] u0$ 
  by (simp add: monoid.nat-pow-pow mult.commute nat-pow-mult)
  thus ?thesis
  by (simp add: assert-in-carrier)
qed
have z0-rewrite':  $\mathbf{g} \ [\ ] (r * \alpha * u0 + v0 * \alpha) \otimes ((b0 \otimes (\text{inv } (h0 \ [\ ] r)))) \ [\ ]$ 
 $u0 = \mathbf{g} \ [\ ] (r * \alpha * u0 + v0 * \alpha) \otimes \mathbf{g} \ [\ ] (t * u0)$ 
  for  $u0 \ v0$ 
  by (metis generator-closed nat-pow-pow t)
have z0-rewrite'':  $\mathbf{g} \ [\ ] (r * \alpha * u0 + v0 * \alpha) \otimes \mathbf{g} \ [\ ] (t * u0) \otimes \mathbf{g} \ [\ ] s = b0$ 
 $[\ ] u0 \otimes h0 \ [\ ] v0 \otimes x0$ 
  for  $u0 \ v0$ 
  using assert-in-carrier s z0-rewrite z0-rewrite' by auto
have P2-ideal-model-end ( $x0, x1$ ) ( $b0 \otimes (\text{inv } (h0 \ [\ ] r))$ ) ( $(h0, h1, \mathbf{g} \ [\ ] (r :: \text{nat}), b0, b1), s'$ )
A3 = do {
   $u0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$ 
   $v0 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$ 
   $u1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$ 
   $v1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$ 
   $\text{let } w0 = \mathbf{g} \ [\ ] (r * u0 + v0);$ 
   $\text{let } w1 = (\mathbf{g} \ [\ ] (r :: \text{nat})) \ [\ ] u1 \otimes \mathbf{g} \ [\ ] v1;$ 
   $\text{let } z0 = \mathbf{g} \ [\ ] (r * \alpha * u0 + v0 * \alpha) \otimes ((b0 \otimes (\text{inv } (h0 \ [\ ] r)))) \ [\ ] u0;$ 
   $\text{let } z1 = (b1 \otimes \text{inv } \mathbf{g}) \ [\ ] u1 \otimes h1 \ [\ ] v1 \otimes x1;$ 
   $\text{let } e0 = (w0, z0);$ 

```

