

Polynomial Commitment Schemes

Tobias Rothmann

July 13, 2026

Abstract

This entry formalizes polynomial commitment schemes (PCSs) and the security proofs of two variants of the Kate–Zaverucha–Goldberg (KZG) construction [8]. We define an abstract PCS interface and games for correctness, polynomial binding, evaluation binding, hiding, and knowledge soundness. We also formalize symmetric pairings, the DL, t -DL, t -SDH, and t -BSDH assumptions, and the Algebraic Group Model (AGM) of Fuchsbauer, Kiltz, and Loss [6] using a constraint-programming-inspired approach. Based on these definitions, we verify correctness, polynomial binding, evaluation binding, and knowledge soundness for the standard and batched KZG constructions, as well as weak evaluation hiding for the standard KZG. The security proofs are carried out in the CryptHOL framework and follow Shoup’s sequence-of-games approach with machine-checked game transitions.

Contents

1	Pairings	3
2	Mathematical Primitives	5
2.0.1	mod_ring operations on pow of Gp	6
2.0.2	mod_ring operations on pow of GT	9
2.0.3	bilinearity operations for mod_ring elements	10
3	The Discrete Logarithm Assumption	13
4	The t-Discrete Logarithm Assumption	14
5	The t-Strong Diffie-Hellman Assumption	16
6	The t-Bilinear Strong Diffie-Hellman Assumption	18
7	Polynomial Commitment Schemes	19

8	Restrictive Computations	23
8.1	Select	24
8.2	Constrain	25
8.3	restrict	26
8.4	Examples	27
9	The Algebraic Group Model	28
9.1	Example for the ML function	32
10	KZG function definitions	32
10.1	Setup	33
11	Correctness of the KZG	35
11.0.1	Helping lemmas for the computation of ψ	35
11.0.2	Helping lemmas about the public parameters PK	43
12	Extensions to CryptHOL	47
12.1	SPMF True	48
12.2	Sample uniform set	49
12.3	Sample uniform list	50
12.4	Sample uniform polynomial	55
13	Polynomial Binding of the KZG	59
13.1	t-DL game	60
13.2	Helping lemmas	61
13.2.1	Literal helping lemmas	65
13.3	KZG poly bind game to strong reduction game - equivalence theorem	66
14	Evaluation Binding of the KZG	70
14.1	t-SDH game	71
14.2	Defining a reduction adversary from evaluation binding to t-SDH	71
14.3	Helping definitions	71
14.3.1	Literal helping lemmas	73
14.4	Proof	75
15	Knowledge Soundness of the KZG	80
15.1	Helping definitions	82
15.2	Helping lemmas	82
16	Hiding of the KZG	98
16.1	Definitions for the weak hiding game	98
16.1.1	DL game	99
16.1.2	Reduction	99

16.2	Hiding proof	101
16.2.1	Helping lemmas	101
17	Batch Opening Definition	120
17.1	Polynomial operations and prerequisites	120
17.2	Function definitions	122
18	Correctness of the batched KZG	123
19	Polynomial Binding of the batched KZG	126
20	Evaluation Binding of the batched KZG	126
20.1	t-BSDH game	126
20.2	Binding for the batched evaluation according to KZG10 . . .	126
20.2.1	Defining a reduction function to t-BSDH	127
20.2.2	Helping lemmas	128
20.3	Literal helping lemma	132
20.4	KZG eval bind game to reduction game - equivalence theorem	133
21	Knowledge Soundness of the batched KZG	138
	theory <i>Pairing</i>	

```

imports CryptHOL.CryptHOL CryptHOL.Cyclic-Group Berlekamp-Zassenhaus.Finite-Field
          Sigma-Commit-Crypto.Cyclic-Group-Ext HOL-Number-Theory.Cong
begin

```

1 Pairings

```

hide-const Polynomial.order

```

We formalize symmetric pairings for cryptography following the textbook “A Graduate Course in Applied Cryptography” by Boneh and Shoup [2].

```

locale pairing =
   $G_p$  : cyclic-group  $G_p$  +  $G_T$  : cyclic-group  $G_T$ 
for  $G_p$  :: ('a, 'b) cyclic-group-scheme (structure)
and  $G_T$  :: ('c, 'd) cyclic-group-scheme (structure)
+
fixes  $p$  :: int
  and  $e$ 
assumes
   $p$ -prime : prime  $p$  and
  CARD- $G_p$ : int (order  $G_p$ ) =  $p$  and
  CARD- $G_T$ : int (order  $G_T$ ) =  $p$  and
   $e$ -symmetric:  $e \in \text{carrier } G_p \rightarrow \text{carrier } G_p \rightarrow \text{carrier } G_T$  and
   $e$ -bilinearity[simp]:  $\forall a b :: \text{int} . \forall P Q. P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow$ 
     $e (P [\cdot]_{G_p} a) (Q [\cdot]_{G_p} b)$ 
  =  $(e P Q) [\cdot]_{G_T} (a * b)$  and

```

e-non-degeneracy[simp]: $\neg(\forall P Q. P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow e P Q = \mathbf{1}_{G_T})$
begin

The pairing function e is linear in the first component

lemma *e-linear-in-fst*:

assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
shows $e (P [\cdot]_{G_p} (a::\text{int})) Q = (e P Q) [\cdot]_{G_T} a$

proof –

have $e (P [\cdot]_{G_p} a) Q = e (P [\cdot]_{G_p} a) (Q [\cdot]_{G_p} (1::\text{int}))$ **using** *assms* **by** *simp*
also have $\dots = (e P Q) [\cdot]_{G_T} (a*(1::\text{int}))$ **using** *assms* *e-bilinearity* **by** *fast*
also have $\dots = (e P Q) [\cdot]_{G_T} a$ **by** *simp*
finally show $e (P [\cdot]_{G_p} a) Q = (e P Q) [\cdot]_{G_T} a$.

qed

The pairing function e is linear in the second component

lemma *e-linear-in-snd*:

assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
shows $e P (Q [\cdot]_{G_p} (a::\text{int})) = (e P Q) [\cdot]_{G_T} a$

proof –

have $e P (Q [\cdot]_{G_p} a) = e (P [\cdot]_{G_p} (1::\text{int})) (Q [\cdot]_{G_p} a)$ **using** *assms* **by** *simp*
also have $\dots = (e P Q) [\cdot]_{G_T} ((1::\text{int})*a)$ **using** *assms* *e-bilinearity* **by** *fast*
also have $\dots = (e P Q) [\cdot]_{G_T} a$ **by** *simp*
finally show $e P (Q [\cdot]_{G_p} a) = (e P Q) [\cdot]_{G_T} a$.

qed

lemma *addition-in-exponents-on-e[simp]*:

assumes $x \in \text{carrier } G_p \wedge y \in \text{carrier } G_p$
shows $(e x y) [\cdot]_{G_T} (a::\text{int}) \otimes_{G_T} (e x y) [\cdot]_{G_T} (b::\text{int}) = (e x y) [\cdot]_{G_T} (a+b)$
by (*metis* $G_T.\text{int-pow-mult}$ *assms* PiE *e-symmetric*)

this follows from non-degeneracy

lemma *e-from-generators-ne-1*: $e \mathbf{g}_{G_p} \mathbf{g}_{G_p} \neq \mathbf{1}_{G_T}$

proof

assume *asm*: $e \mathbf{g}_{G_p} \mathbf{g}_{G_p} = \mathbf{1}_{G_T}$
have $\forall P Q. P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow e P Q = \mathbf{1}_{G_T}$

proof (*intro allI*)

fix $P Q$

show $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow e P Q = \mathbf{1}_{G_T}$

proof

assume $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
then obtain $p q::\text{int}$ **where** $\mathbf{g}_{G_p} [\cdot]_{G_p} p = P \wedge \mathbf{g}_{G_p} [\cdot]_{G_p} q = Q$
by (*metis* $G_p.\text{generatorE}$ *int-pow-int*)
then have $e P Q = e (\mathbf{g}_{G_p} [\cdot]_{G_p} p) (\mathbf{g}_{G_p} [\cdot]_{G_p} q)$
by *blast*
also have $\dots = e \mathbf{g}_{G_p} \mathbf{g}_{G_p} [\cdot]_{G_T} (p*q)$
by *force*
also have $\dots = \mathbf{1}_{G_T} [\cdot]_{G_T} (p*q)$

```

    using asm by argo
    also have ... = 1GT
    by fastforce
    finally show e P Q = 1GT .
qed
qed
then show False using e-non-degeneracy by blast
qed

```

```

lemma e-g-g-in-carrier-GT[simp]: e gGp gGp ∈ carrier GT
  using e-symmetric by fast

```

mod relations on the exponent (typically useful for cryptographic proofs)

```

lemma pow-on-eq-card-GT[simp]: (gGT [∧]GT (x::int) = gGT [∧]GT (y::int)) =
([x = y] (mod p))
  by (metis (no-types, lifting) CARD-GT GT.finite-carrier GT.gen-power-0 GT.generator-closed
GT.int-pow-eq GT.ord-ge-1 GT.ord-le-group-order GT.pow-ord-eq-1 One-nat-def add-diff-inverse-nat
arith-extra-simps(6) cong-iff-dvd-diff diff-is-0-eq' less-eq-Suc-le
zero-order(5))

```

```

lemma pow-on-eq-card-GT-carrier-ext'[simp]:
  ((e gGp gGp)) [∧]GT x = ((e gGp gGp)) [∧]GT y  $\longleftrightarrow$  [x = y] (mod p)
  by (metis CARD-GT GT.finite-carrier GT.int-pow-eq GT.ord-dvd-group-order
GT.ord-eq-1 GT.ord-id GT.pow-ord-eq-ord-iff GT.pow-order-eq-1 cong-iff-dvd-diff
dvd-antisym e-from-generators-ne-1 e-g-g-in-carrier-GT p-prime prime-imp-coprime
prime-nat-int-transfer)

```

end

end

theory *Primitives*

imports *Pairing*

begin

2 Mathematical Primitives

We define the mathematical primitives used throughout our formalization as well as some useful lemmas (mostly identities).

```

locale math-primitives = pairing Gp GT p e
  for Gp :: ('a, 'b) cyclic-group-scheme (structure)
  and GT :: ('c, 'd) cyclic-group-scheme (structure)
  and p::int
  and e::'a ⇒ 'a ⇒ 'c
  +
  fixes type-q :: ('q :: prime-card) itself
  and max-deg::nat
  assumes

```

p-gr-two: $p > 2$ **and**
d-pos: $\text{max-deg} > 0$ **and**
CARD-q: $\text{int } (\text{CARD}('q)) = p$ **and**
d-l-p: $\text{max-deg} < p$
begin

abbreviation *pow-mod-ring* (**infixr** $\hat{\cdot}$ 75)
 where $x \hat{\cdot} y \equiv x [\cdot]_1 (\text{to-int-mod-ring } (y::'q \text{ mod-ring}))$

abbreviation *div-in-grp* (**infixr** $\div_1$ 70)
 where $x \div_1 y \equiv x \otimes_1 \text{inv}_1 y$

2.0.1 mod_ring operations on pow of Gp

lemma *carrier-pow-mod-order-Gp*:
 assumes $g \in \text{carrier } G_p$
 shows $g [\cdot]_{G_p} x = g [\cdot]_{G_p} (x \text{ mod } p)$
proof –
 have $p = (\text{order } G_p)$ **by** (*simp add: CARD-Gp*)
 also have $g [\cdot]_{G_p} x = g [\cdot]_{G_p} (x \text{ mod } \text{order } G_p)$
proof –
 have $g [\cdot]_{G_p} x \in \text{carrier } G_p$ **by** (*simp add: assms*)
 let $?d = x \text{ div } (\text{order } G_p)$
 have $g [\cdot]_{G_p} x = g [\cdot]_{G_p} (?d * \text{order } G_p + x \text{ mod } \text{order } G_p)$
 using *div-mult-mod-eq* **by** *presburger*
 also have $\dots = g [\cdot]_{G_p} (?d * \text{order } G_p) \otimes_{G_p} g [\cdot]_{G_p} (x \text{ mod } \text{order } G_p)$
 using $G_p.\text{int-pow-mult}$ *assms* **by** *blast*
 also have $\dots = 1_{G_p} \otimes_{G_p} g [\cdot]_{G_p} (x \text{ mod } \text{order } G_p)$
by (*metis Gp.int-pow-closed Gp.int-pow-pow Gp.pow-order-eq-1 assms int-pow-int*)
 finally show $g [\cdot]_{G_p} x = g [\cdot]_{G_p} (x \text{ mod } \text{order } G_p)$
 using *assms* **by** *fastforce*
qed
 finally show $g [\cdot]_{G_p} x = g [\cdot]_{G_p} (x \text{ mod } p)$.
qed

lemma *pow-mod-order-Gp*: $\mathbf{g}_{G_p} [\cdot]_{G_p} x = \mathbf{g}_{G_p} [\cdot]_{G_p} (x \text{ mod } p)$
 using *carrier-pow-mod-order-Gp* **by** *blast*

lemma *mod-ring-pow-mult-inv-Gp[symmetric]*: $(\mathbf{g}_{G_p} [\cdot]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) \otimes_{G_p} \text{inv}_{G_p} (\mathbf{g}_{G_p} [\cdot]_{G_p} (\text{to-int-mod-ring } b))$
 $= \mathbf{g}_{G_p} [\cdot]_{G_p} (\text{to-int-mod-ring } (a-b))$
proof –
 have $(\mathbf{g}_{G_p} [\cdot]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) \otimes_{G_p} \text{inv}_{G_p} (\mathbf{g}_{G_p} [\cdot]_{G_p} (\text{to-int-mod-ring } b))$
 $= (\mathbf{g}_{G_p} [\cdot]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) \otimes_{G_p} (\mathbf{g}_{G_p} [\cdot]_{G_p} (-\text{to-int-mod-ring } b))$
by (*simp add: Gp.int-pow-neg*)
 also have $\dots = (\mathbf{g}_{G_p} [\cdot]_{G_p} ((\text{to-int-mod-ring } a + -\text{to-int-mod-ring } b) \text{ mod } \text{CARD } ('q)))$

using *pow-mod-order- G_p CARD-q G_p .generator-closed G_p .int-pow-mult* **by**
presburger
also have $\dots = (\mathbf{g}_{G_p} [\bigwedge]_{G_p} ((\text{to-int-mod-ring } a - \text{to-int-mod-ring } b) \bmod \text{CARD}$
 $(\text{'q})))$
by *fastforce*
also have $\dots = \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } (a - b)$
by (*simp add: minus-mod-ring.rep-eq to-int-mod-ring.rep-eq*)
finally show $\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } a \otimes_{G_p} \text{inv}_{G_p} (\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring}$
 $b) = \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } (a - b) .$
qed

lemma *mod-ring-pow-mult- G_p [symmetric]:* $(\mathbf{g}_{G_p} [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a::\text{'q mod-ring})))$
 $\otimes_{G_p} (\mathbf{g}_{G_p} [\bigwedge]_{G_p} (\text{to-int-mod-ring } b))$
 $= \mathbf{g}_{G_p} [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a+b))$

proof –

have $\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } a \otimes_{G_p} \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } b = \mathbf{g}_{G_p}$
 $[\bigwedge]_{G_p} (\text{to-int-mod-ring } a + \text{to-int-mod-ring } b)$
by (*simp add: G_p .int-pow-mult*)
also have $\dots = \mathbf{g}_{G_p} [\bigwedge]_{G_p} ((\text{to-int-mod-ring } a + \text{to-int-mod-ring } b) \bmod (\text{CARD}$
 $(\text{'q})))$
using *pow-mod-order- G_p CARD-q* **by** *blast*
also have $\dots = \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } (a + b)$
by (*simp add: plus-mod-ring.rep-eq to-int-mod-ring.rep-eq*)
finally show $\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } a \otimes_{G_p} \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } b =$
 $\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } (a + b) .$
qed

lemma *mod-ring-pow-pow- G_p [symmetric]:* $(\mathbf{g}_{G_p} [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a::\text{'q mod-ring})))$
 $[\bigwedge]_{G_p} (\text{to-int-mod-ring } (b::\text{'q mod-ring}))$
 $= \mathbf{g}_{G_p} [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a*b::\text{'q mod-ring}))$

proof –

have $(\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } a) [\bigwedge]_{G_p} \text{to-int-mod-ring } b = (\mathbf{g}_{G_p} [\bigwedge]_{G_p} (\text{to-int-mod-ring}$
 $a * \text{to-int-mod-ring } b))$
using *G_p .int-pow-pow* **by** *auto*
also have $\dots = (\mathbf{g}_{G_p} [\bigwedge]_{G_p} ((\text{to-int-mod-ring } a * \text{to-int-mod-ring } b) \bmod \text{CARD}$
 $(\text{'q})))$
using *CARD-q pow-mod-order- G_p* **by** *blast*
also have $\dots = \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } (a * b)$
by (*simp add: times-mod-ring.rep-eq to-int-mod-ring.rep-eq*)
finally show $(\mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } a) [\bigwedge]_{G_p} \text{to-int-mod-ring } b$
 $= \mathbf{g}_{G_p} [\bigwedge]_{G_p} \text{to-int-mod-ring } (a * b) .$
qed

lemma *to-int-mod-ring-ge-0:* *to-int-mod-ring* $x \geq 0$
using *range-to-int-mod-ring* **by** *fastforce*

lemma *pow-on-eq-card:* $(\mathbf{g}_{G_p} \hat{}_{G_p} x = \mathbf{g}_{G_p} \hat{}_{G_p} y) = (x=y)$

proof

assume *assm*: $\mathbf{g} \hat{}_{G_p} x = \mathbf{g} \hat{}_{G_p} y$

then have $\mathbf{g}_{G_p} [\cdot]_{G_p} \text{to-int-mod-ring } x = \mathbf{g}_{G_p} [\cdot]_{G_p} \text{to-int-mod-ring } y$

using *assm* **by** *blast*

then have $\mathbf{g}_{G_p} [\cdot]_{G_p} \text{nat (to-int-mod-ring } x) = \mathbf{g}_{G_p} [\cdot]_{G_p} \text{nat (to-int-mod-ring } y)$

using *to-int-mod-ring-ge-0[of x] to-int-mod-ring-ge-0[of y]* **by** *fastforce*

then have $[\text{nat (to-int-mod-ring } x) = \text{nat (to-int-mod-ring } y)] \text{ (mod order } G_p)$

using *G_p.pow-generator-eq-iff-cong G_p.finite-carrier* **by** *fast*

then have $[\text{to-int-mod-ring } x = \text{to-int-mod-ring } y] \text{ (mod order } G_p)$

using *to-int-mod-ring-ge-0[of x] to-int-mod-ring-ge-0[of y]*

by *(metis cong-int-iff int-nat-eq)*

then have $[\text{to-int-mod-ring } x = \text{to-int-mod-ring } y] \text{ (mod } p)$

using *CARD-G_p* **by** *fast*

then have *to-int-mod-ring x = to-int-mod-ring y*

by *(metis Rep-mod-ring-mod to-int-mod-ring.rep-eq unique-euclidean-semiring-class.cong-def*

CARD-q)

then show $x = y$ **by** *force*

next

assume $x = y$

then show $\mathbf{g} \hat{}_{G_p} x = \mathbf{g} \hat{}_{G_p} y$ **by** *fast*

qed

lemma *g-pow-to-int-mod-ring-of-int-mod-ring*: $\mathbf{g} \hat{}_{G_p} \text{of-int-mod-ring } x = \mathbf{g} [\cdot]_x$

proof –

have $\mathbf{g} \hat{}_{G_p} \text{of-int-mod-ring } x = \mathbf{g} [\cdot] (x \text{ mod } p)$

by *(simp add: CARD-q of-int-mod-ring.rep-eq to-int-mod-ring.rep-eq)*

also have $\dots = \mathbf{g} [\cdot] x$ **using** *CARD-G_p G_p.pow-generator-mod-int* **by** *presburger*

finally show *?thesis* .

qed

lemma *g-pow-to-int-mod-ring-of-int-mod-ring-pow-t*: $\mathbf{g} \hat{}_{G_p} \text{of-int-mod-ring } x \hat{}_{(t::\text{nat})} = \mathbf{g} [\cdot] x \hat{}^t$

by *(metis g-pow-to-int-mod-ring-of-int-mod-ring of-int-of-int-mod-ring of-int-power)*

lemma *carrier-inj-on-multc*: $c \neq 0 \implies \text{inj-on } (\lambda x. x \hat{}_{G_p} c) \text{ (carrier } G_p)$

proof

fix $x y$

assume $c: c \neq 0$

assume $x: x \in \text{carrier } G_p$

assume $y: y \in \text{carrier } G_p$

assume *asm*: $x \hat{}^c = y \hat{}^c$

obtain $x\text{-pow}$ **where** $x\text{-pow}: \mathbf{g} \hat{}_{G_p} x\text{-pow} = x$

using x

by *(metis G_p.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int)*

obtain $y\text{-pow}$ **where** $y\text{-pow}: \mathbf{g} \hat{}_{G_p} y\text{-pow} = y$

using y

```

    by (metis  $G_p$ .generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int)
  then have  $(g \hat{\cdot}_{G_p} x\text{-pow}) \hat{\cdot}_{G_p} c = (g \hat{\cdot}_{G_p} y\text{-pow}) \hat{\cdot}_{G_p} c$ 
    using asm x-pow y-pow by blast
  then have  $(g \hat{\cdot}_{G_p} (x\text{-pow} * c)) = (g \hat{\cdot}_{G_p} (y\text{-pow} * c))$ 
    using mod-ring-pow-pow- $G_p$  by presburger
  then have  $x\text{-pow} * c = y\text{-pow} * c$ 
    using pow-on-eq-card by force
  then have  $x\text{-pow} = y\text{-pow}$ 
    using c by simp
  then show  $x=y$ 
    using x-pow y-pow by blast
qed