```

let e1 = (w1,z1);
out ←  $\mathcal{A}3$  e0 e1 s';
return-spmf ((), out)}
  by(simp add: P2-ideal-model-end-def l-neq-1 funct-OT-12-def w0-rewrite
z0-rewrite)
  also have ... = do {
u0 ← sample-uniform (order  $\mathcal{G}$ );
v0 ← sample-uniform (order  $\mathcal{G}$ );
u1 ← sample-uniform (order  $\mathcal{G}$ );
v1 ← sample-uniform (order  $\mathcal{G}$ );
let w0 =  $\mathbf{g}$  [✓] (r * u0 + v0);
let w1 = ( $\mathbf{g}$  [✓] (r::nat)) [✓] u1  $\otimes$   $\mathbf{g}$  [✓] v1;
let z0 =  $\mathbf{g}$  [✓] (r *  $\alpha$  * u0 + v0 *  $\alpha$ )  $\otimes$   $\mathbf{g}$  [✓] (t * u0);
let z1 = (b1  $\otimes$  inv  $\mathbf{g}$ ) [✓] u1  $\otimes$  h1 [✓] v1  $\otimes$  x1;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ←  $\mathcal{A}3$  e0 e1 s';
return-spmf ((), out)}
  by(simp add: z0-rewrite')
  also have ... = do {
u0 ← map-spmf ( $\lambda$  u0. (order  $\mathcal{G}$  * order  $\mathcal{G}$  - (s * ((nat (((fst (bezw t (order
 $\mathcal{G}$ )))) mod (order  $\mathcal{G}$ )))))) + u0) mod (order  $\mathcal{G}$ )) (sample-uniform (order  $\mathcal{G}$ ));
v0 ← map-spmf ( $\lambda$  v0. (r * s * (nat ((fst (bezw t (order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ ))
+ v0) mod (order  $\mathcal{G}$ )) (sample-uniform (order  $\mathcal{G}$ ));
u1 ← sample-uniform (order  $\mathcal{G}$ );
v1 ← sample-uniform (order  $\mathcal{G}$ );
let w0 =  $\mathbf{g}$  [✓] (r * u0 + v0);
let w1 = ( $\mathbf{g}$  [✓] (r::nat)) [✓] u1  $\otimes$   $\mathbf{g}$  [✓] v1;
let z0 =  $\mathbf{g}$  [✓] (r *  $\alpha$  * u0 + v0 *  $\alpha$ )  $\otimes$   $\mathbf{g}$  [✓] (t * (u0 + (s * (nat ((fst (bezw t
(order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ ))))));
let z1 = (b1  $\otimes$  inv  $\mathbf{g}$ ) [✓] u1  $\otimes$  h1 [✓] v1  $\otimes$  x1;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ←  $\mathcal{A}3$  e0 e1 s';
return-spmf ((), out)}
  by(simp add: bind-map-spmf o-def Let-def s-lt P2-case-l-new-1-gt-e0-rewrite
cong: bind-spmf-cong-simp)
  also have ... = do {
u0 ← sample-uniform (order  $\mathcal{G}$ );
v0 ← sample-uniform (order  $\mathcal{G}$ );
u1 ← sample-uniform (order  $\mathcal{G}$ );
v1 ← sample-uniform (order  $\mathcal{G}$ );
let w0 =  $\mathbf{g}$  [✓] (r * u0 + v0);
let w1 = ( $\mathbf{g}$  [✓] (r::nat)) [✓] u1  $\otimes$   $\mathbf{g}$  [✓] v1;
let z0 =  $\mathbf{g}$  [✓] (r *  $\alpha$  * u0 + v0 *  $\alpha$ )  $\otimes$   $\mathbf{g}$  [✓] (t * (u0 + (s * (nat ((fst (bezw t
(order  $\mathcal{G}$ ))) mod order  $\mathcal{G}$ ))))));
let z1 = (b1  $\otimes$  inv  $\mathbf{g}$ ) [✓] u1  $\otimes$  h1 [✓] v1  $\otimes$  x1;
let e0 = (w0,z0);
let e1 = (w1,z1);

```

```

out ←  $\mathcal{A}3$  e0 e1 s';
return-spmf ((), out)}
  by(simp add: samp-uni-plus-one-time-pad)
also have ... = do {
u0 ← sample-uniform (order  $\mathcal{G}$ );
v0 ← sample-uniform (order  $\mathcal{G}$ );
u1 ← sample-uniform (order  $\mathcal{G}$ );
v1 ← sample-uniform (order  $\mathcal{G}$ );
let w0 =  $\mathbf{g}$  [ $\uparrow$ ] (r * u0 + v0);
let w1 = ( $\mathbf{g}$  [ $\uparrow$ ] (r::nat)) [ $\uparrow$ ] u1  $\otimes$   $\mathbf{g}$  [ $\uparrow$ ] v1;
let z0 =  $\mathbf{g}$  [ $\uparrow$ ] (r *  $\alpha$  * u0 + v0 *  $\alpha$ )  $\otimes$   $\mathbf{g}$  [ $\uparrow$ ] (t * u0)  $\otimes$   $\mathbf{g}$  [ $\uparrow$ ] s;
let z1 = (b1  $\otimes$  inv  $\mathbf{g}$ ) [ $\uparrow$ ] u1  $\otimes$  h1 [ $\uparrow$ ] v1  $\otimes$  x1;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ←  $\mathcal{A}3$  e0 e1 s';
return-spmf ((), out)}
  by(simp add: P2-case-l-neq-1-gt-x0-rewrite t-lt-order-g t-neq-0 cyclic-group-assoc)
also have ... = do {
u0 ← sample-uniform (order  $\mathcal{G}$ );
v0 ← sample-uniform (order  $\mathcal{G}$ );
u1 ← sample-uniform (order  $\mathcal{G}$ );
v1 ← sample-uniform (order  $\mathcal{G}$ );
let w0 = ( $\mathbf{g}$  [ $\uparrow$ ] (r::nat)) [ $\uparrow$ ] u0  $\otimes$   $\mathbf{g}$  [ $\uparrow$ ] v0;
let w1 = ( $\mathbf{g}$  [ $\uparrow$ ] (r::nat)) [ $\uparrow$ ] u1  $\otimes$   $\mathbf{g}$  [ $\uparrow$ ] v1;
let z0 = b0 [ $\uparrow$ ] u0  $\otimes$  h0 [ $\uparrow$ ] v0  $\otimes$  x0;
let z1 = (b1  $\otimes$  inv  $\mathbf{g}$ ) [ $\uparrow$ ] u1  $\otimes$  h1 [ $\uparrow$ ] v1  $\otimes$  x1;
let e0 = (w0,z0);
let e1 = (w1,z1);
out ←  $\mathcal{A}3$  e0 e1 s';
return-spmf ((), out)}
  by(simp add: w0-rewrite z0-rewrite'')
ultimately show ?thesis
  by(simp add: P2-real-model-end-def)
qed
qed

```

lemma P2-ideal-real-eq:

assumes x1-in-carrier: $x1 \in \text{carrier } \mathcal{G}$

and x0-in-carrier: $x0 \in \text{carrier } \mathcal{G}$

shows P2-real-model ($x0,x1$) σ z \mathcal{A} = P2-ideal-model ($x0,x1$) σ z \mathcal{A}

proof–

have P2-real-model' ($x0, x1$) σ z \mathcal{A} = P2-ideal-model' ($x0, x1$) σ z \mathcal{A}

proof–

have 1:do {

let ($\mathcal{A}1, \mathcal{A}2, \mathcal{A}3$) = \mathcal{A} ;

((h0,h1,a,b0,b1),s) ← $\mathcal{A}1$ σ z;

- :: unit ← assert-spmf ($h0 \in \text{carrier } \mathcal{G} \wedge h1 \in \text{carrier } \mathcal{G} \wedge a \in \text{carrier } \mathcal{G} \wedge$

$b0 \in \text{carrier } \mathcal{G} \wedge b1 \in \text{carrier } \mathcal{G}$);

((in1, in2, in3), r),s') ← $\mathcal{A}2$ (h0,h1,a,b0,b1) s;

```

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

```

lemma *malicious-sec-P2*:

```

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

```

lemma *correct*:

```

  assumes x0 ∈ carrier G
  and x1 ∈ carrier G
  shows funct-OT-12 (x0, x1) σ = protocol-ot (x0,x1) σ
proof-

```

have σ -eq-0-output-correct:
 $((\mathbf{g} [\uparrow] \alpha 0) [\uparrow] r) [\uparrow] u 0 \otimes (\mathbf{g} [\uparrow] \alpha 0) [\uparrow] v 0 \otimes x 0 \otimes$
 $\text{inv } (((\mathbf{g} [\uparrow] r) [\uparrow] u 0 \otimes \mathbf{g} [\uparrow] v 0) [\uparrow] \alpha 0) = x 0$
(is ?lhs = ?rhs)
for $\alpha 0$ r $u 0$ $v 0$:: nat
proof–
have *mult-com*: $r * u 0 * \alpha 0 = \alpha 0 * r * u 0$ **by** *simp*
have *in-carrier1*: $((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0))) \otimes (\mathbf{g} [\uparrow] (v 0 * \alpha 0)) \in \text{carrier } \mathcal{G}$ **by**
simp
have *in-carrier2*: $\text{inv } (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0))) \otimes (\mathbf{g} [\uparrow] (v 0 * \alpha 0))) \in \text{carrier } \mathcal{G}$
by *simp*
have ?lhs = $((\mathbf{g} [\uparrow] (\alpha 0 * r * u 0))) \otimes (\mathbf{g} [\uparrow] (\alpha 0 * v 0)) \otimes x 0 \otimes$
 $\text{inv } (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0)) \otimes \mathbf{g} [\uparrow] (v 0 * \alpha 0)))$
by (*simp add: nat-pow-pow pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*)
also have ... = $((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0))) \otimes (\mathbf{g} [\uparrow] (v 0 * \alpha 0)) \otimes x 0 \otimes$
 $(\text{inv } (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0)) \otimes \mathbf{g} [\uparrow] (v 0 * \alpha 0))))$
using *mult.commute mult.assoc mult-com*
by (*metis (no-types) mult.commute*)
also have ... = $x 0 \otimes (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0))) \otimes (\mathbf{g} [\uparrow] (v 0 * \alpha 0))) \otimes$
 $(\text{inv } (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0)) \otimes \mathbf{g} [\uparrow] (v 0 * \alpha 0))))$
using *cyclic-group-commute in-carrier1 assms* **by** *simp*
also have ... = $x 0 \otimes (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0))) \otimes (\mathbf{g} [\uparrow] (v 0 * \alpha 0))) \otimes$
 $(\text{inv } (((\mathbf{g} [\uparrow] (r * u 0 * \alpha 0)) \otimes \mathbf{g} [\uparrow] (v 0 * \alpha 0))))$
using *cyclic-group-assoc in-carrier1 in-carrier2 assms* **by** *auto*
ultimately show ?thesis **using** *assms* **by** *simp*
qed
have σ -eq-1-output-correct:
 $((\mathbf{g} [\uparrow] \alpha 1) [\uparrow] r \otimes \mathbf{g} \otimes \text{inv } \mathbf{g}) [\uparrow] u 1 \otimes (\mathbf{g} [\uparrow] \alpha 1) [\uparrow] v 1 \otimes x 1 \otimes$
 $\text{inv } (((\mathbf{g} [\uparrow] r) [\uparrow] u 1 \otimes \mathbf{g} [\uparrow] v 1) [\uparrow] \alpha 1) = x 1$
(is ?lhs = ?rhs)
for $\alpha 1$ r $u 1$ $v 1$:: nat
proof–
have *com1*: $\alpha 1 * r * u 1 = r * u 1 * \alpha 1$ $v 1 * \alpha 1 = \alpha 1 * v 1$ **by** *simp+*
have *in-carrier1*: $(\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes (\mathbf{g} [\uparrow] (v 1 * \alpha 1)) \in \text{carrier } \mathcal{G}$ **by**
simp
have *in-carrier2*: $\text{inv } (((\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes (\mathbf{g} [\uparrow] (v 1 * \alpha 1)))) \in \text{carrier } \mathcal{G}$
by *simp*
have *lhs*: ?lhs = $((\mathbf{g} [\uparrow] (\alpha 1 * r)) \otimes \mathbf{g} \otimes \text{inv } \mathbf{g}) [\uparrow] u 1 \otimes (\mathbf{g} [\uparrow] (\alpha 1 * v 1)) \otimes x 1$
 \otimes
 $\text{inv } (((\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes \mathbf{g} [\uparrow] (v 1 * \alpha 1)))$
by (*simp add: nat-pow-pow pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*)
also have *lhs1*: ... = $(\mathbf{g} [\uparrow] (\alpha 1 * r)) [\uparrow] u 1 \otimes (\mathbf{g} [\uparrow] (\alpha 1 * v 1)) \otimes x 1 \otimes$
 $\text{inv } (((\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes \mathbf{g} [\uparrow] (v 1 * \alpha 1)))$
by (*simp add: cyclic-group-assoc*)
also have *lhs2*: ... = $(\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes (\mathbf{g} [\uparrow] (v 1 * \alpha 1)) \otimes x 1 \otimes$
 $\text{inv } (((\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes \mathbf{g} [\uparrow] (v 1 * \alpha 1)))$
by (*simp add: nat-pow-pow pow-mult-distrib cyclic-group-commute monoid-comm-monoidI*
com1)
also have ... = $((\mathbf{g} [\uparrow] (r * u 1 * \alpha 1)) \otimes (\mathbf{g} [\uparrow] (v 1 * \alpha 1))) \otimes x 1 \otimes$