```

2.0.2 mod_ring operations on pow of GT

```

lemma pow-mod-order- $G_T$ :  $g \in \text{carrier } G_T \implies g \llbracket_{G_T} x = g \llbracket_{G_T} (x \bmod p)$ 
proof -
  assume asmpt:  $g \in \text{carrier } G_T$ 
  have  $p = (\text{order } G_T)$  by (simp add: CARD- $G_T$ )
  also have  $g \llbracket_{G_T} x = g \llbracket_{G_T} (x \bmod \text{order } G_T)$ 
  proof -
    have  $g \llbracket_{G_T} x \in \text{carrier } G_T$  using asmpt by simp
    let  $?d = x \text{ div } (\text{order } G_T)$ 
    have  $g \llbracket_{G_T} x = g \llbracket_{G_T} (?d * \text{order } G_T + x \bmod \text{order } G_T)$ 
      using div-mult-mod-eq by presburger
    also have  $\dots = g \llbracket_{G_T} (?d * \text{order } G_T) \otimes_{G_T} g \llbracket_{G_T} (x \bmod \text{order } G_T)$ 
      using  $G_T$ .int-pow-mult asmpt by fast
    also have  $\dots = \mathbf{1}_{G_T} \otimes_{G_T} g \llbracket_{G_T} (x \bmod \text{order } G_T)$ 
      by (metis  $G_T$ .int-pow-one  $G_T$ .int-pow-pow  $G_T$ .pow-order-eq-1 int-pow-int
mult commute asmpt)
    finally show  $g \llbracket_{G_T} x = g \llbracket_{G_T} (x \bmod \text{order } G_T)$ 
      using asmpt by fastforce
  qed
  finally show  $g \llbracket_{G_T} x = g \llbracket_{G_T} (x \bmod p)$  .
qed

```

```

lemma mod-ring-pow-mult- $G_T$ [symmetric]:  $x \in \text{carrier } G_T \implies (x \llbracket_{G_T} (\text{to-int-mod-ring }
(a::'q \bmod \text{ring}))) \otimes_{G_T} (x \llbracket_{G_T} (\text{to-int-mod-ring } b))
= x \llbracket_{G_T} (\text{to-int-mod-ring } (a+b))$ 
proof -
  assume asmpt:  $x \in \text{carrier } G_T$ 
  have  $x \llbracket_{G_T} \text{to-int-mod-ring } a \otimes_{G_T} x \llbracket_{G_T} \text{to-int-mod-ring } b = x \llbracket_{G_T}
(\text{to-int-mod-ring } a + \text{to-int-mod-ring } b)$ 
    by (simp add:  $G_T$ .int-pow-mult asmpt)
  also have  $\dots = x \llbracket_{G_T} ((\text{to-int-mod-ring } a + \text{to-int-mod-ring } b) \bmod (\text{CARD }
('q)))$ 
    using pow-mod-order- $G_T$  CARD-q asmpt by blast
  also have  $\dots = x \llbracket_{G_T} \text{to-int-mod-ring } (a + b)$ 

```

by (simp add: plus-mod-ring.rep-eq to-int-mod-ring.rep-eq)
finally show $x [\bigwedge]_{G_T} \text{to-int-mod-ring } a \otimes_{G_T} x [\bigwedge]_{G_T} \text{to-int-mod-ring } b = x [\bigwedge]_{G_T} \text{to-int-mod-ring } (a + b)$.
qed

2.0.3 bilinearity operations for mod_ring elements

lemma *e-bilinear[simp]*: $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \implies$
 $e (P [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) (Q [\bigwedge]_{G_p} (\text{to-int-mod-ring } b))$
 $= (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } (a*b))$
proof –
 assume *asm*: $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
 then have $e (P [\bigwedge] \text{to-int-mod-ring } a) (Q [\bigwedge] \text{to-int-mod-ring } b) = e P Q [\bigwedge]_{G_T}$
 $(\text{to-int-mod-ring } a * \text{to-int-mod-ring } b)$
 by *simp*
 also have $\dots = (e P Q [\bigwedge]_{G_T} ((\text{to-int-mod-ring } a * \text{to-int-mod-ring } b) \text{ mod } \text{CARD } ('q)))$
 using *CARD-q pow-mod-order-G_T asm e-symmetric* **by** *blast*
 also have $\dots = e P Q [\bigwedge]_{G_T} \text{to-int-mod-ring } (a * b)$
 by (simp add: times-mod-ring.rep-eq to-int-mod-ring.rep-eq)
 finally show $e (P [\bigwedge] \text{to-int-mod-ring } a) (Q [\bigwedge] \text{to-int-mod-ring } b) = e P Q$
 $[\bigwedge]_{G_T} \text{to-int-mod-ring } (a * b)$.
qed

lemma *e-linear-in-fst*:
 assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
 shows $e (P [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) (Q) = (e P Q) [\bigwedge]_{G_T}$
 $(\text{to-int-mod-ring } a)$
proof –
 have $e (P [\bigwedge]_{G_p} \text{to-int-mod-ring } a) Q = e (P [\bigwedge]_{G_p} \text{to-int-mod-ring } a) (Q [\bigwedge]_{G_p}$
 $\text{to-int-mod-ring } (1::'q \text{ mod-ring}))$ **using** *assms* **by** *simp*
 also have $\dots = (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } (a*(1::'q \text{ mod-ring})))$ **using**
assms e-bilinear **by** *fast*
 also have $\dots = (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } a)$ **by** *simp*
 finally show $e (P [\bigwedge]_{G_p} (\text{to-int-mod-ring } a)) Q = (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } a)$.
qed

lemma *e-linear-in-snd*:
 assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
 shows $e (P) (Q [\bigwedge]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) = (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } a)$
proof –
 have $e P (Q [\bigwedge]_{G_p} \text{to-int-mod-ring } a) = e (P [\bigwedge]_{G_p} \text{to-int-mod-ring } (1::'q \text{ mod-ring}))$
 $(Q [\bigwedge]_{G_p} \text{to-int-mod-ring } a)$ **using** *assms* **by** *simp*
 also have $\dots = (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } ((1::'q \text{ mod-ring})*a))$ **using**
assms e-bilinear **by** *fast*
 also have $\dots = (e P Q) [\bigwedge]_{G_T} (\text{to-int-mod-ring } a)$ **by** *simp*

finally show $e P (Q [\bigwedge]_{G_p} \text{ to-int-mod-ring } a) = e P Q [\bigwedge]_{G_T} \text{ to-int-mod-ring } a$
 .
 qed

lemma *addition-in-exponents-on-e[simp]*:
 assumes $x \in \text{carrier } G_p \wedge y \in \text{carrier } G_p$
 shows $(e x y) \hat{}_{G_T} a \otimes_{G_T} (e x y) \hat{}_{G_T} b = (e x y) \hat{}_{G_T} (a+b)$
 using *assms*
 by (*metis PiE e-symmetric mod-ring-pow-mult-G_T*)

lemma *e-from-generators-ne-1*: $e \mathbf{g}_{G_p} \mathbf{g}_{G_p} \neq \mathbf{1}_{G_T}$
proof

assume *asm*: $e \mathbf{g}_{G_p} \mathbf{g}_{G_p} = \mathbf{1}_{G_T}$
 have $\forall P Q. P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow e P Q = \mathbf{1}_{G_T}$
proof(*intro allI*)
 fix $P Q$
 show $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow e P Q = \mathbf{1}_{G_T}$
proof
 assume $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
 then obtain $p q :: \text{int}$ where $\mathbf{g}_{G_p} [\bigwedge]_{G_p} p = P \wedge \mathbf{g}_{G_p} [\bigwedge]_{G_p} q = Q$
 by (*metis G_p.generatorE int-pow-int*)
 then have $e P Q = e (\mathbf{g}_{G_p} [\bigwedge]_{G_p} p) (\mathbf{g}_{G_p} [\bigwedge]_{G_p} q)$
 by *blast*
 also have $\dots = e \mathbf{g}_{G_p} \mathbf{g}_{G_p} [\bigwedge]_{G_T} (p*q)$
 by *force*
 also have $\dots = \mathbf{1}_{G_T} [\bigwedge]_{G_T} (p*q)$
 using *asm* by *argo*
 also have $\dots = \mathbf{1}_{G_T}$
 by *fastforce*
 finally show $e P Q = \mathbf{1}_{G_T}$.
 qed
 qed
 then show *False* using *e-non-degeneracy* by *blast*
 qed

lemma *pow-on-eq-card-GT[simp]*: $(\mathbf{g}_{G_T} \hat{}_{G_T} x = \mathbf{g}_{G_T} \hat{}_{G_T} y) = (x=y)$
proof
 assume *asm*: $\mathbf{g}_{G_T} \hat{}_{G_T} x = \mathbf{g}_{G_T} \hat{}_{G_T} y$
 then have $\mathbf{g}_{G_T} [\bigwedge]_{G_T} \text{ to-int-mod-ring } x = \mathbf{g}_{G_T} [\bigwedge]_{G_T} \text{ to-int-mod-ring } y$
 using *asm* by *blast*
 then have $\mathbf{g}_{G_T} [\bigwedge]_{G_T} \text{ nat } (\text{to-int-mod-ring } x) = \mathbf{g}_{G_T} [\bigwedge]_{G_T} \text{ nat } (\text{to-int-mod-ring } y)$
 using *to-int-mod-ring-ge-0[of x] to-int-mod-ring-ge-0[of y]* by *fastforce*
 then have $[\text{nat } (\text{to-int-mod-ring } x) = \text{nat } (\text{to-int-mod-ring } y)] (\text{mod order } G_T)$
 using *G_T.pow-generator-eq-iff-cong G_T.finite-carrier* by *fast*
 then have $[\text{to-int-mod-ring } x = \text{to-int-mod-ring } y] (\text{mod order } G_T)$
 using *to-int-mod-ring-ge-0[of x] to-int-mod-ring-ge-0[of y]*
 by (*metis cong-int-iff int-nat-eq*)
 then have $[\text{to-int-mod-ring } x = \text{to-int-mod-ring } y] (\text{mod } p)$

```

    using CARD-GT by fast
  then have to-int-mod-ring  $x = \text{to-int-mod-ring } y$ 
    by (metis arith-simps(49) to-int-mod-ring-add to-int-mod-ring-hom.hom-0-iff
        unique-euclidean-semiring-class.cong-def CARD-q)
  then show  $x = y$  by force
next
  assume  $x = y$ 
  then show  $\mathbf{g}_{G_T} \hat{\phantom{x}}_{G_T} x = \mathbf{g}_{G_T} \hat{\phantom{x}}_{G_T} y$  by fast
qed

lemma pow-on-eq-card-GT-carrier-ext'[simp]:
   $((e \mathbf{g}_{G_p} \mathbf{g}_{G_p})) \hat{\phantom{x}}_{G_T} x = ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p})) \hat{\phantom{x}}_{G_T} y \longleftrightarrow x=y$ 
proof
  assume g-pow-x-eq-g-pow-y:  $e \mathbf{g} \mathbf{g} \hat{\phantom{x}}_{G_T} x = e \mathbf{g} \mathbf{g} \hat{\phantom{x}}_{G_T} y$ 
  obtain g-exp::nat where  $e \mathbf{g} \mathbf{g} = \mathbf{g}_{G_T} [\hat{\phantom{x}}]_{G_T} g\text{-exp}$ 
    using  $G_T.\text{generatorE } e\text{-g-g-in-carrier-GT}$  by blast
  then have g-exp:  $e \mathbf{g} \mathbf{g} = \mathbf{g}_{G_T} \hat{\phantom{x}}_{G_T} (\text{of-int-mod-ring } (\text{int } g\text{-exp}))$ 
    by (metis CARD-GT  $G_T.\text{pow-generator-mod-int}$  math-primitives.CARD-q math-primitives-axioms
        int-pow-int of-int-mod-ring.rep-eq to-int-mod-ring.rep-eq)
  let ?g-exp = of-int-mod-ring (int g-exp)
  have  $(e \mathbf{g} \mathbf{g}) \hat{\phantom{x}}_{G_T} x = \mathbf{g}_{G_T} \hat{\phantom{x}}_{G_T} (\text{of-int-mod-ring } (\text{int } g\text{-exp}) * x)$ 
    using g-exp
  by (metis CARD-GT  $G_T.\text{generator-closed } G_T.\text{int-pow-pow } G_T.\text{pow-generator-mod-int}$ 
        math-primitives.CARD-q math-primitives-axioms times-mod-ring.rep-eq to-int-mod-ring.rep-eq)
  moreover have  $(e \mathbf{g} \mathbf{g}) \hat{\phantom{x}}_{G_T} y = \mathbf{g}_{G_T} \hat{\phantom{x}}_{G_T} (\text{of-int-mod-ring } (\text{int } g\text{-exp}) * y)$ 
    using g-exp
  by (metis CARD-GT  $G_T.\text{generator-closed } G_T.\text{int-pow-pow } G_T.\text{pow-generator-mod-int}$ 
        math-primitives.CARD-q math-primitives-axioms times-mod-ring.rep-eq to-int-mod-ring.rep-eq)
  ultimately show  $x = y$ 
    using g-pow-x-eq-g-pow-y pow-on-eq-card-GT e-from-generators-ne-1 g-exp by
force
next
  assume  $x = y$ 
  then show  $(e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{\phantom{x}}_{G_T} x = (e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{\phantom{x}}_{G_T} y$ 
    by blast
qed

end

end

theory DL-assumption

imports Sigma-Commit-Crypto.Commitment-Schemes Berlekamp-Zassenhaus.Finite-Field

begin

```

3 The Discrete Logarithm Assumption

The DL game and advantage as in Definition 2.1 of the original KZG paper “Constant-Size Commitments to Polynomials and Their Applications” [8].

```

locale DL = G : cyclic-group G
  for G:: ('a, 'b) cyclic-group-scheme (structure)
    and to-type :: nat  $\Rightarrow$  ('c::prime-card) mod-ring
    and exp :: 'a  $\Rightarrow$  'c mod-ring  $\Rightarrow$  'a
begin

type-synonym ('grp, 'mr) adversary = 'grp  $\Rightarrow$  'mr mod-ring spmf

```

The discrete Logarithm game

```

definition game :: ('a, 'c) adversary  $\Rightarrow$  bool spmf where
  game A = TRY do {
    a  $\leftarrow$  sample-uniform (Coset.order G);
    a'  $\leftarrow$  A (exp g (to-type a));
    return-spmf (to-type a = a')
  } ELSE return-spmf False

```

The advantage is that the Adversary wins the game. For the DL assumption to hold this advantage should be negligible.

```

definition advantage :: ('a, 'c) adversary  $\Rightarrow$  real
  where advantage A = spmf (game A) True

```

An alternative but equivalent game for the DL-game. This alternative game encapsulates the event that the Adversary wins in the `assert_spmf` statement. adapted proof from `Sigma_Commit_Crypto.Commitment_Schemes` `bind_game_alt_def`

```

lemma game-alt-def:
  game A = TRY do {
    a  $\leftarrow$  sample-uniform (Coset.order G);
    a'  $\leftarrow$  A (exp g (to-type a));
    ::unit  $\leftarrow$  assert-spmf (to-type a = a');
    return-spmf True
  } ELSE return-spmf False
  (is ?lhs = ?rhs)

```

proof –

```

  have ?lhs = TRY do {
    a  $\leftarrow$  sample-uniform (Coset.order G);
    TRY do {
      a'  $\leftarrow$  A (exp g (to-type a));
      TRY do {
        return-spmf (to-type a = a')
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False

```

```

unfolding split-def game-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
also have ... = TRY do {
  a ← sample-uniform (Coset.order G);
  TRY do {
    a' ← A (exp g (to-type a));
    TRY do {
      - :: unit ← assert-spmf (to-type a = a');
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
} ELSE return-spmf False
by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong
bind-spmf-cong)
also have ... = ?rhs
unfolding split-def Let-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
finally show ?thesis .
qed

```

```

lemma game-alt-def2:
  game A = TRY do {
    a ← map-spmf to-type (sample-uniform (Coset.order G));
    a' ← A (exp g a);
    return-spmf (a = a')
  } ELSE return-spmf False
by (simp add: game-def bind-map-spmf o-def)

```

end

end

theory *tDL-assumption*

imports *Sigma-Commit-Crypto.Commitment-Schemes Berlekamp-Zassenhaus.Finite-Field*

begin

4 The t-Discrete Logarithm Assumption

The t-DL game and advantage defined as in section 6 of “The Algebraic Group Model and its Applications” by Fuchsbauer, Kiltz, and Loss [6].

```

locale t-DL = G : cyclic-group G
  for G:: ('a, 'b) cyclic-group-scheme (structure)
  and t::nat
  and to-type :: nat ⇒ ('c::prime-card) mod-ring
  and exp :: 'a ⇒ 'c mod-ring ⇒ 'a
begin

```

type-synonym ('grp,'mr) adversary = 'grp list $\Rightarrow$ ('mr mod-ring) spmf

The t-DL game states that given a t+1-long tuple in the form of $(g, g^\alpha, g^{\alpha^2}, \dots, g^{\alpha^t})$ the Adversary has to return α .

definition game :: ('a,'c) adversary $\Rightarrow$ bool spmf **where**
 game $\mathcal{A} = \text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G);$
 $\alpha' \leftarrow \mathcal{A} (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \wedge t')) [0..<t+1]);$
 $\text{return-spmf } (\text{to-type } \alpha = \alpha')$
 $\} \text{ ELSE return-spmf False}$

The advantage is that the Adversary wins the game. For the t-DL assumption to hold this advantage should be negligible.

definition advantage :: ('a,'c) adversary $\Rightarrow$ real
where advantage $\mathcal{A} = \text{spmfs } (\text{game } \mathcal{A}) \text{ True}$

An alternative but equivalent game for the t-DL-game. This alternative game encapsulates the event that the Adversary wins in the assert_spmf statement. adapted proof from Sigma_Commit_Crypto.Commitment_Schemes bind_game_alt_def

lemma game-alt-def:

game $\mathcal{A} = \text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G);$
 $\alpha' \leftarrow \mathcal{A} (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \wedge t')) [0..<t+1]);$
 $\text{--::unit} \leftarrow \text{assert-spmf } (\text{to-type } \alpha = \alpha');$
 return-spmf True
 $\} \text{ ELSE return-spmf False}$
 (is ?lhs = ?rhs)

proof –

have ?lhs = $\text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G);$
 $\text{TRY do } \{$
 $\alpha' \leftarrow \mathcal{A} (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \wedge t')) [0..<t+1]);$
 $\text{TRY return-spmf } (\text{to-type } \alpha = \alpha')$
 $\text{ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$
unfolding split-def game-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
also have ... = $\text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G);$
 $\text{TRY do } \{$
 $\alpha' \leftarrow \mathcal{A} (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \wedge t')) [0..<t+1]);$
 $\text{TRY do } \{$
 $\text{--::unit} \leftarrow \text{assert-spmf } (\text{to-type } \alpha = \alpha');$
 return-spmf True
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$

```

    } ELSE return-spmf False
  by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong
bind-spmf-cong)
  also have ... = ?rhs
    unfolding split-def Let-def
    by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
  finally show ?thesis .
qed

end

end

theory tSDH-assumption

```

```

imports Sigma-Commit-Crypto.Commitment-Schemes Berlekamp-Zassenhaus.Finite-Field

begin

```

5 The t-Strong Diffie-Hellman Assumption

The t-SDH game and advantage as in section 2.3 of the original KZG paper “Constant-Size Commitments to Polynomials and Their Applications” [8].

```

locale tSDH = G : cyclic-group G
  for G:: ('a, 'b) cyclic-group-scheme (structure)
  and t::nat
  and to-type :: nat  $\Rightarrow$  ('c::prime-card) mod-ring
  and exp :: 'a  $\Rightarrow$  'c mod-ring  $\Rightarrow$  'a
begin

```

```

type-synonym ('grp, 'mr) adversary = 'grp list  $\Rightarrow$  ('mr mod-ring * 'grp) spmf

```

The t-SDH game states that given a $t+1$ -long tuple in the form of $(g, g^\alpha, g^{\alpha^2}, \dots, g^{\alpha^t})$ the Adversary has to return an element c and $g^{1/(c+\alpha)}$.

```

definition game :: ('a, 'c) adversary  $\Rightarrow$  bool spmf where
  game  $\mathcal{A}$  = TRY do {
     $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;
     $(c, g) \leftarrow \mathcal{A} (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \wedge t')) [0..<t+1])$ ;
    return-spmf (exp  $\mathbf{g} (1/((\text{to-type } \alpha)+c)) = g$ )
  } ELSE return-spmf False

```

The advantage is that the Adversary wins the game. For the t-SDH assumption to hold this advantage should be negligible.

```

definition advantage :: ('a, 'c) adversary  $\Rightarrow$  real
  where advantage  $\mathcal{A}$  = spmf (game  $\mathcal{A}$ ) True

```

An alternative but equivalent game for the t-SDH-game. This alternative game encapsulates the event that the Adversary wins in the `assert_spmf` state-

ment. adapted proof from Sigma_Commit_Crypto.Commitment_Schemes
bind_game_alt_def

lemma *game-alt-def*:

```
game  $\mathcal{A}$  = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;
   $(c, g) \leftarrow \mathcal{A} \text{ (map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')) [0..<t+1])$ ;
   $\text{--::unit} \leftarrow \text{assert-spmf } (\text{exp } \mathbf{g} (1/((\text{to-type } \alpha)+c)) = g)$ ;
  return-spmf True
} ELSE return-spmf False
(is ?lhs = ?rhs)
```

proof –

```
have ?lhs = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;
  TRY do {
     $(c, g) \leftarrow \mathcal{A} \text{ (map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')) [0..<t+1])$ ;
    TRY return-spmf  $(\text{exp } \mathbf{g} (1/((\text{to-type } \alpha)+c)) = g)$ 
    ELSE return-spmf False
  } ELSE return-spmf False
} ELSE return-spmf False
```

unfolding *split-def game-def*

by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) *simp*

also have ... = TRY do {

```
 $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;
TRY do {
   $(c, g) \leftarrow \mathcal{A} \text{ (map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')) [0..<t+1])$ ;
  TRY do {
     $\text{--} :: \text{unit} \leftarrow \text{assert-spmf } (\text{exp } \mathbf{g} (1/((\text{to-type } \alpha)+c)) = g)$ ;
    return-spmf True
  } ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
```

by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong

bind-spmf-cong)

also have ... = ?rhs

unfolding *split-def Let-def*

by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) *simp*

finally show ?thesis .

qed

end

end

theory *tBDH-assumption*

imports *Sigma-Commit-Crypto.Commitment-Schemes Berlekamp-Zassenhaus.Finite-Field*

begin

6 The t-Bilinear Strong Diffie-Hellman Assumption

The t-BSDH game and advantage as in section 2.4 of the original KZG paper “Constant-Size Commitments to Polynomials and Their Applications” [8].

The t-BSDH assumption is an extension of the t-SDH assumption (of section 2.3. in the paper). Similar to the t-SDH assumption it requires the adversary to put out some c and some group element g' such that a certain group element g exponentiated with $1/(a+c)$ is equal to g' . While the group element g was simply the generator of G in the t-SDH assumption, it is now the result of the bilinear function e of the same generator of G .

```

locale t-BSDH = G : cyclic-group G + GT : cyclic-group GT
  for G :: ('a, 'b) cyclic-group-scheme (structure) and GT :: ('c, 'd) cyclic-group-scheme
(structure)
  and t::nat
  and to-type :: nat ⇒ ('e::prime-card) mod-ring
  and exp :: 'a ⇒ 'e mod-ring ⇒ 'a
  and exp-GT :: 'c ⇒ 'e mod-ring ⇒ 'c
  and e :: 'a ⇒ 'a ⇒ 'c — bilinear function from G to GT
begin

```

```

type-synonym ('grp, 'mr, 'tgrp) adversary = 'grp list ⇒ ('mr mod-ring * 'tgrp)
spmf

```

The t-BSDH game states that given a $t+1$ -long tuple in the form of $(g, g^\alpha, g^{\alpha^2}, \dots, g^{\alpha^t})$ the Adversary has to return an element c and $e(g, g)^{1/(c+\alpha)}$.

```

definition game :: ('a, 'e, 'c) adversary ⇒ bool spmf where
  game  $\mathcal{A}$  = TRY do {
     $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;
     $(c, g) \leftarrow \mathcal{A} (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \sim^t)) [0..t+1])$ ;
    return-spmf (exp-GT (e  $\mathbf{g}$   $\mathbf{g}$ ) (1/((to-type  $\alpha$ ) + c)) = g)
  } ELSE return-spmf False

```

The advantage is that the Adversary wins the game. For the t-BSDH assumption to hold this advantage should be negligible.

```

definition advantage :: ('a, 'e, 'c) adversary ⇒ real
  where advantage  $\mathcal{A}$  = spmf (game  $\mathcal{A}$ ) True

```

An alternative but equivalent game for the t-BSDH-game. This alternative game encapsulates the event that the Adversary wins in the `assert_spmf` statement. adapted proof from `Sigma_Commit_Crypto.Commitment_Schemes`

```

bind_game_alt_def

```

```

lemma game-alt-def:
  game  $\mathcal{A}$  = TRY do {
     $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;

```

```

    (c, g) ←  $\mathcal{A}$  (map ( $\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')$ ) [0.. $t+1$ ]);
    ::unit ← assert-spmf (exp- $G_T$  (e  $\mathbf{g} \mathbf{g}$ ) (1/((to-type  $\alpha$ )+c)) = g);
    return-spmf True
  } ELSE return-spmf False
(is ?lhs = ?rhs)
proof –
  have ?lhs = TRY do {
     $\alpha$  ← sample-uniform (Coset.order  $G$ );
    TRY do {
      (c, g) ←  $\mathcal{A}$  (map ( $\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')$ ) [0.. $t+1$ ]);
      TRY return-spmf (exp- $G_T$  (e  $\mathbf{g} \mathbf{g}$ ) (1/((to-type  $\alpha$ )+c)) = g) ELSE
return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False
  unfolding split-def game-def
  by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
  also have ... = TRY do {
     $\alpha$  ← sample-uniform (Coset.order  $G$ );
    TRY do {
      (c, g) ←  $\mathcal{A}$  (map ( $\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')$ ) [0.. $t+1$ ]);
      TRY do {
        - :: unit ← assert-spmf (exp- $G_T$  (e  $\mathbf{g} \mathbf{g}$ ) (1/((to-type  $\alpha$ )+c)) = g);
        return-spmf True
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False
  by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong
bind-spmf-cong)
  also have ... = ?rhs
  unfolding split-def Let-def
  by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
  finally show ?thesis .
qed

end

end
theory Polynomial-Commitment-Schemes
imports CryptHOL.CryptHOL HOL-Computational-Algebra.Polynomial
Sigma-Commit-Crypto.Commitment-Schemes
begin

```

7 Polynomial Commitment Schemes

This theory captures the notion of Polynomial Commitment Schemes, introduced in Kate, Zaverucha, and Goldberg’s “Constant-Size Commitments to Polynomials and Their Applications” [8].

The formalization differs slightly from the early notion of Kate, Zaverucha,

and Goldberg, as it aims to capture, and also draws from, newer approaches to polynomial commitment schemes. Examples are WHIR [1], BaseFold [11], Binius [5], and Dory [9].

Additionally, the formalization is formulated even more general to be able to capture batching schemes as well, which were already introduced by Kate, Zaverucha, and Goldberg.

```
locale abstract-polynomial-commitment-scheme =
  fixes key-gen :: ('ck × 'vk) spmf — outputs the keys received by the two parties
  and commit :: 'ck ⇒ 'r::comm-monoid-add poly ⇒ ('commit × 'trapdoor) spmf

    — outputs the commitment as well as the secret, which might be used to derive
    witnesses, and the opening values sent by the committer in the reveal phase
  and verify-poly :: 'vk ⇒ 'r poly ⇒ 'commit ⇒ 'trapdoor ⇒ bool
    — checks whether the polynomial corresponds to the commitment
  and eval :: 'ck ⇒ 'trapdoor ⇒ 'r poly ⇒ 'argument ⇒ ('evaluation × 'witness)
    — outputs a point and a witness
  and verify-eval :: 'vk ⇒ 'commit ⇒ 'argument ⇒ ('evaluation × 'witness) ⇒
bool
    — checks whether the point is on the polynomial corresponding to the com-
    mitment
  and valid-poly :: 'r poly ⇒ bool — checks whether a polynomial is a valid
  message e.g. it's degree is bounded
  and valid-argument :: 'argument ⇒ bool — checks whether an argument is a
  valid message e.g. set size in a batched version
  and valid-eval :: ('evaluation × 'witness) ⇒ bool — checks whether an evaluation
  is a valid message e.g. a valid group element
begin
```

A polynomial commitment scheme is an extension of a standard commitment scheme. We reuse the work by Butler, Lochbihler, Aspinall and Gascón, who already formalized commitment schemes [3].

```
sublocale cs: abstract-commitment key-gen commit verify-poly valid-poly .
```

```
definition correct-cs-game :: 'r poly ⇒ bool spmf
  where correct-cs-game ≡ cs.correct-game
```

```
definition correct-cs
  where correct-cs ≡ cs.correct
```

This game captures the correctness property of eval i.e. the results of eval will always verify.

```
definition correct-eval-game :: 'r poly ⇒ 'argument ⇒ bool spmf
  where correct-eval-game p i = do {
    (ck, vk) ← key-gen;
    (c, d) ← commit ck p;
    let w = eval ck d p i;
    return-spmf (verify-eval vk c i w)
```

}

lemma *lossless-correct-eval-game*: $\llbracket \text{lossless-spmf key-gen};$
 $\bigwedge ck\ p. \text{valid-msg } p \implies \text{lossless-spmf } (\text{commit } ck\ p) \rrbracket$
 $\implies \text{valid-msg } p \implies \text{lossless-spmf } (\text{correct-eval-game } p\ i)$
by (*simp add: correct-eval-game-def split-def Let-def*)

captures the perfect correctness property of eval

definition *correct-eval*

where *correct-eval* $\equiv (\forall p\ i. \text{valid-poly } p \longrightarrow \text{valid-argument } i \longrightarrow \text{spm}f\ (\text{correct-eval-game } p\ i)\ \text{True} = 1)$

We again reuse the previous work on commitment schemes

definition *poly-bind-game*

where *poly-bind-game* $\equiv cs.\text{bind-game}$

definition *poly-bind-advantage*

where *poly-bind-advantage* $\equiv cs.\text{bind-advantage}$

type-synonym ('ck', 'commit', 'argument', 'evaluation', 'witness') *eval-bind-adversary*

=

'ck' $\Rightarrow$ ('commit' $\times$ 'argument' $\times$ 'evaluation' $\times$ 'witness' $\times$ 'evaluation' $\times$ 'witness') *spm*f

captures the evaluation binding game i.e. verifying two contradicting evaluations ($p(i) \neq p(i')$).

definition *eval-bind-game* :: ('ck', 'commit', 'argument', 'evaluation', 'witness') *eval-bind-adversary* $\Rightarrow$ *bool spm*f

where *eval-bind-game* $\mathcal{A} = \text{TRY do } \{$
 $(ck, vk) \leftarrow \text{key-gen};$
 $(c, i, v, w, v', w') \leftarrow \mathcal{A}\ ck;$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (v \neq v' \wedge \text{valid-argument } i \wedge \text{valid-eval } (v, w) \wedge \text{valid-eval } (v', w'));$
 $\text{let } b = \text{verify-eval } vk\ c\ i\ (v, w);$
 $\text{let } b' = \text{verify-eval } vk\ c\ i\ (v', w');$
 $\text{return-spmf } (b \wedge b') \}$ *ELSE return-spmf False*

We capture the advantage of an adversary over winning the evaluation binding game. This has to be negligible for evaluation binding to hold.

definition *eval-bind-advantage* :: ('ck', 'commit', 'argument', 'evaluation', 'witness') *eval-bind-adversary* $\Rightarrow$ *real*

where *eval-bind-advantage* $\mathcal{A} \equiv \text{spm}f\ (\text{eval-bind-game } \mathcal{A})\ \text{True}$

type-synonym ('vk', 'argument', 'state') *eval-hiding-adversary1* =

'vk' $\Rightarrow$ ('argument' *list* $\times$ 'state') *spm*f

type-synonym ('r', 'vk', 'commit', 'argument', 'evaluation', 'witness', 'state') *eval-hiding-adversary2* =

$(\text{'vk'} \Rightarrow \text{'state'} \Rightarrow \text{'commit'} \Rightarrow \text{'argument'} \text{ list} \Rightarrow (\text{'evaluation'} \times \text{'witness'}) \text{ list} \Rightarrow (\text{'r'} \text{ poly}) \text{ spmf})$

captures the hiding property of the Commit and Eval functions in combination. Note, this property deviates from the typical indistinguishability games for hiding in general. Kate, Zaverucha, and Goldberg introduced this notion in their work.

definition $\text{eval-hiding-game} :: \text{'r poly} \Rightarrow (\text{'vk}, \text{'argument}, \text{'state}) \text{ eval-hiding-adversary1} \Rightarrow$
 $(\text{'r}, \text{'vk}, \text{'commit}, \text{'argument}, \text{'evaluation}, \text{'witness}, \text{'state}) \text{ eval-hiding-adversary2}$
 $\Rightarrow \text{bool spmf}$
where $\text{eval-hiding-game } p \text{ } \mathcal{A1} \text{ } \mathcal{A2} = \text{TRY do } \{$
 $(ck, vk) \leftarrow \text{key-gen};$
 $(I, \sigma) \leftarrow \mathcal{A1} \text{ } vk;$
 $(c, d) \leftarrow \text{commit } ck \text{ } p;$
 $\text{let } W = \text{map } (\lambda i. \text{eval } ck \text{ } d \text{ } p \text{ } i) \text{ } I;$
 $p' \leftarrow \mathcal{A2} \text{ } vk \text{ } \sigma \text{ } c \text{ } I \text{ } W;$
 $\text{return-spmf } (p = p') \text{ } \text{ELSE return-spmf False}$

We capture the advantage of an adversary over winning the hiding game. This has to be negligible for hiding to hold.

definition $\text{eval-hiding-advantage} :: \text{'r poly} \Rightarrow (\text{'vk}, \text{'argument}, \text{'state}) \text{ eval-hiding-adversary1} \Rightarrow$
 $(\text{'r}, \text{'vk}, \text{'commit}, \text{'argument}, \text{'evaluation}, \text{'witness}, \text{'state}) \text{ eval-hiding-adversary2}$
 $\Rightarrow \text{real}$
where $\text{eval-hiding-advantage } p \text{ } \mathcal{A1} \text{ } \mathcal{A2} \equiv \text{spmf } (\text{eval-hiding-game } p \text{ } \mathcal{A1} \text{ } \mathcal{A2})$
 True

type-synonym $(\text{'ck'}, \text{'commit'}, \text{'state'}) \text{ knowledge-soundness-adversary1} = \text{'ck'} \Rightarrow$
 $(\text{'commit'} \times \text{'state'}) \text{ spmf}$

type-synonym $(\text{'state'}, \text{'ck'}, \text{'argument'}, \text{'evaluation'}, \text{'witness'}) \text{ knowledge-soundness-adversary2}$
 $= \text{'ck'} \Rightarrow \text{'state'} \Rightarrow (\text{'argument'} \times (\text{'evaluation'} \times \text{'witness'})) \text{ spmf}$

type-synonym $(\text{'r'}, \text{'commit'}, \text{'trapdoor'}) \text{ extractor} = \text{'commit'} \Rightarrow (\text{'r'} \text{ poly} \times$
 $\text{'trapdoor'}) \text{ spmf}$

captures intuitively the fact that an adversary has to have knowledge of a polynomial in order to create an evaluation that verifies. This property is typically required for succinct non-interactive arguments of knowledge (SNARKs) built from polynomial commitment schemes, e.g. PLONK [7], Marlin [4], and Binius [5].

definition $\text{knowledge-soundness-game} :: (\text{'ck'}, \text{'commit'}, \text{'state'}) \text{ knowledge-soundness-adversary1}$
 $\Rightarrow (\text{'state'}, \text{'ck'}, \text{'argument'}, \text{'evaluation'}, \text{'witness'}) \text{ knowledge-soundness-adversary2}$
 $\Rightarrow (\text{'r'}, \text{'commit'}, \text{'trapdoor'}) \text{ extractor} \Rightarrow \text{bool spmf}$

```

where knowledge-soundness-game  $\mathcal{A}1 \ \mathcal{A}2 \ E = TRY \ do \ \{$ 
   $(ck, vk) \leftarrow key-gen;$ 
   $(c, \sigma) \leftarrow \mathcal{A}1 \ ck;$ 
   $(p, d) \leftarrow E \ c;$ 
   $(i, (p-i, \pi)) \leftarrow \mathcal{A}2 \ ck \ \sigma;$ 
   $let \ w = (p-i, \pi);$ 
   $let \ (p-i', -) = eval \ ck \ d \ p \ i;$ 
   $return-spmf \ (verify-eval \ vk \ c \ i \ w \wedge p-i \neq p-i' \wedge valid-argument \ i \wedge valid-eval \ w)$ 
   $\}$  ELSE  $return-spmf \ False$ 

```

We capture the advantage of an adversary over winning the knowledge soundness game. This has to be negligible for knowledge soundness to hold.

definition *knowledge-soundness-advantage* :: $('ck, 'commit, 'state) \text{ knowledge-soundness-adversary1}$

$\Rightarrow ('state, 'ck, 'argument, 'evaluation, 'witness) \text{ knowledge-soundness-adversary2}$

$\Rightarrow ('r, 'commit, 'trapdoor) \text{ extractor} \Rightarrow real$

where *knowledge-soundness-advantage* $\mathcal{A}1 \ \mathcal{A}2 \ E \equiv spmf \ (knowledge-soundness-game \ \mathcal{A}1 \ \mathcal{A}2 \ E) \ True$

end

end

theory *Restrictive-Comp*

imports *CryptHOL.CryptHOL*

begin

8 Restrictive Computations

We formalize the notion of restrictive computational models. These are models in which the adversary is restricted, in the sense that its output follows certain rules that can be enforced by constraints composed of the input to and the output of the adversary.

We formalize this notion in 3 components: 1. the Select record – purpose: select relevant elements from the input 2. the Constrain record – purpose: constrain output elements to the list of relevant (selected) seen elements. 3. Restrictive_Comp locale – purpose: select relevant elements from the input and enforce constraints on the output.

This model is an abstraction over computational models like the Algebraic Group Model (AGM), the Algebraic Group Action Model (AGAM), the Algebraic Isogeny Model (AIM), and variants thereof.

We implement generalizations for model-type specific implementations to standard type constructors in Isabelle, including (e.g. group) lists, trees, multisets, finite sets, products, and finite maps. E.g. a model specific implementation of a group type can be generalized to a group list implementation.

Additionally, we provide default implementations `noSelect` and `noConstrain`, which select no element and define no constraints on all inputs.