```

      inv ((g [↑] (r * u1 * α1)) ⊗ g [↑] (v1 * α1))
    using in-carrier1 in-carrier2 assms cyclic-group-assoc by blast
  also have ... = (x1 ⊗ ((g [↑] (r * u1 * α1)) ⊗ (g [↑] (v1 * α1)))) ⊗
    inv ((g [↑] (r * u1 * α1)) ⊗ g [↑] (v1 * α1))
    using in-carrier1 assms cyclic-group-commute by simp
  ultimately show ?thesis
    using cyclic-group-assoc assms in-carrier1 in-carrier1 assms cyclic-group-commute
  lhs1 lhs2 lhs by force
qed
show ?thesis
  unfolding funct-OT-12-def protocol-ot-def Let-def
  by(cases σ; auto simp add: assms σ-eq-1-output-correct σ-eq-0-output-correct
  bind-spmf-const
  lossless-sample-uniform-units order-gt-0 P1-assert-correct1 P1-assert-correct2
  lossless-weight-spmfD)
qed

```

lemma correctness:

```

  assumes x0 ∈ carrier G
    and x1 ∈ carrier G
  shows mal-def.correct (x0,x1) σ
  unfolding mal-def.correct-def
  by(simp add: correct assms)

```

end

locale OT-asymp =

```

  fixes G :: nat ⇒ 'grp cyclic-group
  assumes ot: ∧η. ot (G η)

```

begin

sublocale ot G n **for** n **using** ot **by** simp

lemma correctness-asym:

```

  assumes x0 ∈ carrier (G n)
    and x1 ∈ carrier (G n)
  shows mal-def.correct n (x0,x1) σ
  using assms correctness by simp

```

lemma P1-security-asym:

```

  negligible (λ n. mal-def.adv-P1 n M σ z (P1-S1 n, P1-S2) A D)
  if neg1: negligible (λ n. ddh.advantage n (P1-DDH-mal-adv-σ-true n M z A D))
    and neg2: negligible (λ n. ddh.advantage n (ddh.DDH-A' n (P1-DDH-mal-adv-σ-true
  n M z A D)))
    and neg3: negligible (λ n. ddh.advantage n (P1-DDH-mal-adv-σ-false n M z A
  D))
    and neg4: negligible (λ n. ddh.advantage n (ddh.DDH-A' n (P1-DDH-mal-adv-σ-false
  n M z A D)))
  proof –

```

```

have neg-add1: negligible ( $\lambda n. \text{ddh.}\text{advantage } n \text{ (P1-DDH-mal-adv-}\sigma\text{-true } n \text{ M } z \text{ } \mathcal{A} \text{ D)}$ )
  +  $\text{ddh.}\text{advantage } n \text{ (ddh.DDH-}\mathcal{A}' \text{ } n \text{ (P1-DDH-mal-adv-}\sigma\text{-true } n \text{ M } z \text{ } \mathcal{A} \text{ D))}$ )
and neg-add2: negligible ( $\lambda n. \text{ddh.}\text{advantage } n \text{ (P1-DDH-mal-adv-}\sigma\text{-false } n \text{ M } z \text{ } \mathcal{A} \text{ D)}$ )
  +  $\text{ddh.}\text{advantage } n \text{ (ddh.DDH-}\mathcal{A}' \text{ } n \text{ (P1-DDH-mal-adv-}\sigma\text{-false } n \text{ M } z \text{ } \mathcal{A} \text{ D))}$ )