8.1 Select

A record to select the relevant elements from a type (data structure). The naming-convention to support automation is *your_typeS*, e.g. for `int`: `intS`

record (*'a, 'b*) *Select* = *select*::*'a* $\Rightarrow$ *'b* *list* ($\langle \bowtie_1 \rangle$)

We provide a few generalized type constructor implementations.

context

fixes *r* :: (*'a, 'b*) *Select*

begin

definition *listS* ::

(*'a list, 'b*) *Select*

where *listS* $\equiv$ ($\downarrow$ *select* = ($\lambda xs.$ *concat* (*map* ($\lambda x.$ $\bowtie_r x$) *xs*)))

definition *treeS* ::

(*'a tree, 'b*) *Select*

where *treeS* $\equiv$ ($\downarrow$ *select* = ($\lambda x.$ $\bowtie_{listS}$ (*inorder* *x*)))

end

context

fixes *r* :: (*'a::linorder, 'b*) *Select*

begin

definition *multisetS* ::

(*'a multiset, 'b*) *Select*

where *multisetS* $\equiv$ ($\downarrow$ *select* = ($\lambda x.$ $\bowtie_{listS\ r}$ (*sorted-list-of-multiset* *x*)))

definition *fsetS* ::

(*'a fset, 'b*) *Select*

where *fsetS* $\equiv$ ($\downarrow$ *select* = ($\lambda x.$ $\bowtie_{listS\ r}$ (*sorted-list-of-fset* *x*)))

end

context

fixes *ra* :: (*'a, 'c*) *Select*

and *rb* :: (*'b, 'c*) *Select*

begin

definition *prodS* ::

((*'a $\times$ 'b*), *'c*) *Select* **where**

prodS $\equiv$ ($\downarrow$ *select* = ($\lambda(a,b).$ $\bowtie_{ra\ a} @ \bowtie_{rb\ b}$)))

```

end

context
  fixes ra :: ('a::linorder,'c) Select
  and rb :: ('b,'c) Select
begin

definition fmapS::
  (('a, 'b) fmap, 'c) Select where
  fmapS ≡ (select = (λfm. ⌊listS (prodS ra rb) (sorted-list-of-fmap fm)⌋))

end

definition noSelect :: ('a, 'b) Select
  where noSelect ≡ (select = (λx. []))

lemma noSelectEmpty[simp]: ⋀x. ⌊noSelect x = []
  by (simp add: noSelect-def)

```

8.2 Constrain

A record to constrain an (output) element given a list of (input) values. `constrain` returns a Boolean value that states whether the constraint holds. The naming-convention to support automation is *your_typeC*, e.g. for `int`: `intC`

```

record ('b,'c) Constrain = constrain::'b list ⇒ 'c ⇒ bool(infixl ⋈1 70)

```

We provide a few generalized (data) structures that extend any definition for a *Constrain*.

```

context
  fixes r :: ('b,'c) Constrain
begin

definition listC::
  ('b, 'c list) Constrain
  where listC ≡ (constrain = (λip op. list-all (λx. ip ⋈r x) op))

definition treeC ::
  ('b, 'c tree) Constrain
  where treeC ≡ (constrain = (λip op. ip ⋈listC (inorder op)))

end

context
  fixes r :: ('b,'c::linorder) Constrain
begin

definition multisetC ::

```

```

('b, 'c multiset) Constrain
where multisetC  $\equiv$  ( $\llbracket$ constrain = ( $\lambda ip\ op.\ ip \triangleright_{listC\ r} (sorted\text{-}list\text{-}of\text{-}multiset\ op)$ ) $\rrbracket$ )

definition fsetC ::
('b, 'c fset) Constrain
where fsetC  $\equiv$  ( $\llbracket$ constrain = ( $\lambda ip\ op.\ ip \triangleright_{listC\ r} (sorted\text{-}list\text{-}of\text{-}fset\ op)$ ) $\rrbracket$ )

end

context
fixes ra :: ('b, 'c) Constrain
and rb :: ('b, 'd) Constrain
begin

definition prodC ::
('b, 'c  $\times$  'd) Constrain where
prodC  $\equiv$  ( $\llbracket$ constrain = ( $\lambda ip\ (opa, opb).\ ip \triangleright_{ra} opa \wedge ip \triangleright_{rb} opb$ ) $\rrbracket$ )

end

context
fixes ra :: ('b, 'c::linorder) Constrain
and rb :: ('b, 'd) Constrain
begin

definition fmapC ::
('b, ('c, 'd) fmap) Constrain where
fmapC  $\equiv$  ( $\llbracket$ constrain = ( $\lambda ip\ op.\ ip \triangleright_{listC\ (prodC\ ra\ rb)} (sorted\text{-}list\text{-}of\text{-}fmap\ op)$ ) $\rrbracket$ )

end

definition noConstrain :: ('b, 'c) Constrain
where noConstrain  $\equiv$  ( $\llbracket$ constrain = ( $\lambda ip\ op.\ True$ ) $\rrbracket$ )

lemma noConstrainTrue[simp]:  $\bigwedge ip\ op.\ ip \triangleright_{noConstrain} op = True$ 
by (simp add: noConstrain-def)

```

8.3 restrict

```

locale Restrictive-Comp =
fixes sel :: ('a, 'b) Select
and con :: ('b, 'c) Constrain
begin

```

Applied to an adversary, *restrict* itself becomes a *restricted* adversary that returns the given adversary's output if and only if it meets the constraints, otherwise it fails. The constraints are primarily characterized by the constrain function in con (select in sel is essentially pre-processing).

definition *restrict* :: ('a ⇒ 'c spmf) ⇒ 'a ⇒ 'c spmf **where**
restrict *A* *arg* ≡ *do* {
res ← *A* *arg*;
 -::unit ← *assert-spmf* ((⊢_{sel} *arg*) ▷_{con} *res*);
return-spmf *res*
}

end

8.4 Examples

locale *examples*

begin

examples for select

definition *intS* :: (int, int) *Select*
where *intS* ≡ (λ*select* = (λ*x*. [*x*]))

lemma ⊢_{intS} (1::int) = [1::int]
unfolding *intS-def*
by *simp*

lemma ⊢_{listS intS} [1] = [1]
by(*simp add: listS-def intS-def*)

lemma ⊢_{listS (listS intS)} [[1]] = [1]
unfolding *listS-def intS-def*
by *fastforce*

lemma ⊢_{prodS (listS intS) intS} ([1], 0) = [1, 0]
unfolding *prodS-def listS-def intS-def*
by *fastforce*

examples for constrain

definition *intC* :: (int, int) *Constrain*
where *intC* ≡ (λ*constrain* = (λ*ip op*. *sum-list ip = op*))

lemma [0, 1, 2] ▷_{intC} 3
by (*simp add: intC-def*)

lemma [0, 1, 2] ▷_{listC intC} [3, 3, 3]
unfolding *listC-def intC-def*
by *simp*

lemma [0, 1, 2] ▷_{prodC (listC intC) intC} ([3, 3, 3], 3)
unfolding *listC-def intC-def prodC-def*
by *fastforce*

end

```

end
theory Algebraic-Group-Model
  imports CryptHOL.CryptHOL Restrictive-Comp
  keywords
    lift-to-algebraicT :: thy-decl
begin

```

9 The Algebraic Group Model

This theory extends CryptHOL for the Algebraic Group Model according to Fuchsbauer, Kiltz, and Loss’ “The Algebraic Group Model and its Applications” [6].

Adversaries in CryptHOL are modelled as uninitialized parameters (arbitrary functions), thus we cannot ensure that the adversary algorithm itself is algebraic. Instead we enforce the algebraic rules on the outputs of the adversary, counting rule breaking as losing the game. Hence, every adversary with non-negligible advantage has to be an algebraic algorithm.

We formalize this relaxed notion of an algebraic algorithm as a sub case of Restrictive_Comp, i.e. we define the AGM in terms of a RCM.

We provide group specific records groupS and groupC for Select and constrain. Furthermore, we define some automation and sanity check functions in Isabelle/ML

```

context cyclic-group
begin

```

group elements are the single important type we want to select.

```

definition groupS :: ('a,'a) Select
  where groupS  $\equiv$  ( $\text{select} = (\lambda x. [x])$ )

```

The above definitions can be composed with the existing data structure extensions from Restrictive_Comp and should cover a wide range of possible inputs. In case some case is missing, the Restrictive_Comp records can be extended to more data structures (see Restrictive_Comp for examples) and this locale can be extended for more atomic type definitions.

check the rules of an algebraic algorithm i.e. given the elements g, a group element, and the vector $\text{vec} = [c_0, \dots, c_n]$ from the algorithm ensure that $g = s_0 [\wedge] c_0 \otimes \dots \otimes s_n [\wedge] c_n$, where $[s_0, \dots, s_n] = \text{seen}$ are the group values the algorithm was supplied with.

```

definition constrain-grp :: 'a list  $\Rightarrow$  ('a  $\times$  int list)  $\Rightarrow$  bool
  where constrain-grp seen-vec res  $\equiv$ 
    let (g, c-vec) = res in
    (length seen-vec = length c-vec
      $\wedge$  g = fold ( $\lambda i \text{ acc. acc} \otimes_G \text{seen-vec}!i [\wedge]_G (\text{c-vec}!i)$ ) [0.. $\text{length seen-vec}$ ]  $\mathbf{1}_G$ )

```

definition $groupC :: ('a, ('a \times int\ list))\ Constrain$
where $groupC \equiv (\llbracket constrain = (\lambda ip\ op.\ constrain-grp\ ip\ op) \rrbracket)$

group values that follow the rules of an algebraic algorithm are actually in the group

lemma *constrain-grp-is-in-carrier*:

assumes $\forall g \in set\ seen-vec.\ g \in carrier\ G$

and *constrain-grp seen-vec* ($g, c-vec$)

shows $g \in carrier\ G$

proof –

have $g = fold\ (\lambda\ i\ acc.\ acc \otimes seen-vec!i\ [\neg]\ (c-vec!i))\ [0..<length\ seen-vec]\ \mathbf{1}$

using *assms(2) constrain-grp-def* **by** *auto*

also have $length\ seen-vec = length\ c-vec$

using *assms(2) constrain-grp-def* **by** *fastforce*

then have $fold\ (\lambda\ i\ acc.\ acc \otimes seen-vec!i\ [\neg]\ (c-vec!i))\ [0..<length\ seen-vec]\ \mathbf{1} \in carrier\ G$

using *assms(1)*

proof (*induct seen-vec c-vec rule: rev-induct2*)

case ($_4\ x\ xs\ y\ ys$)

then have *fold-xs-carrier*: $fold\ (\lambda i\ acc.\ acc \otimes xs!\ i\ [\neg]\ ys!\ i)\ [0..<length\ xs]$

$\mathbf{1} \in carrier\ G$

by *fastforce*

moreover have $x\ [\neg]\ y \in carrier\ G$

by (*simp add: _4(3)*)

moreover have $fold\ (\lambda i\ acc.\ acc \otimes (xs\ @\ [x])!\ i\ [\neg]\ (ys\ @\ [y])!\ i)\ [0..<length\ (xs\ @\ [x])]\ \mathbf{1}$

$= (fold\ (\lambda i\ acc.\ acc \otimes xs!\ i\ [\neg]\ ys!\ i)\ [0..<length\ xs]\ \mathbf{1}) \otimes x\ [\neg]\ y$

proof –

let $?fyys = \lambda xs::'a\ list.\ (\lambda i\ acc.\ acc \otimes xs!\ i\ [\neg]\ (ys\ @\ [y])!\ i)$

have $fold\ (\lambda i\ acc.\ acc \otimes (xs\ @\ [x])!\ i\ [\neg]\ (ys\ @\ [y])!\ i)\ [0..<length\ (xs\ @\ [x])]\ \mathbf{1}$

$= fold\ (?fyys\ (xs\ @\ [x]))\ [0..<length\ (xs\ @\ [x])]\ \mathbf{1}$ **by** *blast*

moreover have $\dots = (fold\ (?fyys\ (xs\ @\ [x]))\ [length\ (xs\ @\ [x]) - 1] \circ fold\ (?fyys\ (xs\ @\ [x]))\ [0..<length\ (xs)])\ \mathbf{1}$

by *auto*

moreover have $\dots = ((\lambda acc.\ acc \otimes x\ [\neg]\ y) \circ fold\ (?fyys\ (xs\ @\ [x]))\ [0..<length\ (xs)])\ \mathbf{1}$

by (*smt (verit, del-insts) _4(2) One-nat-def add-Suc-right append.right-neutral diff-Suc-Suc diff-zero fold.simps(1) fold-Cons id-comp*

length-append list.size(3,4) nth-append-length o-apply)

moreover have $\dots = (fold\ (?fyys\ (xs\ @\ [x]))\ [0..<length\ (xs)]\ \mathbf{1}) \otimes x\ [\neg]\ y$

by *force*

moreover have $\dots = (fold\ (\lambda i\ acc.\ acc \otimes xs!\ i\ [\neg]\ ys!\ i)\ [0..<length\ (xs)]\ \mathbf{1}) \otimes x\ [\neg]\ y$

proof –

have $fold\ (?fyys\ (xs\ @\ [x]))\ [0..<length\ (xs)]\ \mathbf{1}$

$= fold\ (\lambda i\ acc.\ acc \otimes xs!\ i\ [\neg]\ ys!\ i)\ [0..<length\ xs]\ \mathbf{1}$

proof(*rule fold-cong*)

```

    fix i
    assume i ∈ set [0..<length xs]
    then have xs-le-xs: i < length xs
      by force
    then have (xs @ [x]) ! i = xs ! i
      using nth-append-left by blast
    moreover have (ys @ [y]) ! i = ys ! i
      using 4(2) nth-append-left xs-le-xs by auto
    ultimately show (λacc. acc ⊗ (xs @ [x]) ! i [⌈] (ys @ [y]) ! i) = (λacc.
acc ⊗ xs ! i [⌈] ys ! i)
      by presburger
    qed force+
    then show ?thesis by presburger
  qed
  ultimately show ?thesis by argo
  qed
  ultimately show ?case
    by auto
  qed force+
  finally show g ∈ carrier G by blast
qed

```

end

```

locale Algebraic-Algorithm = G : cyclic-group G
  for G :: ('a, 'b) cyclic-group-scheme (structure)
  +
  fixes sel :: ('x, 'a) Select
  and con :: ('a, 'y) Constrain

```

sublocale Algebraic-Algorithm ⊆ Restrictive-Comp sel con .

```

locale algebraic-algorithm-examples = cyclic-group
begin

```

To obtain an algebraic algorithm one needs to simply instantiate the restrictive_comp locale with the record composition that one needs and apply the non-algebraic algorithm to the obtained restrictive_comp.restrict.

The trivial example of only one group element as in and output.

For simplicity let the adversary be id

```

definition A-id :: 'a ⇒ ('a × int list) spmf
  where A-id = (λx. return-spmf (x, [1::int]))

```

```

interpretation Algebraic-Algorithm G groupS groupC
  by (unfold-locales)

```

lemma

```

assumes  $g \in \text{carrier } G$ 
shows  $\text{restrict } \mathcal{A}\text{-id } g$ 
  =  $\mathcal{A}\text{-id } g$ 
unfolding  $\mathcal{A}\text{-id-def Restrictive-Comp.restrict-def}$ 
unfolding  $\text{groupS-def groupC-def constrain-grp-def}$ 
by ( $\text{simp add: assms}$ )

```

Now the same for a list of input elements and and output elements

```

definition  $\mathcal{A}\text{-list-fst} :: 'a \text{ list} \Rightarrow ('a \times \text{int list}) \text{ list } \text{pmf}$ 
  where  $\mathcal{A}\text{-list-fst} = (\lambda x. \text{return-pmf } (\text{map } (\lambda -. (x!0, [1::\text{int}, 0])) x))$ 

```

```

interpretation  $\text{list-id: Restrictive-Comp } (\text{listS groupS}) \text{ listC groupC} .$ 

```

lemma

```

assumes  $g1 \in \text{carrier } G \wedge g2 \in \text{carrier } G \wedge g3 \in \text{carrier } G$ 
shows  $\text{list-id.restrict } \mathcal{A}\text{-list-fst } [g1, g2, g3]$ 
  =  $\mathcal{A}\text{-list-fst } [g1, g2, g3]$ 
unfolding  $\mathcal{A}\text{-list-fst-def Restrictive-Comp.restrict-def}$ 
unfolding  $\text{listS-def groupS-def listC-def groupC-def}$ 
unfolding  $\text{constrain-grp-def}$ 
by ( $\text{simp add: assms}$ )

```

We define some useful ML functions for the AGM.

`lift_to_algebraicT` operates on the type level, lifting standard model function types into their corresponding type as an algebraic algorithm. I.e. extend the outputs that are group elements with a vector.

This takes any algorithm/function type and lifts it to the algebraic algorithm equivalent type. For examples take a look at the end of this file.

ML $\langle$

```

  local
    (*apply function f to the codomain of a function*)
     $\text{fun } \text{rcodomain-transf } f \text{ (Type (fun, [T, U]))} = (\text{Type (fun, [T, rcodomain-transf } f \text{ U]})$ 
    |  $\text{rcodomain-transf } f \text{ T} = f \text{ T};$ 

```

```

    (*adjoin vec to t as a product type if the Type = t*)
     $\text{fun } \text{adjoin } t \text{ vec} = \text{fn } T \Rightarrow \text{if } T = t \text{ then Type (Product-Type.prod, [T, vec])}$ 
    else  $T$ 

```

```

    (*decurry a function*)
     $\text{fun } \text{decurry } (\text{Type (fun, [T1, Type (fun, [T2, T3])])}) =$ 
       $\text{decurry } (\text{Type (fun, [Type (Product-Type.prod, [T1, T2]), T3])})$ 
    |  $\text{decurry } T = T$ 

```

```

     $\text{fun } \text{extract-type-params } t = \text{Term.add-tfree-namesT } t []$ 

```

```

     $\text{fun } \text{lift-to-algebraicT } \text{grpT } \text{vec} = (\text{rcodomain-transf } (\text{Term.map-atyps } (\text{adjoin } \text{grpT } \text{vec}))) \text{ o } \text{decurry}$ 

```

```

in
val - =
  Outer-Syntax.local-theory command-keyword ⟨lift-to-algebraicT⟩ lift to alge-
braic type
  (Parse.typ -- (Parse.term -- (keyword ⟨=>⟩ | -- Parse.binding)) >>
    (fn (alg,(grp,b)) => fn lthy => Local-Theory.raw-theory (fn thy =>
      let
        val algT = Syntax.read-typ lthy alg;
        val grp-desc = Syntax.read-term lthy grp;
        val grpT = Term.fasttype-of grp-desc |> Term.dest-Type-args |> hd;
        val agmT = lift-to-algebraicT grpT @{typ int list} algT;
        val tps = extract-type-params agmT;
        val thy' = Sign.add-type-abbrev lthy (b, tps, agmT) thy
      in thy' end) lthy));
  end;
⟩

```

9.1 Example for the ML function

We define a standard model adversary with group G of type $'a$

type-synonym ($'a'$)adv = $'a'$ list $\Rightarrow 'a' \Rightarrow \text{nat} \Rightarrow ('a' * \text{int} * \text{nat}) \text{ spmf}$

We lift the type into the AGM (its algebraic algorithm pendant):

lift-to-algebraicT ($'a'$) adv $G \Rightarrow \text{algebraic-adv}$

end

end

theory KZG-def

imports Polynomial-Commitment-Schemes Primitives

begin

In this theory we formalize the KZG polynomial commitment scheme as introduced in Kate, Zaverucha, and Goldberg’s “Constant-Size Commitments to Polynomials and Their Applications” [8].

10 KZG function definitions

Define the KZG with functions that match the abstract polynomial commitment scheme and instantiate the KZG as a polynomial commitment scheme.

locale KZG = *math-primitives*

begin

type-synonym trapdoor = *unit*

```

type-synonym 'a' ck = 'a' list
type-synonym 'a' vk = 'a' list
type-synonym 'a' commit = 'a'
type-synonym 'e' argument = 'e' mod-ring
type-synonym 'e' evaluation = 'e' mod-ring
type-synonym 'a' witness = 'a'

```

10.1 Setup

We do not compute the Groups for the bilinear pairing but assume them and compute a uniformly random secret key α and from that the structured reference string (srs)/public key $PK = (g, g^{\alpha}, \dots, g^{\alpha^t})$. Setup is a trusted Setup function, which generates the shared public key for both parties (prover and verifier).

definition $Setup :: ('e \text{ mod-ring} \times 'a \text{ list}) \text{ spmf}$
where
 $Setup = do \{$
 $x :: nat \leftarrow \text{sample-uniform } (order \ G_p);$
 $let \alpha :: 'e \text{ mod-ring} = of-int-mod-ring \ (int \ x) \ in$
 $return-spmf \ (\alpha, \ map \ (\lambda t. \ \mathbf{g}_{G_p}^{\alpha^t} \ (\alpha^t)) \ [0..<max-deg+1])$
 $\}$

This function computes $g^{\varphi(\alpha)}$, given the by Setup generated public key. (α being the from Setup generated private key)

The function is basically a Prod of $public \ key!i \wedge coeff \ \varphi \ i$, which computes $g^{\varphi(a)}$, given the public key: $\prod [0 \dots degree \ \varphi]. \ PK!i \wedge coeff \ \varphi \ i = \prod [0 \dots degree \ \varphi]. \ g^{\alpha^i \wedge coeff \ \varphi \ i} = \prod [0 \dots degree \ \varphi]. \ g^{coeff \ \varphi \ i * \alpha^i} = g^{\sum [0 \dots degree \ \varphi]. \ coeff \ \varphi \ i * \alpha^i} = g^{\varphi(\alpha)}$

fun $g\text{-pow-PK-Prod} :: 'a \text{ list} \Rightarrow 'e \text{ mod-ring poly} \Rightarrow 'a$ **where**
 $g\text{-pow-PK-Prod} \ PK \ \varphi = fold \ (\lambda i \ acc. \ acc \otimes_{G_p} \ PK!i \wedge_{G_p} \ (poly.coeff \ \varphi \ i)) \ [0..<Suc \ (degree \ \varphi)] \ \mathbf{1}_{G_p}$

q_coeffs is a accumulator for the fold function. $fold \ coeffs_n$ creates a vertical summation by going through the power_diff_sumr2 and instead of adding the horizontal row, mapping it into a list, which is added onto the previous list of coefficients, hence summing the coefficients vertical in a list. Illustration:

```

0: [0 ] => map (+) 1: [f1 ] => map(+) 2: [f1 + f2*u , f2*x ] => map(+)
3: [f1 + f2*u + f3*u^2 , f2*x+f3*u*x , f3*x^2] ... n: [f1 + ... + fn*u^(n-1)
, ... , f(i-1)*x^i +...+fn*u^((n-1)-i)*x^i , ... , fn*x^(n-1)]

```

Hence the resulting list represents the vertical sum with coefficient i at position $(i-1)$. The correctness proof is to be found in the correctness theory `KZG_correct`.

definition $coeffs_n :: 'e \text{ mod-ring poly} \Rightarrow 'e \text{ mod-ring} \Rightarrow 'e \text{ mod-ring list} \Rightarrow nat \Rightarrow 'e \text{ mod-ring list}$

where $\text{coeffs-n } \varphi \ u = (\lambda q\text{-coeffs}. \lambda n. \text{map } (\lambda(i::\text{nat}). (\text{nth-default } 0 \ q\text{-coeffs } i + \text{poly.coeff } \varphi \ n * u \wedge (n - \text{Suc } i))) \ [0..<n])$

The objective of this function is to extract ψ in $\varphi \ x - \varphi \ u = (x-u) * \psi$
 x Idea: the polynomial φ can be represented as a sum, hence: $\varphi \ x - \varphi \ u = (\sum_{i \leq \text{degree } \varphi}. \text{coeff } \varphi \ i * x^i) - (\sum_{i \leq \text{degree } \varphi}. \text{coeff } \varphi \ i * x^i) = (x-u)(\sum_{i \leq \text{degree } \varphi}. \text{coeff } \varphi \ i * (\sum_{j < i}. u \wedge (i - \text{Suc } j) * x^j))$ (for the last step see the lemma `power_diff_sumr2`) Hence: $\psi \ x = (\sum_{i \leq \text{degree } \varphi}. \text{coeff } \varphi \ i * (\sum_{j < i}. u \wedge (i - \text{Suc } j) * x^j))$ However, to build a polynomial ψ in Isabelle, we need the individual coefficients for every power of x (i.e. bring the sum into a form of $(\sum_{i \leq \text{degree } \varphi}. \text{coeff-}i * x^i)$ where `coeff_i` is the individual coefficients for every power i of x . This translation is the main-purpose of the extractor function. The key idea is reordering the summation obtained from the `power_diff_sumr2` lemma:

One can imagine the summation of `power_diff_sumr2` to be horizontal, in the sense that it computes the coefficients from 0 to $\text{degree } \varphi = n$, and adds (or could add) to each coefficient in every iteration: 0: $0 + 1: f_1 + 2: f_2 * u + f_2 * x + 3: f_3 * u^2 + f_3 * u * x + f_3 * x^2 \dots n: f_n * u^{(n-1)} + \dots f_n * u^{((n-1)-i)} * x^i \dots + f_n * x^{(n-1)}$ we order it to compute the coefficients one by one (to compute vertical) 1: $0 + f_1 + \dots + f_n * u^{(n-1)} + 2: 0 + f_2 * x + \dots + f_n * u^{((n-1)-1)} * x + \dots n: 0 + f_n * x^{(n-1)}$

In formulas: $(\sum_{i \leq \text{degree } \varphi}. \text{coeff } \varphi \ i * (\sum_{j < i}. u \wedge (i - \text{Suc } j) * x^j)) = (\sum_{i \leq \text{degree } \varphi}. (\sum_{j \in \{i < .. < \text{Suc } (\text{degree } \varphi)\}}. \text{coeff } \varphi \ j * u \wedge (j - \text{Suc } i)) * x^i)$

definition $\psi\text{-of} :: 'e \text{ mod-ring poly} \Rightarrow 'e \text{ mod-ring} \Rightarrow 'e \text{ mod-ring poly}$

where $\psi\text{-of } \varphi \ u = (\text{let}$
 $\text{ } \varphi\text{-coeffs} = \text{foldl } (\text{coeffs-n } \varphi \ u) \ [] \ [0..<\text{Suc } (\text{degree } \varphi)] \text{ --- coefficients of } \psi$
 $\text{in Poly } \varphi\text{-coeffs}) \text{ --- } \psi$

a wrapper around Setup that hands the srs to both parties

definition $\text{key-gen} :: ('a \text{ ck} \times 'a \text{ vk}) \text{ spmf}$

where $\text{key-gen} = \text{do } \{$
 $\text{ } (\alpha, PK) \leftarrow \text{Setup};$
 $\text{ } \text{return-spmf } (PK, PK)$
 $\text{ } \}$

the KZG functions follow the description in section 3.2 of the KZG paper, but mirror the structure and naming of the abstract polynomial commitment scheme.

definition $\text{commit} :: 'a \text{ ck} \Rightarrow 'e \text{ mod-ring poly} \Rightarrow ('a \text{ commit} \times \text{trapdoor}) \text{ spmf}$

where $\text{commit } PK \ \varphi = \text{return-spmf } (g\text{-pow-PK-Prod } PK \ \varphi, ()) \text{ --- } g^{\wedge} \varphi(\alpha)$

definition $\text{verify-poly} :: 'a \text{ vk} \Rightarrow 'e \text{ mod-ring poly} \Rightarrow 'a \text{ commit} \Rightarrow \text{trapdoor} \Rightarrow \text{bool}$

where $\text{verify-poly } PK \ \varphi \ C \ \text{td} = (C = g\text{-pow-PK-Prod } PK \ \varphi) \text{ --- } C = g^{\wedge} \varphi(\alpha)$

This is the createWitness function in the KZG paper

definition *Eval* :: 'a ck $\Rightarrow$ trapdoor $\Rightarrow$ 'e mod-ring poly $\Rightarrow$ 'e mod-ring $\Rightarrow$ ('e mod-ring $\times$ 'a witness)

where *Eval* PK td φ i = (let $\psi = \psi\text{-of } \varphi$ i $\longrightarrow \psi$ in $\varphi(x) - \varphi(i) = (x-i) * \psi(x)$
in (poly φ i, g-pow-PK-Prod PK ψ) $\longrightarrow (\varphi(i), g^{\wedge} \psi(\alpha))$)

definition *verify-eval* :: 'a vk $\Rightarrow$ 'a commit $\Rightarrow$ 'e mod-ring $\Rightarrow$ ('e mod-ring $\times$ 'a witness) $\Rightarrow$ bool

where *verify-eval* PK C i val = (let (eval,w) = val
in (e w (PK!1 $\div_{G_p}$ ($\mathbf{g}_{G_p}^{\wedge_{G_p}} i$)) $\otimes_{G_T}$ ((e $\mathbf{g}_{G_p}$ $\mathbf{g}_{G_p}$) $\wedge_{G_T}$ eval) = e C $\mathbf{g}_{G_p}$))
 $\longrightarrow e(g^{\wedge} \psi(\alpha), g^{\wedge} \alpha / g^{\wedge} i) \otimes e(g,g)^{\wedge} \varphi(i) = e(C, g)$)

definition *valid-poly*::'e mod-ring poly $\Rightarrow$ bool

where *valid-poly* φ = (degree $\varphi \leq \text{max-deg}$)

definition *valid-argument* :: 'e argument $\Rightarrow$ bool

where *valid-argument* - = True

definition *valid-eval*::('e mod-ring $\times$ 'a witness) $\Rightarrow$ bool

where *valid-eval* val = (let (-,w) = val in w $\in$ carrier G_p)

the KZG is a polynomial commitment scheme

sublocale *abstract-polynomial-commitment-scheme key-gen commit verify-poly Eval*
verify-eval

valid-poly valid-argument valid-eval .

end

end

theory *KZG-correct*

imports *KZG-def*

begin

11 Correctness of the KZG

locale *KZG-PCS-correct* = *KZG*

begin

11.0.1 Helping lemmas for the computation of ψ

Helping lemmas for the computation of ψ (function $\psi\text{-of}$) in $\varphi(x) - \varphi(c) = (x-c) * \psi(x)$, which is used to compute ψ in CreateWitness.

lemma *coeffs-n-length[simp]*: length (coeffs-n φ u q-co n) = n

unfolding *coeffs-n-def* **by** fastforce

lemma *coeffs-n-add-nth[simp]*: $\forall i < n. \text{coeffs-n } \varphi \text{ u l n ! } i = \text{nth-default } 0 \text{ l } i + \text{poly.coeff } \varphi \text{ n } * \text{u}^{\wedge} (n - \text{Suc } i)$

unfolding *coeffs-n-def* **by** auto

lemma ψ -coeffs-length: $\text{length } (\text{foldl } (\text{coeffs-n } \varphi \ u) \ [] \ [0..<\text{Suc } n]) = n$
by *auto*

lemma *sum-split*: $m \leq n \implies (\sum i \leq n. f \ i) = (\sum i \leq m. f \ i) + (\sum i \in \{m < .. < \text{Suc } n\}. f \ i)$
proof –
assume $m \leq n$
then show $(\sum i \leq n. f \ i) = (\sum i \leq m. f \ i) + (\sum i \in \{m < .. < \text{Suc } n\}. f \ i)$
proof (*induction n arbitrary: m*)
case 0
then show ?case
using *greaterThanLessThan-upt* **by** *fastforce*
next
case (*Suc n*)
then show ?case
using *greaterThanLessThan-upt*
by (*metis Suc-le-mono atLeast0AtMost atLeastLessThanSuc-atLeastAtMost atLeastSucLessThan-greaterThanLessThan less-eq-nat.simps(1) sum.atLeastLessThan-concat*)
qed
qed

state that the computed polynomial ψ , is of degree less equal to φ .

lemma *degree-q-le-φ*: $\text{degree } (\psi\text{-of } \varphi \ u) \leq \text{degree } \varphi$
unfolding *ψ-of-def*
by (*metis degree-Poly ψ-coeffs-length*)

This lemma is essentially resorting the summation according to the idea given in KZG_def above the CreateWitness definition.

The left-hand side computes the coefficients horizontal, in the sense that it computes the coefficients from 0 to degree $\varphi = n$, and adds (or could add) to each coefficient in every iteration: 0: 0 + 1: f_1 + 2: $f_2^*u + f_2^*x$ + 3: $f_3^*u^2 + f_3^*u^*x + f_3^*x^2$... n: $f_n^*u^{(n-1)} + \dots f_n^*u^{((n-1)-i)}*x^i \dots + f_n^*x^{(n-1)}$

The right-hand side captures the vertical summation in a sum in the sum. Hence computing the coefficient in the inner sum first, before multiplying it with x^i . Illustrated: 0: $(0 + f_1 + f_2^*u + f_3^*u^2 + \dots + f_n^*u^{(n-1)})^*x^0$ + 1: $(0 + 0 + f_2 + f_3^*u + \dots + f_n^*u^{(n-2)})^*x^1$... n: $(0 + 0 + 0 + 0 + \dots + f_n)^*x^{(n-1)}$

lemma *sum-horiz-to-vert*: $n \leq \text{degree } (\varphi :: 'e \text{ mod-ring poly}) \implies$
 $(\sum i \leq n. \text{poly.coeff } \varphi \ i * (\sum j < i. u^{(i - \text{Suc } j)} * x^{(j)}))$
 $= (\sum i \leq n. (\sum j \in \{i < .. < \text{Suc } n\}. \text{poly.coeff } \varphi \ j * u^{(j - \text{Suc } i)}) * x^{(i)})$
proof (*induction n arbitrary: φ*)
case 0
have $(\sum i \leq 0. \text{poly.coeff } \varphi \ i * (\sum j < i. u^{(i - \text{Suc } j)} * x^{(j)})) = 0$ **by** *fastforce*
also have $(\sum i \leq 0. (\sum j \in \{i < .. < \text{Suc } 0\}. \text{poly.coeff } \varphi \ j * u^{(j - \text{Suc } i)}) * x^{(i)}) = 0$

```

    by (simp add: greaterThanLessThan-upt)
    ultimately show ?case by argo
next
case (Suc n)
have (∑ i ≤ Suc n. poly.coeff φ i * (∑ j < i. u ^ (i - Suc j) * x ^ j))
  = (∑ i ≤ n. poly.coeff φ i * (∑ j < i. u ^ (i - Suc j) * x ^ j))
  + poly.coeff φ (Suc n) * (∑ j < (Suc n). u ^ (Suc n - Suc j) * x ^ j) by auto
also have ... = (∑ i ≤ n. (∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc i))
  * x ^ i)
  + poly.coeff φ (Suc n) * (∑ j < (Suc n). u ^ (Suc n - Suc j) * x ^ j)
  using Suc.IH Suc.premis Suc-leD by presburger
also have ... = (∑ i ≤ n. (∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc i)) *
  x ^ i)
  + (∑ j < (Suc n). poly.coeff φ (Suc n) * u ^ (Suc n - Suc j) * x ^ j)
  by (metis (mono-tags, lifting) mult.assoc mult-hom.hom-sum sum.cong)
also have ... = (∑ i < Suc n. (∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc
  i)) * x ^ i)
  + (∑ j < Suc n. poly.coeff φ (Suc n) * u ^ (Suc n - Suc j) * x ^ j)
  using lessThan-Suc-atMost by presburger
also have ... = (∑ i < Suc n. (∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc
  i)) * x ^ i)
  + poly.coeff φ (Suc n) * u ^ (Suc n - Suc i) * x ^ i)
  by (simp add: sum.distrib)
also have ... = (∑ i < Suc n. ((∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc
  i)) + poly.coeff φ (Suc n) * u ^ (Suc n - Suc i)) * x ^ i)
  by (simp add: distrib-left mult.commute)
also have ... = (∑ i < Suc n. (∑ j ∈ {i < .. < Suc (Suc n)}. poly.coeff φ j * u ^ (j -
  Suc i)) * x ^ i)
proof -
  have ∀ (i :: nat) < Suc n. ((∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc i))
  + poly.coeff φ (Suc n) * u ^ (Suc n - Suc i))
  = (∑ j ∈ {i < .. < Suc (Suc n)}. poly.coeff φ j * u ^ (j - Suc i))
proof
  fix i
  show i < Suc n ⟶
    (∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc i)) + poly.coeff φ (Suc
  n) * u ^ (Suc n - Suc i) =
    (∑ j ∈ {i < .. < Suc (Suc n)}. poly.coeff φ j * u ^ (j - Suc i))
proof
  let ?f = (λj. poly.coeff φ j * u ^ (j - Suc i))
  assume asmp: i < Suc n
  then show (∑ j ∈ {i < .. < Suc n}. ?f j) + ?f (Suc n) = (∑ j ∈ {i < .. < Suc (Suc
  n)}. ?f j)
  by (smt (verit) atLeastSucLessThan-greaterThanLessThan not-less-eq sum.op-ivl-Suc)
qed
qed
then show (∑ i < Suc n. ((∑ j ∈ {i < .. < Suc n}. poly.coeff φ j * u ^ (j - Suc
  i)) + poly.coeff φ (Suc n) * u ^ (Suc n - Suc i)) * x ^ i) =
  (∑ i < Suc n. (∑ j ∈ {i < .. < Suc (Suc n)}. poly.coeff φ j * u ^ (j - Suc i)) * x ^

```

i)
by *fastforce*
qed
also have $\dots = (\sum i \leq \text{Suc } n. (\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)) * x ^ i)$
proof –
have $(\sum j \in \{\text{Suc } n < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } (\text{Suc } n))) * x ^ (\text{Suc } n) = 0$
by (*simp add: greaterThanLessThan-upt*)
then have $(\sum i < \text{Suc } n. (\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)) * x ^ i)$
 $= (\sum i < \text{Suc } n. (\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)) * x ^ i)$
 $+ (\sum j \in \{\text{Suc } n < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } (\text{Suc } n))) * x ^ (\text{Suc } n)$
by *force*
also have $\dots = (\sum i \leq \text{Suc } n. (\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)) * x ^ i)$
by (*simp add: lessThan-Suc-atMost*)
ultimately show $(\sum i < \text{Suc } n. (\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)) * x ^ i)$
 $= (\sum i \leq \text{Suc } n. (\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)) * x ^ i)$
by *argo*
qed
ultimately show *?case using Suc.premis Suc-leD by argo*
qed

We now show that the inner sum from the last lemma, which calculates the *i*-th coefficient for ψ , is equal to the *i*-th coefficient calculated from the ψ -of function.

lemma ψ -of-ith-coeff-eq-sum-ith-coeff: $i < n \implies \text{foldl } (\text{coeffs-}n \ \varphi \ u) \ [] \ [0..<\text{Suc } n] ! i$

$$= (\sum j \in \{i < .. < \text{Suc } n\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i))$$

Suc i)

proof (*induction n arbitrary: i*)

case 0

then show *?case by blast*

next

case (*Suc n*)

then show *?case*

proof (*cases i < n*)

case *True*

have $\text{foldl } (\text{coeffs-}n \ \varphi \ u) \ [] \ [0..<\text{Suc } (\text{Suc } n)]$

$= \text{map } (\lambda i. \text{nth-default } 0 \ (\text{foldl } (\text{coeffs-}n \ \varphi \ u) \ [] \ [0..<\text{Suc } n]) \ i + \text{poly.coeff } \varphi (\text{Suc } n) * u ^ (\text{Suc } n - \text{Suc } i))$

$[0..<\text{Suc } n]$

unfolding *coeffs-n-def by force*

then have $\text{foldl } (\text{coeffs-}n \ \varphi \ u) \ [] \ [0..<\text{Suc } (\text{Suc } n)] ! i$

```

    = nth-default 0 (foldl (coeffs-n  $\varphi$  u) [] [0.. $\text{Suc } n$ ]) i + poly.coeff  $\varphi$  (Suc
n) * u ^ (Suc n - Suc i)
    by (metis (lifting) Suc.premss add-0 diff-zero nth-map-upt)
    also have ... = ( $\sum j \in \{i < .. < \text{Suc } n\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)$ ) + poly.coeff
 $\varphi$  (Suc n) * u ^ (Suc n - Suc i)
    using Suc.IH[of i] by (simp add: True nth-default-def)
    also have ... = ( $\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)$ )
    proof -
      have  $\forall x y f. x < y \longrightarrow (\sum j \in \{x < .. < y\}. f j) + f y = (\sum j \in \{x < .. < \text{Suc } y\}. f j)$ 
      by (metis Suc-le-eq atLeastSucLessThan-greaterThanLessThan sum.atLeastLessThan-Suc)
      then show ( $\sum j \in \{i < .. < \text{Suc } n\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)$ ) + poly.coeff
 $\varphi$  (Suc n) * u ^ (Suc n - Suc i) =
      ( $\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)$ ) using Suc.premss
    by blast
  qed
  ultimately show ?thesis by argo
next
case False
then have i-eq-n: i = n using Suc.premss by simp
have foldl (coeffs-n  $\varphi$  u) [] [0.. $\text{Suc } (\text{Suc } n)$ ]
  = coeffs-n  $\varphi$  u (foldl (coeffs-n  $\varphi$  u) [] [0.. $\text{Suc } n$ ]) (Suc n) by simp
also have ... = map ( $\lambda(i::\text{nat}). (\text{nth-default } 0 (\text{foldl } (\text{coeffs-n } \varphi) u) [] [0.. $\text{Suc } n$ ]) i$ 
  + poly.coeff  $\varphi$  (Suc n) * u ^ ((Suc n) - Suc i)) [0.. $\text{Suc } n$ ]
  unfolding coeffs-n-def by blast
ultimately have foldl (coeffs-n  $\varphi$  u) [] [0.. $\text{Suc } (\text{Suc } n)$ ] ! i
  = map ( $\lambda(i::\text{nat}). (\text{nth-default } 0 (\text{foldl } (\text{coeffs-n } \varphi) u) [] [0.. $\text{Suc } n$ ]) i$ 
  + poly.coeff  $\varphi$  (Suc n) * u ^ ((Suc n) - Suc i)) [0.. $\text{Suc } n$ ] ! i
  by argo
also have ... = (nth-default 0 (foldl (coeffs-n  $\varphi$  u) [] [0.. $\text{Suc } n$ ]) i
  + poly.coeff  $\varphi$  (Suc n) * u ^ ((Suc n) - Suc i))
  using Suc.premss calculation by auto
also have ... = poly.coeff  $\varphi$  (Suc n) * u ^ ((Suc n) - Suc i)
  by (simp add: nth-default-eq-dflt-iff i-eq-n)
also have ( $\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)$ )
  = poly.coeff  $\varphi$  (Suc n) * u ^ ((Suc n) - Suc i)
    proof -
      have  $\{i < .. < \text{Suc } (\text{Suc } n)\} = \{\text{Suc } n\}$ 
      proof
        show  $\{i < .. < \text{Suc } (\text{Suc } n)\} \subseteq \{\text{Suc } n\}$ 
        by (simp add: greaterThanLessThan-upt i-eq-n)
        show  $\{\text{Suc } n\} \subseteq \{i < .. < \text{Suc } (\text{Suc } n)\}$ 
        by (simp add: i-eq-n)
      qed
      then show ( $\sum j \in \{i < .. < \text{Suc } (\text{Suc } n)\}. \text{poly.coeff } \varphi j * u ^ (j - \text{Suc } i)$ )
        = poly.coeff  $\varphi$  (Suc n) * u ^ ((Suc n) - Suc i)
        by simp
    qed
  qed

```

ultimately show *?thesis* by argo
qed
qed

We now take together the last few lemmas and definitions and show that ψ -of-poly calculates the correct ψ . With the `sum_horiz_to_vert` lemma, we restructure the left-hand side to calculate the coefficients of ψ before multiplying with x^i . With the ψ -of-ith-coeff-eq-sum-ith-coeff lemma, show the coefficients of the result of `sum_horiz_to_vert` equal to the coefficients calculated by ψ -of-poly and thus showing $\text{poly } (\psi\text{-of-poly } \varphi \ u) \ x$ equal to the result `sum` of `sum_horiz_to_vert`.