using neg1 neg2 neg3 neg4 negligible-plus by(blast)+
show ?thesis
proof(cases  $\sigma$ )
  case True
    have bound-mod:  $|\text{mal-def.}\text{adv-P1 } n \text{ M } \sigma \text{ } z \text{ (P1-S1 } n, \text{ P1-S2) } \mathcal{A} \text{ D}|$ 
       $\leq \text{ddh.}\text{advantage } n \text{ (P1-DDH-mal-adv-}\sigma\text{-true } n \text{ M } z \text{ } \mathcal{A} \text{ D)}$ 
      +  $\text{ddh.}\text{advantage } n \text{ (ddh.DDH-}\mathcal{A}' \text{ } n \text{ (P1-DDH-mal-adv-}\sigma\text{-true } n \text{ M } z \text{ } \mathcal{A} \text{ D))}$ 
for  $n$ 
    by (metis (no-types) True abs-idempotent P1-adv-real-ideal-model-def P1-advantages-eq
P1-real-ideal-DDH-advantage-true-bound)
    then show ?thesis
      using P1-real-ideal-DDH-advantage-true-bound that bound-mod that negligible-le neg-add1 by presburger
  next
    case False
      have bound-mod:  $|\text{mal-def.}\text{adv-P1 } n \text{ M } \sigma \text{ } z \text{ (P1-S1 } n, \text{ P1-S2) } \mathcal{A} \text{ D}|$ 
         $\leq \text{ddh.}\text{advantage } n \text{ (P1-DDH-mal-adv-}\sigma\text{-false } n \text{ M } z \text{ } \mathcal{A} \text{ D)}$ 
        +  $\text{ddh.}\text{advantage } n \text{ (ddh.DDH-}\mathcal{A}' \text{ } n \text{ (P1-DDH-mal-adv-}\sigma\text{-false } n \text{ M } z \text{ } \mathcal{A} \text{ D))}$ 
for  $n$ 
      proof –
        have  $|\text{spmf } (P1\text{-real-model } n \text{ M } \sigma \text{ } z \text{ } \mathcal{A} \ggg \text{D}) \text{ True} - \text{spmf } (P1\text{-ideal-model } n \text{ M } \sigma \text{ } z \text{ } \mathcal{A} \ggg \text{D}) \text{ True}|$ 
           $\leq \text{local.}\text{ddh.}\text{advantage } n \text{ (P1-DDH-mal-adv-}\sigma\text{-false } n \text{ M } z \text{ } \mathcal{A} \text{ D)}$ 
          +  $\text{local.}\text{ddh.}\text{advantage } n \text{ (ddh.DDH-}\mathcal{A}' \text{ } n \text{ (P1-DDH-mal-adv-}\sigma\text{-false } n \text{ M } z \text{ } \mathcal{A} \text{ D))}$ 
by (metis (no-types) False P1-adv-real-ideal-model-def P1-advantages-eq
P1-real-ideal-DDH-advantage-false-bound)
        then show ?thesis
          by (simp add: P1-adv-real-ideal-model-def P1-advantages-eq)
      qed
    then show ?thesis using P1-real-ideal-DDH-advantage-false-bound bound-mod
that negligible-le neg-add2 by presburger
  qed
qed

```

lemma P2-security-*asym*:

```

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

```

end

end

References

- [1] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *IACR Cryptology ePrint Archive*, 2017:753, 2017.
- [2] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
- [3] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.
- [4] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.
- [5] Y. Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.
- [6] A. Lochbihler. CryptHOL. *Archive of Formal Proofs*, 2017. <http://isa-afp.org/entries/CryptHOL.shtml>, Formal proof development.
- [7] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457. ACM/SIAM, 2001.
- [8] A. C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.