lemma $\varphi x\text{-m-}\varphi u\text{-eq-}\varphi x$: $\forall \varphi :: 'e \text{ mod-ring poly. } \text{poly } \varphi \ x - \text{poly } \varphi \ u = (x - u) * \text{poly } (\psi\text{-of } \varphi \ u) \ x$

proof

fix $\varphi :: 'e \text{ mod-ring poly}$
fix $u \ x :: 'e \text{ mod-ring}$
show $\text{poly } \varphi \ x - \text{poly } \varphi \ u = (x - u) * \text{poly } (\psi\text{-of } \varphi \ u) \ x$
proof –
let $?q\text{-coeffs} = \text{foldl } (\text{coeffs-n } \varphi \ u) \ [] \ [0..<\text{Suc } (\text{degree } \varphi)]$
let $?q\text{-dirty} = (\lambda x. (\sum i \leq \text{degree } \varphi. \text{poly.coeff } \varphi \ i * (\sum j < i. u^{\wedge}(i - \text{Suc } j) * x^{\wedge}j)))$
let $?q\text{-vert} = (\lambda x. (\sum i \leq \text{degree } \varphi. (\sum j \in \{i < .. < \text{Suc } (\text{degree } \varphi)\}. \text{poly.coeff } \varphi \ j * u^{\wedge}(j - \text{Suc } i)) * x^{\wedge}i))$
let $?q = \psi\text{-of } \varphi \ u$

have $(\sum i \leq \text{degree } \varphi. \text{poly.coeff } \varphi \ i * x^{\wedge}i) - (\sum i \leq \text{degree } \varphi. \text{poly.coeff } \varphi \ i * u^{\wedge}i)$
= $(\sum i \leq \text{degree } \varphi. \text{poly.coeff } \varphi \ i * (x^{\wedge}i - u^{\wedge}i))$
by (simp add: sum-subtractf right-diff-distrib)
also have $\dots = (\sum i \leq \text{degree } \varphi. (x - u) * \text{poly.coeff } \varphi \ i * (\sum j < i. u^{\wedge}(i - \text{Suc } j) * x^{\wedge}j))$
by (simp add: mult.assoc mult.left-commute power-diff-sumr2)
also have $\dots = (x - u) * (?q\text{-dirty } x)$
by (metis (mono-tags, lifting) mult.assoc mult-hom.hom-sum sum.cong)
also have $\dots = (x - u) * (?q\text{-vert } x)$ using sum-horiz-to-vert by auto
also have $?q\text{-vert } x = \text{poly } ?q \ x$
proof –

have $(\sum i \leq \text{degree } \varphi. \text{nth-default } 0 \ ?q\text{-coeffs } i * x^{\wedge}i)$
= $(\sum i \leq \text{degree } ?q. \text{nth-default } 0 \ ?q\text{-coeffs } i * x^{\wedge}i)$
proof –
have $\text{degree } ?q \leq \text{degree } \varphi$ by (rule degree-q-le- φ)
also have $\forall n. n \geq \text{degree } ?q \wedge n \leq \text{degree } \varphi \longrightarrow (\sum i \leq n. \text{nth-default } 0 \ ?q\text{-coeffs } i * x^{\wedge}i)$
= $(\sum i \leq \text{degree } ?q. \text{nth-default } 0 \ ?q\text{-coeffs } i * x^{\wedge}i)$

proof
fix n

```

show  $n \geq \text{degree } ?q \wedge n \leq \text{degree } \varphi \longrightarrow (\sum i \leq n. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i)$ 
 $= (\sum i \leq \text{degree } ?q. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i)$ 

proof
  let  $?f = \text{nth-default } 0 \text{ } ?q\text{-coeffs}$ 
  assume  $\text{asmp}: n \geq \text{degree } ?q \wedge n \leq \text{degree } \varphi$ 
  have  $\forall i > \text{degree } ?q. ?f \ i = 0$ 
  using  $\text{coeff-eq-0 coeffs-n-def}$ 
  by  $(\text{metis } \psi\text{-of-def coeff-Poly-eq})$ 
  then have  $(\sum i \in \{\text{degree } ?q < .. < \text{Suc } n\}. ?f \ i * x^i) = 0$ 
  by  $\text{fastforce}$ 
  also have  $(\sum i \leq n. ?f \ i * x^i) = (\sum i \leq \text{degree } ?q. ?f \ i * x^i) + (\sum i \in \{\text{degree } ?q < .. < \text{Suc } n\}. ?f \ i * x^i)$ 
  using  $\text{sum-split asmp by blast}$ 
  ultimately show  $(\sum i \leq n. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i) = (\sum i \leq \text{degree } ?q. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i)$ 
  using  $\text{asmp by auto}$ 
qed
qed
ultimately show  $(\sum i \leq \text{degree } \varphi. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i) = (\sum i \leq \text{degree } ?q. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i)$ 
by  $\text{blast}$ 
qed
also have  $?q\text{-vert } x = (\sum i \leq \text{degree } \varphi. \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^i)$ 
proof –
  have  $\forall i \leq \text{degree } \varphi. (\sum j \in \{i < .. < \text{Suc } (\text{degree } \varphi)\}. \text{poly.coeff } \varphi \ j * u^{j - \text{Suc } i}) = \text{nth-default } 0 \text{ } ?q\text{-coeffs } i$ 
proof
  fix  $i$ 
  show  $i \leq \text{degree } \varphi \longrightarrow (\sum j \in \{i < .. < \text{Suc } (\text{degree } \varphi)\}. \text{poly.coeff } \varphi \ j * u^{j - \text{Suc } i}) = \text{nth-default } 0 \text{ } ?q\text{-coeffs } i$ 
  proof
    assume  $\text{asmp}: i \leq \text{degree } \varphi$ 
    then show  $(\sum j \in \{i < .. < \text{Suc } (\text{degree } \varphi)\}. \text{poly.coeff } \varphi \ j * u^{j - \text{Suc } i}) = \text{nth-default } 0 \text{ } ?q\text{-coeffs } i$ 
    proof  $(\text{cases } i < \text{degree } \varphi)$ 
      case  $\text{True}$ 
      have  $\text{length } ?q\text{-coeffs} = \text{degree } \varphi$  by  $\text{simp}$ 
      then have  $\text{nth-default } 0 \text{ } ?q\text{-coeffs } i = ?q\text{-coeffs } ! \ i$ 
      by  $(\text{simp add: True nth-default-nth})$ 
      then show  $?thesis$  using  $\text{True } \psi\text{-of-ith-coeff-eq-sum-ith-coeff by presburger}$ 
    next
    case  $\text{False}$ 

```

then have $i = \text{degree } \varphi$ **using** *asmp* **by** *fastforce*
 have $\text{length } ?q\text{-coeffs} = \text{degree } \varphi$ **by** *simp*
 then have $\text{nth-default } 0 \text{ } ?q\text{-coeffs } i = 0$
 by (*simp add: ‹i = degree ‹var› nth-default-beyond›*)
 also have $(\sum_{j \in \{i < .. < \text{Suc } (\text{degree } \varphi)\}} \text{poly.coeff } \varphi \ j * u^{\wedge} (j - \text{Suc } i))$
 $= 0$ **using** *False greaterThanLessThan-upt* **by** *auto*
 ultimately show *?thesis* **by** *argo*
 qed
 qed
 qed
 then show $?q\text{-vert } x = (\sum_{i \leq \text{degree } \varphi} \text{nth-default } 0 \text{ } ?q\text{-coeffs } i * x^{\wedge} i)$
 by *force*
 qed
 ultimately show $?q\text{-vert } x = \text{poly } ?q \ x$
 by (*metis (no-types, lifting) ‹psi-of-def coeff-Poly-eq poly-altdef sum.cong›*)
 qed
 ultimately show $\text{poly } \varphi \ x - \text{poly } \varphi \ u = (x - u) * \text{poly } (\psi\text{-of } \varphi \ u) \ x$
 by (*simp add: poly-altdef*)
 qed
 qed

Taking the result to the bilinear function. We know $\varphi(x) - \varphi(u) = (x - u)\psi(x)$ from the previous corollary, now we show the equality is also valid with the bilinear function e.

lemma *eq-on-e*: $(e (\mathbf{g}_{G_p}^{\wedge_{G_p}} (\text{poly } (\psi\text{-of } \varphi \ i) \ \alpha))) (\mathbf{g}_{G_p}^{\wedge_{G_p}} \alpha \div_{G_p} \mathbf{g}_{G_p}^{\wedge_{G_p}} i))$

$$\begin{aligned}
 & \otimes_{G_T} (e \mathbf{g}_{G_p} \mathbf{g}_{G_p})^{\wedge_{G_T}} (\text{poly } \varphi \ i) \\
 & = e (\mathbf{g}_{G_p}^{\wedge_{G_p}} (\text{poly } \varphi \ \alpha)) \mathbf{g}_{G_p}
 \end{aligned}$$

proof –

have *e-in-carrier*: $(e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \in \text{carrier } G_T$ **using** *e-symmetric* **by** *blast*
 have $e (\mathbf{g}_{G_p}^{\wedge_{G_p}} \text{poly } (\psi\text{-of } \varphi \ i) \ \alpha) (\mathbf{g}_{G_p}^{\wedge_{G_p}} \alpha \div_{G_p} \mathbf{g}_{G_p}^{\wedge_{G_p}} i) \otimes_{G_T} e \mathbf{g}_{G_p}$
 $\mathbf{g}_{G_p}^{\wedge_{G_T}} \text{poly } \varphi \ i$
 $= e (\mathbf{g}_{G_p}^{\wedge_{G_p}} \text{poly } (\psi\text{-of } \varphi \ i) \ \alpha) (\mathbf{g}_{G_p}^{\wedge_{G_p}} (\alpha - i)) \otimes_{G_T} e \mathbf{g}_{G_p} \mathbf{g}_{G_p}^{\wedge_{G_T}}$
 $\text{poly } \varphi \ i$
 using *mod-ring-pow-mult-inv- G_p* **by** *force*
 also have $\dots = (e \mathbf{g}_{G_p} \mathbf{g}_{G_p})^{\wedge_{G_T}} ((\text{poly } (\psi\text{-of } \varphi \ i) \ \alpha) * (\alpha - i)) \otimes_{G_T} e \mathbf{g}_{G_p}$
 $\mathbf{g}_{G_p}^{\wedge_{G_T}} \text{poly } \varphi \ i$
 using *G_p .generator-closed e-bilinear* **by** *presburger*
 also have $\dots = (e \mathbf{g}_{G_p} \mathbf{g}_{G_p})^{\wedge_{G_T}} ((\text{poly } (\psi\text{-of } \varphi \ i) \ \alpha) * (\alpha - i) + \text{poly } \varphi \ i)$
 using *mod-ring-pow-mult- G_T e-in-carrier* **by** *presburger*
 also have $\dots = (e \mathbf{g}_{G_p} \mathbf{g}_{G_p})^{\wedge_{G_T}} (\text{poly } \varphi \ \alpha)$
 by (*metis Groups.mult-ac(2) ‹var-m- φ u-eq-xmu- ψ x diff-add-cancel›*)
 also have $\dots = e (\mathbf{g}_{G_p}^{\wedge_{G_p}} (\text{poly } \varphi \ \alpha)) \mathbf{g}_{G_p}$
 by (*simp add: e-linear-in-fst*)
 finally show $e (\mathbf{g}_{G_p}^{\wedge_{G_p}} \text{poly } (\psi\text{-of } \varphi \ i) \ \alpha) (\mathbf{g}_{G_p}^{\wedge_{G_p}} \alpha \div_{G_p} \mathbf{g}_{G_p}^{\wedge_{G_p}} i) \otimes_{G_T}$
 $e \mathbf{g}_{G_p} \mathbf{g}_{G_p}^{\wedge_{G_T}} \text{poly } \varphi \ i =$
 $e (\mathbf{g}_{G_p}^{\wedge_{G_p}} \text{poly } \varphi \ \alpha) \mathbf{g}_{G_p}$

qed

11.0.2 Helping lemmas about the public parameters PK

Lemma that proves that the construction to calculate the public parameters in Isabelle actually computes the public parameters. Showing that the i th public parameter is actually the i th public parameter ($g^{\wedge(\alpha \wedge i)}$)

```

lemma PK-i:  $i \leq t \implies \text{map } (\lambda t. \mathbf{g}^{\wedge_{G_p} \alpha \wedge t}) [0..<t+1] ! i = \mathbf{g}_{G_p}^{\wedge_{G_p} (\alpha \wedge i)}$ 
proof (induction t)
  case 0
  then show ?case by force
next
  case (Suc t)
  then show ?case
  proof (cases  $i \leq t$ )
    case True
    then show ?thesis
    by (metis (no-types, lifting) Groups.add-ac(2) Suc(1) Suc(2) diff-zero le-imp-less-Suc
nth-map-upt plus-1-eq-Suc)
  next
  case False
  then show ?thesis
  by (metis (no-types, lifting) Suc(2) add-Suc-shift le-SucE le-imp-less-Suc
less-diff-conv nth-map-upt plus-1-eq-Suc semiring-norm(51))
  qed
qed

```

show $(\prod \text{PK}. \phi) = \mathbf{g} * \mathbf{g}^{\wedge(\alpha \wedge \text{1st coefficient of } \phi)} * \mathbf{g}^{\wedge((\alpha \wedge 2) \wedge \text{2nd coefficient of } \phi)} * \dots * \mathbf{g}^{\wedge((\alpha \wedge t) \wedge \text{t-th coefficient of } \phi)}$ Which is the first prestep to showing $(\prod \text{PK}. \phi) = \mathbf{g}^{\wedge \phi(\alpha)}$.

```

lemma g-pow-PK-Prod-to-fold[simp]:  $\text{degree } \varphi \leq t \implies g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}^{\wedge_{G_p} \alpha \wedge t}) [0..<t+1]) \varphi$ 
  = fold  $(\lambda pk \ g. g \otimes_{G_p} (\mathbf{g}_{G_p}^{\wedge_{G_p} (\alpha \wedge pk)}) \wedge_{G_p} (\text{poly.coeff } \varphi \ pk)) [0..<\text{Suc } (\text{degree } \varphi)] \mathbf{1}_{G_p}$ 
proof -
  let ?PK = map  $(\lambda t. \mathbf{g}^{\wedge_{G_p} \alpha \wedge t}) [0..<t+1]$ 
  let ?g-pow-PK = g-pow-PK-Prod ?PK  $\varphi$ 
  assume asmt:  $\text{degree } \varphi \leq t$ 
  have ?g-pow-PK = fold  $(\lambda pk \ g. g \otimes_{G_p} ?PK ! pk \wedge_{G_p} (\text{poly.coeff } \varphi \ pk)) [0..<\text{Suc } (\text{degree } \varphi)] \mathbf{1}_{G_p}$ 
  by auto
  also have fold  $(\lambda pk \ g. g \otimes_{G_p} (?PK) ! pk \wedge_{G_p} (\text{poly.coeff } \varphi \ pk)) [0..<\text{Suc } (\text{degree } \varphi)] \mathbf{1}_{G_p}$ 
  = fold  $(\lambda pk \ g. g \otimes_{G_p} (\mathbf{g}_{G_p}^{\wedge_{G_p} (\alpha \wedge pk)}) \wedge_{G_p} (\text{poly.coeff } \varphi \ pk)) [0..<\text{Suc } (\text{degree } \varphi)] \mathbf{1}_{G_p}$ 
  proof (rule List.fold-cong)

```

show $\bigwedge x. x \in \text{set } [0..<\text{Suc } (\text{degree } \varphi)] \implies$
 $(\lambda g. g \otimes_{G_p} ?PK ! x \hat{}_{G_p} \text{poly.coeff } \varphi x)$
 $= (\lambda g. g \otimes_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} x) \hat{}_{G_p} \text{poly.coeff } \varphi x)$
proof
fix $x::\text{nat}$
fix $g::'a$
assume $x \in \text{set } [0..<\text{Suc } (\text{degree } \varphi)]$
then have $?PK ! x = (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} x)$
using *PK-i asmt by auto*
then show $g \otimes_{G_p} ?PK ! x \hat{}_{G_p} \text{poly.coeff } \varphi x = g \otimes_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} x)$
 $\hat{}_{G_p} \text{poly.coeff } \varphi x$
by *presburger*
qed
qed *simp-all*
ultimately show $g\text{-pow-PK-Prod } ?PK \varphi = \text{fold } (\lambda pk g. g \otimes_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} pk) \hat{}_{G_p} \text{poly.coeff } \varphi pk) [0..<\text{Suc } (\text{degree } \varphi)] \mathbf{1}_{G_p}$
by *argo*
qed

show $g^{\wedge}(\sum i \leq n. \text{coeff } \varphi i * \alpha^{\wedge} i) = g * g^{\wedge}(\alpha^{\wedge} \text{1st coefficient of } \phi) * g^{\wedge}((\alpha^{\wedge} 2)^{\wedge} \text{2nd coefficient of } \phi) * \dots * g^{\wedge}((\alpha^{\wedge} t)^{\wedge} \text{t-th coefficient of } \phi)$ Which is the first prestep to showing $(\prod PK. \phi) = g^{\wedge} \phi(\alpha)$.

lemma *poly-altdef-to-fold[symmetric]*: $n \leq \text{degree } \varphi \implies \mathbf{g}_{G_p} \hat{}_{G_p} (\sum i \leq n. \text{poly.coeff } \varphi i * \alpha^{\wedge} i)$
 $= \text{fold } (\lambda n g. g \otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi n * \alpha^{\wedge} n))$
 $[0..<\text{Suc } n] \mathbf{1}_{G_p}$
proof (*induction n*)
case 0
have $\mathbf{g}_{G_p} \hat{}_{G_p} (\sum i \leq 0. \text{poly.coeff } \varphi i * \alpha^{\wedge} i) = \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi 0 * \alpha^{\wedge} 0)$
by *force*
moreover have $\text{fold } (\lambda n g. g \otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi n * \alpha^{\wedge} n)) [0..<\text{Suc } 0] \mathbf{1}_{G_p}$
 $= \mathbf{1}_{G_p} \otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi (0::\text{nat}) * \alpha^{\wedge} (0::\text{nat}))$ **by** *force*
moreover have $\mathbf{1}_{G_p} \otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi (0::\text{nat}) * \alpha^{\wedge} (0::\text{nat}))$
 $= \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi (0::\text{nat}) * \alpha^{\wedge} (0::\text{nat}))$ **using** $G_p.\text{generator-closed}$
 $G_p.\text{generator } G_p.l\text{-one}$ **by** *simp*
ultimately show *?case by argo*
next
case (*Suc n*)
have $\mathbf{g}_{G_p} \hat{}_{G_p} (\sum i \leq \text{Suc } n. \text{poly.coeff } \varphi i * \alpha^{\wedge} i)$
 $= \mathbf{g}_{G_p} \hat{}_{G_p} ((\sum i \leq n. \text{poly.coeff } \varphi i * \alpha^{\wedge} i)$
 $+ \text{poly.coeff } \varphi (\text{Suc } n) * \alpha^{\wedge} (\text{Suc } n))$ **by** *force*
also have $\dots = \mathbf{g}_{G_p} \hat{}_{G_p} (\sum i \leq n. \text{poly.coeff } \varphi i * \alpha^{\wedge} i)$
 $\otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi (\text{Suc } n) * \alpha^{\wedge} (\text{Suc } n))$
using *mod-ring-pow-mult- G_p* **by** *fastforce*
also have $\dots = \text{fold } (\lambda n g. g \otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly.coeff } \varphi n * \alpha^{\wedge} n)) [0..<\text{Suc } n]$

$n] \mathbf{1}_{G_p}$
 $\otimes_{G_p} \mathbf{g}_{G_p} \hat{G_p} (poly.coeff \ \varphi \ (Suc \ n) * \alpha \hat{ } (Suc \ n))$
using Suc **by** $auto$
also have $\dots = fold \ (\lambda n \ g. \ g \otimes_{G_p} \mathbf{g}_{G_p} \hat{G_p} (poly.coeff \ \varphi \ n * \alpha \hat{ } n)) \ [0..<Suc \ (Suc \ n)] \ \mathbf{1}_{G_p}$
by $simp$
finally show $?case$.
qed

finally pull the last two lemmas together to show that the public parameters can be used to calculate $\hat{g}\phi(\alpha)$ from the public parameters, $(\prod PK. \phi) = \hat{g}\phi(\alpha)$. With lemma $g_pow_PK_Prod_to_fold$, we form the $g_pow_PK_Prod$ part, which represents $(\prod PK. \phi)$, into $g * \hat{g}(\alpha \hat{ } 1st \text{ coefficient of } \phi) * \hat{g}((\alpha \hat{ } 2) \hat{ } 2nd \text{ coefficient of } \phi) * \dots * \hat{g}((\alpha \hat{ } t) \hat{ } t\text{-th coefficient of } \phi)$. Which we further form into $\hat{g}(\sum i \leq n. coeff \ \varphi \ i * \alpha \hat{ } i)$, which is nothing else then $\hat{g}\phi(\alpha)$ ($poly_altdef$), with the $poly_altdef_to_fold$ lemma

lemma $g\text{-}pow\text{-}PK\text{-}Prod\text{-}correct: degree \ \varphi \leq t$
 $\implies g\text{-}pow\text{-}PK\text{-}Prod \ (map \ (\lambda t. \ \mathbf{g}_{G_p} \hat{G_p} \alpha \hat{ } t) \ [0..<t + 1]) \ \varphi$
 $= \mathbf{g}_{G_p} \hat{G_p} (poly \ \varphi \ \alpha)$
proof –
let $?g\text{-}pow\text{-}PK = g\text{-}pow\text{-}PK\text{-}Prod \ (map \ (\lambda t. \ \mathbf{g}_{G_p} \hat{G_p} \alpha \hat{ } t) \ [0..<t + 1]) \ \varphi$
assume $asmt: degree \ \varphi \leq t$
have $\mathbf{g}_{G_p} \hat{G_p} poly \ \varphi \ \alpha = \mathbf{g}_{G_p} \hat{G_p} (\sum i \leq degree \ \varphi. poly.coeff \ \varphi \ i * \alpha \hat{ } i)$
by ($simp \ add: poly\text{-}altdef$)
moreover have $?g\text{-}pow\text{-}PK = fold \ (\lambda n \ g. \ g \otimes_{G_p} \mathbf{g}_{G_p} \hat{G_p} (poly.coeff \ \varphi \ n * \alpha \hat{ } n)) \ [0..<Suc \ (degree \ \varphi)] \ \mathbf{1}_{G_p}$
proof –
have $?g\text{-}pow\text{-}PK = fold \ (\lambda pk \ g. \ g \otimes_{G_p} (\mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{ } pk)) \ \hat{G_p} (poly.coeff \ \varphi \ pk)) \ [0..<Suc \ (degree \ \varphi)] \ \mathbf{1}_{G_p}$
using $g\text{-}pow\text{-}PK\text{-}Prod\text{-}to\text{-}fold \ asmt$ **by** $blast$
moreover have $\forall n \ g. \ g \otimes_{G_p} (\mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{ } n)) \ \hat{G_p} (poly.coeff \ \varphi \ n)$
 $= g \otimes_{G_p} \mathbf{g}_{G_p} \hat{G_p} (poly.coeff \ \varphi \ n * \alpha \hat{ } n)$
by ($simp \ add: mod\text{-}ring\text{-}pow\text{-}pow\text{-}G_p \ mult.commute \ G_p.int\text{-}pow\text{-}pow$)
ultimately show $g\text{-}pow\text{-}PK\text{-}Prod \ (map \ (\lambda t. \ \mathbf{g}_{G_p} \hat{G_p} \alpha \hat{ } t) \ [0..<t + 1]) \ \varphi$
 $= fold \ (\lambda n \ g. \ g \otimes_{G_p} \mathbf{g}_{G_p} \hat{G_p} (poly.coeff \ \varphi \ n * \alpha \hat{ } n)) \ [0..<Suc \ (degree \ \varphi)] \ \mathbf{1}_{G_p}$
by $presburger$
qed
ultimately show $g\text{-}pow\text{-}PK\text{-}Prod \ (map \ (\lambda t. \ \mathbf{g}_{G_p} \hat{G_p} \alpha \hat{ } t) \ [0..<t + 1]) \ \varphi = \mathbf{g}_{G_p} \hat{G_p} poly \ \varphi \ \alpha$
using $poly\text{-}altdef\text{-}to\text{-}fold[of \ degree \ \varphi \ \varphi \ \alpha]$ **by** $fastforce$
qed

Finally put everything together and show perfect correctness of Eval and $verify_eval$

theorem $KZG\text{-}correct: correct\text{-}eval$

proof –

```

have  $\bigwedge \varphi i. \text{valid-poly } \varphi \implies \text{spmf } (\text{correct-eval-game } \varphi i) \text{ True} = 1$ 
proof -
  fix  $\varphi i$ 
  assume assms: valid-poly  $\varphi$ 
  show spmf (correct-eval-game  $\varphi i$ ) True = 1
  proof -
    let  $? \alpha = \lambda x. \text{of-int-mod-ring } (\text{int } x)$ 
    let  $?PK = \lambda x. (\text{map } (\lambda t. \mathbf{g} \hat{G}_p ? \alpha x \hat{t}) [0..<\text{max-deg}+1])$ 

    have correct-eval-game  $\varphi i = \text{do } \{$ 
      (ck, vk)  $\leftarrow \text{key-gen};$ 
      (c, d)  $\leftarrow \text{commit ck } \varphi;$ 
      let w = Eval ck d  $\varphi i;$ 
      return-spmf (verify-eval vk c i w)
    }
    unfolding correct-eval-game-def ..
    also have ... = do {
      x::nat  $\leftarrow \text{sample-uniform } (\text{order } G_p);$ 
      return-spmf
        (e (g-pow-PK-Prod ( $?PK x$ ) ( $\psi\text{-of } \varphi i$ ))(( $?PK x$ )!1  $\div_{G_p} (\mathbf{g}_{G_p} \hat{G}_p$ 
i))
           $\otimes_{G_T} e \mathbf{g}_{G_p} \mathbf{g}_{G_p} \hat{G_T} \text{poly } \varphi i$ 
          = e (g-pow-PK-Prod ( $?PK x$ )  $\varphi$ )  $\mathbf{g}_{G_p}$ )}
    unfolding commit-def Eval-def verify-eval-def key-gen-def Setup-def
    by (auto simp add: Let-def)
    also have ... = do {
      x::nat  $\leftarrow \text{sample-uniform } (\text{order } G_p);$ 
      return-spmf
        (e ( $\mathbf{g}_{G_p} \hat{G_p} (\text{poly } (\psi\text{-of } \varphi i) (? \alpha x))$ ) (( $\mathbf{g}_{G_p} \hat{G_p} (? \alpha x)$ )  $\div_{G_p}$ 
( $\mathbf{g}_{G_p} \hat{G_p} i$ ))  $\otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p} \hat{G_T} (\text{poly } \varphi i))$ 
          = e ( $\mathbf{g}_{G_p} \hat{G_p} (\text{poly } \varphi (? \alpha x))$ )  $\mathbf{g}_{G_p}$ })}
    proof -
      let g-pow-φ =  $\lambda x. \text{g-pow-PK-Prod } (?PK x) \varphi$ 
      let g-pow-ψ =  $\lambda x. \text{g-pow-PK-Prod } (?PK x) (\psi\text{-of } \varphi i)$ 
      let g-pow-α =  $\lambda x. (?PK x)!1$ 
      have g-pow-φ: g-pow-φ = ( $\lambda x. \mathbf{g}_{G_p} \hat{G_p} (\text{poly } \varphi (? \alpha x))$ )
      using g-pow-PK-Prod-correct assms unfolding valid-poly-def by presburger
      have degree-ψ: degree ( $\psi\text{-of } \varphi i$ )  $\leq \text{max-deg}$ 
      using assms degree-q-le-φ le-trans unfolding valid-poly-def by fast
      have g-pow-ψ: g-pow-ψ = ( $\lambda x. \mathbf{g}_{G_p} \hat{G_p} (\text{poly } (\psi\text{-of } \varphi i) (? \alpha x))$ )
      using g-pow-PK-Prod-correct [OF degree-ψ] by presburger
      have g-pow-α: g-pow-α = ( $\lambda x. \mathbf{g}_{G_p} \hat{G_p} (? \alpha x)$ )
      using PK-i d-pos by auto
      show ?thesis using g-pow-φ g-pow-ψ g-pow-α by metis
      qed
    also have ... = do {
      x::nat  $\leftarrow \text{sample-uniform } (\text{order } G_p);$ 
      return-spmf True}
    using eq-on-e by presburger

```

```

      also have ... = scale-spmf (weight-spmf (sample-uniform (order Gp)))
    (return-spmf True)
      using bind-spmf-const by metis
      finally show ?thesis by (simp add: Gp.order-gt-0)
    qed
  qed
  then show ?thesis
    unfolding correct-eval-def by blast
qed
end

end
theory CryptHOL-ext

imports CryptHOL.Cyclic-Group-SPMF HOL-Computational-Algebra.Polynomial
      Berlekamp-Zassenhaus.Finite-Field Polynomial-Interpolation.Polynomial-Interpolation

begin

```

12 Extensions to CryptHOL

Here we collect a handful of lemmas about CryptHOL games, that we use in our proofs. These lemmas are particularly useful to resort asserts (`assert_spmf`) within games and to prove so-called bridging steps i.e. subtle differences in games, as they allow to “peel off layers” of a game.

lemma *ennreal-spmf*: $\text{ennreal} (\text{spmf game1 True}) \leq \text{ennreal} (\text{spmf game2 True})$
 $\implies$
 $\text{spmf game1 True} \leq \text{spmf game2 True}$
 by *simp*

lemma *unpack-bind-spmf*: $X = X' \implies \text{bind-spmf } X \ Y = \text{bind-spmf } X' \ Y$
 by *simp*

lemma *unpack-bind-spmf'*: $Y = Y' \implies \text{bind-spmf } X \ Y = \text{bind-spmf } X \ Y'$
 by *simp*

lemma *unpack-bind-spmf-fun*: $X = X' \implies \text{bind-spmf } X \ (\lambda y. f \ y) = \text{bind-spmf } X' \ (\lambda y. f \ y)$
 by (fact *CryptHOL-ext.unpack-bind-spmf*)

lemma *unpack-try-spmf*: $X = X' \implies \text{TRY } X \ \text{ELSE } Y = \text{TRY } X' \ \text{ELSE } Y$
 by *simp*

lemma *unpack-try-spmf'*: $Y = Y' \implies \text{TRY } X \ \text{ELSE } Y = \text{TRY } X \ \text{ELSE } Y'$
 by *simp*

lemma *spmf-eqI'*: $X = X' \implies \text{spmf } X \ Y = \text{spmf } X' \ Y$
by *simp*

12.1 SPMF True

lemma *return-spmf-assert*: $\text{TRY return-spmf } X \ \text{ELSE return-spmf } \text{False} =$
 $\text{TRY bind-spmf (assert-spmf } X) (\lambda_. \text{return-spmf True}) \ \text{ELSE return-spmf False}$
by (*simp add: try-bind-assert-spmf*)

lemma *bind-spmf-independent-return-spmf*:
 $\text{lossless-spmf } x \implies \text{bind-spmf } x \ (\lambda x. \text{return-spmf } y) = \text{return-spmf } y$
by (*simp add: bind-eq-return-spmf*)

lemma *bind-spmf-le*:
 $(\bigwedge x. \text{spmf } (f \ x) \ \text{True} \leq \text{spmf } (f' \ x) \ \text{True}) \implies \text{spmf } (\text{bind-spmf } p \ f) \ \text{True} \leq$
 $\text{spmf } (\text{bind-spmf } p \ f') \ \text{True}$
apply (*simp add: spmf-bind integral-measure-spmf*)
apply (*rule integral-mono*)
apply (*rule integrableI-bounded*)
apply *simp*
apply (*smt (verit, del-Insts) ennreal-less-top ennreal-spmf-bind infinity-ennreal-def*
 $\text{nn-integral-cong pmf-nonneg real-norm-def}$)
apply (*rule integrableI-bounded*)
apply *simp*
apply (*smt (verit, del-Insts) ennreal-less-top ennreal-spmf-bind infinity-ennreal-def*
 $\text{nn-integral-cong pmf-nonneg real-norm-def}$)
apply *simp*
done

lemma *try-spmf-eq*:
assumes $\text{spmf } x \ \text{True} = \text{spmf } x' \ \text{True}$
shows $\text{spmf } (\text{TRY } x \ \text{ELSE return-spmf False}) \ \text{True} = \text{spmf } (\text{TRY } x' \ \text{ELSE}$
 $\text{return-spmf False}) \ \text{True}$
apply (*simp add: spmf-try-spmf*)
apply (*rule assms*)
done

lemma *try-spmf-le*:
assumes $\text{spmf } x \ \text{True} \leq \text{spmf } x' \ \text{True}$
shows $\text{spmf } (\text{TRY } x \ \text{ELSE return-spmf False}) \ \text{True} \leq \text{spmf } (\text{TRY } x' \ \text{ELSE}$
 $\text{return-spmf False}) \ \text{True}$
apply (*rule ennreal-spmf*)
apply (*simp add: spmf-try-spmf*)
apply (*rule assms*)
done

lemma *try-spmf-true-else-false-le*: $\text{spmf } (\text{TRY } X \ \text{ELSE return-spmf False}) \ \text{True}$
 $\leq \text{spmf } X \ \text{True}$
by (*simp add: spmf-try-spmf*)

lemma *del-assert*: $\text{spmf } (\text{bind-spmf } (\text{assert-spmf } X) (\lambda\cdot . Y)) \text{ True} \leq \text{spmf } Y$
True

apply (*rule ennreal-spmf*)
apply (*simp add: spmf-try-spmf ennreal-spmf-bind*)
apply (*simp add: mult-left-le measure-spmf.emeasure-space-le-1*)
done

lemma *assert-imp*: $(X \longrightarrow X') \implies$
 $\text{spmf } (\text{bind-spmf } (\text{assert-spmf } X) (\lambda\cdot . Y)) \text{ True}$
 $\leq \text{spmf } (\text{bind-spmf } (\text{assert-spmf } X') (\lambda\cdot . Y)) \text{ True}$
using *del-assert* **by** *fastforce*

lemma *assert-ret-unit*: $\text{bind-spmf } (\text{assert-spmf } x) (\lambda x . y) = \text{bind-spmf } (\text{assert-spmf } x) (\lambda\cdot . y)$
by *presburger*

lemma *assert-based-eq*:
assumes $x \implies y = y'$
shows $\text{bind-spmf } (\text{assert-spmf } x) (\lambda\cdot . y)$
 $= \text{bind-spmf } (\text{assert-spmf } x) (\lambda\cdot . y')$
by (*auto simp add: assms assert-spmf-def*)

lemma *assert-commute*: $\text{bind-spmf } X (\lambda x . \text{bind-spmf } (\text{assert-spmf } Y) (\lambda\cdot . Z x))$
 $= \text{bind-spmf } (\text{assert-spmf } Y) (\lambda\cdot . \text{bind-spmf } X (\lambda x . Z x))$
by (*rule bind-commute-spmf*)

lemma *assert-collapse*: $\text{bind-spmf } (\text{assert-spmf } X) (\lambda\cdot . \text{bind-spmf } (\text{assert-spmf } Y) (\lambda\cdot . Z))$
 $=$
 $\text{bind-spmf } (\text{assert-spmf } (X \wedge Y)) (\lambda\cdot . Z)$
by (*smt (verit) assert-spmf-simps(1,2) return-None-bind-spmf return-bind-spmf*)

lemma *assert-cong*: $X = Y \implies \text{rel-spmf } (=) (\text{assert-spmf } X) (\text{assert-spmf } Y)$
by (*simp add: spmf-rel-eq*)

lemma *rel-spmf-bind-assert-refl*: $(Z \implies \text{rel-spmf } P X Y) \implies$
 $\text{rel-spmf } P (\text{bind-spmf } (\text{assert-spmf } Z) (\lambda\cdot . X)) (\text{bind-spmf } (\text{assert-spmf } Z) (\lambda\cdot . Y))$
using *rel-spmf-bind-refl* **by** *fastforce*

We provide sampling of uniform random sets and lists. Additionally, we provide uniform sampling of polynomials over finite fields and show them equivalent to interpolating on a zipped list of coordinates where the evaluations are a uniformly random chosen list.

12.2 Sample uniform set

definition *sample-uniform-set* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat set spmf}$
where *sample-uniform-set* $k n = \text{spmf-of-set } \{x. x \subseteq \{..<n\} \wedge \text{card } x = k\}$

lemma *spmf-sample-uniform-set*: *spmf* (sample-uniform-set *k n*) *x*
 = indicator {*x*. $x \subseteq \{..<n\} \wedge \text{card } x = k\}$ *x* / (*n choose k*)
by (*simp add: n-subsets sample-uniform-set-def spmf-of-set*)

lemma *weight-sample-uniform-set*: *weight-spmf* (sample-uniform-set *k n*) = of-bool
 ($k \leq n$)
apply (*simp add: sample-uniform-set-def weight-spmf-of-set split: if-splits*)
apply (*rule conjI*)
apply (*metis card-lessThan card-mono finite-lessThan*)
using *card-lessThan* **by** *blast*

lemma *weight-sample-uniform-set-k-0* [*simp*]: *weight-spmf* (sample-uniform-set 0
n) = 1
by (*auto simp add: weight-sample-uniform-set*)

lemma *weight-sample-uniform-set-n-0* [*simp*]: *weight-spmf* (sample-uniform-set *k*
 0) = of-bool ($k=0$)
by (*auto simp add: weight-sample-uniform-set*)

lemma *weight-sample-uniform-set-k-le-n* [*simp*]: $k \leq n \implies \text{weight-spmf}$ (sample-uniform-set
k n) = 1
by (*auto simp add: weight-sample-uniform-set indicator-def gr0-conv-Suc*)

lemma *lossless-sample-uniform-set* [*simp*]: *lossless-spmf* (sample-uniform-set *k n*)
 $\longleftrightarrow k \leq n$
by (*auto simp add: lossless-spmf-def weight-sample-uniform-set intro: ccontr*)

lemma *set-spmf-sample-uniform-set* [*simp*]: *set-spmf* (sample-uniform-set *k n*) =
 {*x*. $x \subseteq \{..<n\} \wedge \text{card } x = k\}$
by (*simp add: sample-uniform-set-def*)

12.3 Sample uniform list

definition *sample-uniform-list* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat list* *spmf*
where *sample-uniform-list* *k n* = *spmf-of-set* {*x*. $\text{set } x \subseteq \{..<n\} \wedge \text{length } x = k\}$

lemma *spmf-sample-uniform-list*: *spmf* (sample-uniform-list *k n*) *x*
 = indicator {*x*. $\text{set } x \subseteq \{..<n\} \wedge \text{length } x = k\}$ *x* / (*n* ^ *k*)
by (*simp add: card-lists-length-eq sample-uniform-list-def spmf-of-set*)

lemma *weight-sample-uniform-list*: *weight-spmf* (sample-uniform-list *k n*) = of-bool
 ($k=0 \vee 0 < n$)

proof –

have $0 < n \longrightarrow (\exists x. \text{set } x \subseteq \{..<n\} \wedge \text{length } x = k)$

proof

assume *zero-l-n*: $0 < n$

show $\exists x. \text{set } x \subseteq \{..<n\} \wedge \text{length } x = k$

proof

```

    let ?x = replicate k 0
    show set ?x  $\subseteq$  {.. $n$ }  $\wedge$  length ?x = k
    using zero-l-n by fastforce
  qed
qed
then show ?thesis
  by (simp add: finite-lists-length-eq sample-uniform-list-def weight-spmf-of-set)
qed

lemma weight-sample-uniform-list-k-0 [simp]: weight-spmf (sample-uniform-list 0
n) = 1
  by (auto simp add: weight-sample-uniform-list)

lemma weight-sample-uniform-list-n-0 [simp]: weight-spmf (sample-uniform-list k
0) = of_bool (k=0)
  by (auto simp add: weight-sample-uniform-list)

lemma weight-sample-uniform-list-k-le-n [simp]: k < n  $\implies$  weight-spmf (sample-uniform-list
k n) = 1
  by (auto simp add: weight-sample-uniform-list less-iff-Suc-add)

lemma lossless-sample-uniform-list [simp]: lossless-spmf (sample-uniform-list k n)
 $\longleftrightarrow$  (k=0  $\vee$  0 < n)
  by (auto simp add: lossless-spmf-def weight-sample-uniform-list intro: ccontr)

lemma set-spmf-sample-uniform-list [simp]: set-spmf (sample-uniform-list k n) =
{x. set x  $\subseteq$  {.. $n$ }  $\wedge$  length x = k}
  by (simp add: finite-lists-length-eq sample-uniform-list-def)

the two following lemmas are helping lemmas for Cons_random_list_split

lemma set-spmf-lhs: set-spmf (map-spmf ((#) x) (sample-uniform-list k p))
= {xs. set (tl xs)  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k+1  $\wedge$  hd xs = x}

proof –
  fix x
  have set-spmf (map-spmf ((#) x) (sample-uniform-list k p))
    = ((#) x) ‘ {xs. set xs  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k}
    unfolding sample-uniform-list-def
    by (simp add: finite-lists-length-eq)
  also have ... = {xs. set (tl xs)  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k+1  $\wedge$  hd xs = x}
  proof (rule equalityI; rule subsetI)
    fix ys assume ys  $\in$  ((#) x) ‘ {xs. set xs  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k}
    then obtain zs where zs  $\in$  {xs. set xs  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k} and ys =
x # zs
    by blast
    then show ys  $\in$  {xs. set (tl xs)  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k+1  $\wedge$  hd xs = x}
    by simp
  next
    fix ys assume ys: ys  $\in$  {xs. set (tl xs)  $\subseteq$  {.. $p$ }  $\wedge$  length xs = k+1  $\wedge$  hd xs =
x}

```

```

then have facts: set (tl ys)  $\subseteq$  {.. $p$ } length ys =  $k+1$  hd ys =  $x$  by simp-all
then obtain a zs where azs: ys = a # zs and len-zs: length zs =  $k$ 
  by (metis Suc-eq-plus1 Suc-length-conv)
have ys =  $x$  # zs using azs facts by simp
moreover have zs  $\in$  {xs. set xs  $\subseteq$  {.. $p$ }  $\wedge$  length xs =  $k$ }
  using azs facts len-zs by simp
ultimately show ys  $\in$  (#)  $x$  ‘ {xs. set xs  $\subseteq$  {.. $p$ }  $\wedge$  length xs =  $k$ }
  by (metis image-eqI)
qed
finally show set-spmf (map-spmf ((#)  $x$ ) (sample-uniform-list  $k$  p))
  = {xs. set (tl xs)  $\subseteq$  {.. $p$ }  $\wedge$  length xs =  $k + 1$   $\wedge$  hd xs =  $x$ }
  .
qed

lemma set-spmf-lhs-imp-length-gr-0: xs  $\in$  set-spmf (map-spmf ((#)  $x$ ) (sample-uniform-list
 $k$  p))
 $\implies$  length xs  $> 0$ 
unfolding set-spmf-lhs by force

sampling a uniform random list can be split up into prepending a uniform
random element to a one element short uniform random list. Intuitively:
sample_uniform_list ( $k+1$ ) = Cons (sample_uniform) (sample_uniform_list
 $k$ )

lemma Cons-random-list-split:
  assumes  $p > 1$ 
  shows do { $x \leftarrow$  sample-uniform  $p$ ;
    map-spmf ((#)  $x$ ) (sample-uniform-list  $k$  p)} = sample-uniform-list ( $k+1$ )
 $p$ 
(is ?lhs = ?rhs)
proof -
  have  $\forall s. \text{spm}f \text{ ?lhs } s = \text{spm}f \text{ ?rhs } s$ 
  proof
    fix  $s$ 
    have spmf-rhs: spmf ?rhs  $s = \text{indicator } \{x. \text{set } x \subseteq \{.. $p\}\} \wedge \text{length } x = k+1\}$ 
 $s / \text{real } p^{k+1}$ 
    unfolding spmf-sample-uniform-list ..
    have spmf-lhs: spmf ?lhs  $s$ 
      = ( $\sum x < p. \text{ennreal } (\text{spm}f (\text{map-spmf } ((\#) \text{ } x) (\text{sample-uniform-list } k \text{ } p)) \text{ } s))$ 
      / of-nat (card {.. $p$ })
    unfolding ennreal-spmf-bind
    unfolding sample-uniform-def
    unfolding nn-integral-spmf-of-set
    ..
  show spmf ?lhs  $s = \text{spm}f \text{ ?rhs } s$ 
  proof (cases  $s$ )
    case Nil
    have spmf ?lhs  $s = 0$ 
    proof -$ 
```

```

have  $\bigwedge x. \text{pmf } (\text{map-pmf } ((\#) x) (\text{sample-uniform-list } k p)) s = 0$ 
  using set-pmf-lhs set-pmf-lhs-imp-length-gr-0 pmf-eq-0-set-pmf
  unfolding Nil
  by fast
then show ?thesis
  using pmf-lhs by force
qed
moreover have pmf ?rhs s = 0
  using pmf-rhs
  unfolding Nil
  by force
ultimately show ?thesis by presburger
next
case (Cons s-hd s-tl)
then show ?thesis
proof (cases s-hd < p)
  case True
  have pmf ?lhs s
    = (indicator  $\{x. \text{set } x \subseteq \{..<p\} \wedge \text{length } x = k\} s\text{-tl} / (\text{real } p^{(k+1)})$ )
  proof -
    have pmf-is-sum: pmf ?lhs s =
      sum ( $\lambda x. \text{ennreal } (\text{pmf } (\text{map-pmf } ((\#) x) (\text{sample-uniform-list } k p))$ 
s)  $\{..<p\}$ 
      / real (card  $\{..<p\}$ )
    unfolding pmf-lhs
    using ennreal-of-nat-eq-real-of-nat by presburger
    also have sum ( $\lambda x. \text{ennreal } (\text{pmf } (\text{map-pmf } ((\#) x) (\text{sample-uniform-list } k p))$ 
s)  $\{..<p\}$ 
      = sum ( $\lambda x. \text{ennreal } (\text{pmf } (\text{map-pmf } ((\#) x) (\text{sample-uniform-list } k p))$ 
s)  $(\{..<p\} - \{s\text{-hd}\}) \cup \{s\text{-hd}\}$ )
    by (simp add: True insert-absorb)
    also have ... = sum ( $\lambda x. \text{ennreal } (\text{pmf } (\text{map-pmf } ((\#) x) (\text{sample-uniform-list } k p))$ 
s)  $(\{..<p\} - \{s\text{-hd}\})$ 
      + ennreal (pmf (map-pmf  $((\#) s\text{-hd})$  (sample-uniform-list k p) s)
    by (metis (no-types, lifting) Un-insert-right add commute dual-order.refl
finite-nat-iff-bounded insert-Diff-single sum.insert-remove sup-bot.right-neutral)
    also have ... = ennreal (pmf (map-pmf  $((\#) s\text{-hd})$  (sample-uniform-list
k p) s)
  proof -
    have  $\bigwedge x. x \in \{..<p\} - \{s\text{-hd}\} \longrightarrow \text{ennreal } (\text{pmf } (\text{map-pmf } ((\#) x) (\text{sample-uniform-list } k p)) s) = 0$ 
  proof
    fix x
    assume  $x \in \{..<p\} - \{s\text{-hd}\}$ 
    then have  $x \neq s\text{-hd}$  by blast
    then show ennreal (pmf (map-pmf  $((\#) x)$  (sample-uniform-list k
p) s) = 0
      using set-pmf-lhs
      unfolding Cons

```

```

    by (simp add: spmf-eq-0-set-spmf)
  qed
  then show ?thesis by force
  qed
  also have ... = ennreal (spmf (sample-uniform-list k p) s-tl)
    unfolding Cons
    using spmf-map-inj
    by (simp add: spmf-map-inj')
  also have ... = indicator {x. set x ⊆ {..

```

```

    using False
    by fastforce
  ultimately show ?thesis by presburger
qed
qed
qed
then show ?thesis
  using spmf-eqI by blast
qed

```

This corollary puts the last lemma in a more readable and thus workable definition. Intuitively: $\text{sample_uniform_list } (k+1) = \text{Cons } (\text{sample_uniform}) (\text{sample_uniform_list } k)$

```

corollary pretty-Cons-random-list-split:
  assumes  $p > 1$ 
  shows  $\text{sample-uniform-list } (k+1) \ p =$ 
     $\text{do } \{x \leftarrow \text{sample-uniform } p;$ 
       $xs \leftarrow (\text{sample-uniform-list } k \ p);$ 
       $\text{return-spmf } (x \# xs)\}$ 
  (is ?lhs = ?rhs)
proof -
  have ?rhs =  $\text{do } \{x \leftarrow \text{sample-uniform } p;$ 
     $\text{map-spmf } ((\#) \ x) \ (\text{sample-uniform-list } k \ p)\}$ 
  by (simp add: map-spmf-conv-bind-spmf)
  then show ?thesis
    using Cons-random-list-split[symmetric, OF assms]
    by presburger
qed

```

12.4 Sample uniform polynomial

definition $\text{sample-uniform-poly} :: \text{nat} \Rightarrow 'a::\text{zero poly spmf}$
 where $\text{sample-uniform-poly } t = \text{spmfs-of-set } \{p. \text{degree } p \leq t\}$

lemma $\text{of-int-mod-inj-on-ff}: \text{inj-on } (\text{of-int-mod-ring} \circ \text{int} :: \text{nat} \Rightarrow 'e::\text{prime-card mod-ring}) \ \{.. < \text{CARD } ('e)\}$

```

proof
  fix x
  fix y
  assume  $x: x \in \{.. < \text{CARD } ('e)\}$ 
  assume  $y: y \in \{.. < \text{CARD } ('e)\}$ 
  assume  $\text{app-x-eq-y}: (\text{of-int-mod-ring} \circ \text{int} :: \text{nat} \Rightarrow 'e \text{ mod-ring}) \ x = (\text{of-int-mod-ring} \circ \text{int} :: \text{nat} \Rightarrow 'e \text{ mod-ring}) \ y$ 
  show  $x = y$ 
    using x y app-x-eq-y
    by (metis comp-apply lessThan-iff nat-int of-nat-0-le-iff of-nat-less-iff to-int-mod-ring-of-int-mod-ring)
qed

```

sampling a uniform random polynomial is equivalent to interpolating a poly-

nomial from a list of uniform random chosen evaluations

lemma *sample-uniform-evals-is-sample-poly*:

assumes *distinct I*

and *length I = t+1*

shows $(\text{sample-uniform-poly } t::'e \text{ mod-ring poly } \text{spm}f) = \text{do } \{$
 $\text{evals}::('e::\text{prime-card}) \text{ mod-ring list} \leftarrow \text{map-spmf } (\text{map } (\text{of-int-mod-ring} \circ \text{int}::$
 $\text{nat} \Rightarrow 'e \text{ mod-ring})) (\text{sample-uniform-list } (t+1) (\text{CARD } ('e)));$
 $\text{return-spmf } (\text{lagrange-interpolation-poly } (\text{zip } I \text{ evals}))\}$
(is ?lhs = ?rhs)

proof $-$

have *?rhs = do {*
 $\text{evals} \leftarrow \text{spm}f\text{-of-set } \{x::'e \text{ mod-ring list. length } x = t + 1\};$
 $\text{return-spmf } (\text{lagrange-interpolation-poly } (\text{zip } I \text{ evals}))\}$

proof $-$

have *uni-list-set-is-card-set*: $(\bigcup (\text{set } ' \{x::\text{nat list. set } x \subseteq \{..<\text{CARD}('e)\} \wedge$
 $\text{length } x = t + (1::\text{nat})\}))$
 $= \{..<\text{CARD}('e)\}$

proof

show $\{..<\text{CARD}('e)\} \subseteq \bigcup (\text{set } ' \{x::\text{nat list. set } x \subseteq \{..<\text{CARD}('e)\} \wedge \text{length}$
 $x = t + (1::\text{nat})\})$

proof

fix *x*

assume $x \in \{..<\text{CARD}('e)\}$

then have *replicate (t+1) x* $\in \{x::\text{nat list. set } x \subseteq \{..<\text{CARD}('e)\} \wedge \text{length}$
 $x = t + (1::\text{nat})\}$

by *fastforce*

then show $x \in \bigcup (\text{set } ' \{x::\text{nat list. set } x \subseteq \{..<\text{CARD}('e)\} \wedge \text{length } x =$
 $t + (1::\text{nat})\})$

by *fastforce*

qed

qed *auto*

have $\text{map-spmf } (\text{map } (\text{of-int-mod-ring} \circ \text{int}:: \text{nat} \Rightarrow 'e \text{ mod-ring})) (\text{sample-uniform-list}$
 $(t + 1) \text{ CARD}('e))$

$= \text{spm}f\text{-of-set } ((\text{map } (\text{of-int-mod-ring} \circ \text{int}:: \text{nat} \Rightarrow 'e \text{ mod-ring})) ' \{x. \text{set } x \subseteq$
 $\{..<\text{CARD}('e)\} \wedge \text{length } x = t+1\})$

(is ?map = ?set)

unfolding *sample-uniform-list-def*

apply *(rule map-spmf-of-set-inj-on)*

apply *(rule inj-on-mapI)*

unfolding *uni-list-set-is-card-set*

by *(rule of-int-mod-inj-on-ff)*

also have $(\text{map } (\text{of-int-mod-ring} \circ \text{int}:: \text{nat} \Rightarrow 'e \text{ mod-ring})) ' \{x. \text{set } x \subseteq$
 $\{..<\text{CARD}('e)\} \wedge \text{length } x = t+1\}$

$= \{x::'e \text{ mod-ring list. length } x = t + 1\}$

proof

show $\{x::'e \text{ mod-ring list. length } x = t + (1::\text{nat})\}$

$\subseteq \text{map } (\text{of-int-mod-ring} \circ \text{int}:: \text{nat} \Rightarrow 'e \text{ mod-ring}) ' \{x::\text{nat list. set } x \subseteq$
 $\{..<\text{CARD}('e)\} \wedge \text{length } x = t + (1::\text{nat})\}$

proof

```

fix x
assume asm:  $x \in \{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
obtain x-int where x-int : x-int = map to-int-mod-ring x by force
have x-eq-map-x-int:  $x = \text{map of-int-mod-ring } x\text{-int}$ 
  unfolding x-int
  by (simp add: nth-equalityI)
obtain x-nat where x-nat: x-nat = map nat x-int by simp
have x-int-x-nat: x-int = map int x-nat
proof –
  have map int x-nat = map ( $\lambda i. \text{if } i \geq 0 \text{ then } i \text{ else } 0$ ) x-int
  unfolding x-nat using int-nat-eq by simp
  moreover have  $\forall x \in \text{set } x\text{-int. } x \geq 0$ 
  unfolding x-int
  using range-to-int-mod-ring
  by (metis atLeastLessThan-iff ex-map-conv rangeI)
  ultimately show ?thesis
  by (simp add: map-idI)
qed
have x-nat  $\in \{x::nat \text{ list. set } x \subseteq \{..<CARD('e)\} \wedge \text{length } x = t + (1::nat)\}$ 
  unfolding x-nat x-int
  apply (rule CollectI, rule conjI)
  apply (simp only: set-map)
  apply (rule subsetI)
  apply (metis range-to-int-mod-ring UNIV-I atLeastLessThan-iff image-iff
nat-less-iff lessThan-iff)
  using asm by simp
  moreover have  $x = \text{map (of-int-mod-ring } \circ \text{int}::nat \Rightarrow 'e \text{ mod-ring}) } x\text{-nat}$ 
  by (simp add: x-eq-map-x-int x-int-x-nat)
  ultimately show  $x \in \text{map (of-int-mod-ring } \circ \text{int}::nat \Rightarrow 'e \text{ mod-ring})$ 
    ‘  $\{x::nat \text{ list. set } x \subseteq \{..<CARD('e)\} \wedge \text{length } x = t + (1::nat)\}$ 
  by blast
qed
qed auto
finally show ?thesis by metis
qed
also have ... = do {
  evals  $\leftarrow$  spmf-of-set  $\{x::'e \text{ mod-ring list. length } x = t + 1\}$ ;
  let  $\varphi = \text{lagrange-interpolation-poly (zip I evals)}$ ;
  return-spmf  $\varphi$  } unfolding Let-def ..
also have ... = map-spmf ( $\lambda \text{evals. lagrange-interpolation-poly (zip I evals)}$ )
  (spm-f-of-set  $\{x::'e \text{ mod-ring list. length } x = t + 1\}$ )
  by (simp add: map-spmf-conv-bind-spmf)
also have ... = spmf-of-set ( $\lambda \text{evals. lagrange-interpolation-poly (zip I evals)}$ ) ‘
 $\{x::'e \text{ mod-ring list. length } x = t + 1\}$ 
proof (rule map-spmf-of-set-inj-on)
  show inj-on ( $\lambda \text{evals}::'e \text{ mod-ring list. lagrange-interpolation-poly (zip I evals)}$ )
     $\{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
proof
  fix x y

```

```

    assume x-in:  $x \in \{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
    assume y-in:  $y \in \{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
    assume asm: lagrange-interpolation-poly (zip I x) = lagrange-interpolation-poly
(zip I y)
    have poly-xi:  $\bigwedge i. i < \text{length } I \implies \text{poly } (\text{lagrange-interpolation-poly } (\text{zip } I \ x))$ 
(I!i) = x!i
    using lagrange-interpolation-poly[of zip I x lagrange-interpolation-poly (zip
I x)]
    assms(1)
    by (smt (verit, del-insts) assms(2) length-zip map-fst-zip mem-Collect-eq
min-less-iff-conj nth-mem nth-zip x-in)
    have poly-y:  $\bigwedge i. i < \text{length } I \implies \text{poly } (\text{lagrange-interpolation-poly } (\text{zip } I \ y))$ 
(I!i) = y!i
    using lagrange-interpolation-poly[of zip I y lagrange-interpolation-poly (zip
I y)]
    assms(1)
    by (smt (verit, del-insts) assms(2) length-zip map-fst-zip mem-Collect-eq
min-less-iff-conj nth-mem nth-zip y-in)
    then have  $\bigwedge i. i < \text{length } I \implies \text{poly } (\text{lagrange-interpolation-poly } (\text{zip } I \ x))$ 
(I!i) = y!i
    using asm by presburger
    then show  $x=y$ 
    using poly-y assms(2) x-in y-in
    by (simp add: nth-equalityI poly-xi)
qed
qed
also have ... = spmf-of-set {p. degree p ≤ t}
(is ?map = ?set)
proof -
    have {p. degree p ≤ t} = (λevals::'e mod-ring list. lagrange-interpolation-poly
(zip I evals)) '
    {x::'e mod-ring list. length x = t + (1::nat)}
    proof
    show {p::'e mod-ring poly. degree p ≤ t}
    ⊆ (λevals::'e mod-ring list. lagrange-interpolation-poly (zip I evals)) '
    {x::'e mod-ring list. length x = t + (1::nat)}
    proof
    fix x
    assume asm:  $x \in \{p::'e \text{ mod-ring poly. degree } p \leq t\}$ 
    obtain x-evals where x-evals:  $x\text{-evals} = \text{map } (\lambda i. \text{poly } x \ i) \ I$  by simp
    then have length x-evals = t+1
    by (simp add: assms(2))
    moreover have  $x = \text{lagrange-interpolation-poly } (\text{zip } I \ x\text{-evals})$ 
    apply (rule uniqueness-of-interpolation-point-list[of zip I x-evals x la-
grange-interpolation-poly (zip I x-evals)])
    apply (simp add: assms(1) x-evals)
    apply (metis fst-eqD in-set-zip nth-map snd-eqD x-evals)
    using asm assms(2) calculation apply force
    apply (metis assms(1) assms(2) calculation lagrange-interpolation-poly

```

```

map-fst-zip)
  apply (metis Suc-eq-plus1 assms(2) calculation degree-lagrange-interpolation-poly
diff-Suc-1 le-refl length-zip less-Suc-eq min.absorb1 order.strict-trans1)
  done
  ultimately show  $x \in (\lambda evals::'e \text{ mod-ring list. lagrange-interpolation-poly}$ 
(zip I evals)) ‘
     $\{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
  by blast
qed
show  $(\lambda evals::'e \text{ mod-ring list. lagrange-interpolation-poly (zip I evals))$  ‘
 $\{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
 $\subseteq \{p::'e \text{ mod-ring poly. degree } p \leq t\}$ 
proof
  fix x
  assume asm:  $x \in (\lambda evals::'e \text{ mod-ring list. lagrange-interpolation-poly (zip}$ 
I evals)) ‘
     $\{x::'e \text{ mod-ring list. length } x = t + (1::nat)\}$ 
  then show  $x \in \{p::'e \text{ mod-ring poly. degree } p \leq t\}$ 
  using degree-lagrange-interpolation-poly
  by (smt (verit) Nat.le-diff-conv2 One-nat-def Suc-eq-plus1 add-leE diff-Suc-1
diff-is-0-eq image-iff le-trans length-zip mem-Collect-eq min.absorb1 min.absorb2
nat-le-linear plus-1-eq-Suc zero-le)
qed
qed
then show ?thesis by argo
qed
finally show ?thesis
  unfolding sample-uniform-poly-def by argo
qed

end
theory KZG-poly-bind

imports KZG-correct CryptHOL-ext tDL-assumption
Berlekamp-Zassenhaus.Finite-Field-Factorization Elimination-Of-Repeated-Factors.ERF-Algorithm

begin

```

13 Polynomial Binding of the KZG

We show that the KZG is polynomial binding for every polynomial of degree $\leq \text{max_deg}$. We use the `Sigma_Commit_Crypto` template to prove the binding game. The proof is adapted from Appendix C.1 of the original KZG paper [8].

```

locale KZG-CS-binding = KZG-PCS-correct
begin

```

13.1 t-DL game

We reduce the polynomial binding to the t-DL assumption

```
sublocale t-DL-Gp: t-DL Gp max-deg of-int-mod-ring  $\circ$  int pow-mod-ring Gp
  unfolding t-DL-def
  by (rule Gp.cyclic-group-axioms)
```

Intuetively what we want to show is that if we have an adversary that can compute two polynomials such that they have the same commitment in polynomial time, we can construct an algorithm, using that adversary, that can solve the t-DL in polynomial time, thus breaking the t-DL assumption.

This functions purpose is to extract α based on the inputs g^α and ϕ , where ϕ has a root at α . The function factorizes ϕ and filters for all roots. Since α 's `mod_ring` is of the same cardinality as g 's group's order, we can conclude that if $g^r = g^\alpha$ then $r = \alpha$

```
fun find- $\alpha$ -square-free :: 'a  $\Rightarrow$  'e mod-ring poly  $\Rightarrow$  'e mod-ring where
  find- $\alpha$ -square-free g-pow- $\alpha$   $\varphi$  = (let (c, polys) = finite-field-factorization  $\varphi$ ;
    deg1-polys = filter ( $\lambda f$ . degree f = 1) polys;
    root-list = map ( $\lambda p$ . poly.coeff p 0) deg1-polys;
     $\alpha$ -roots = filter ( $\lambda r$ . g-pow- $\alpha$  =  $\mathbf{g}_{G_p}^{\wedge_{G_p} -r}$ ) root-list
  in - $\alpha$ -roots!0)
```

The radical is executable via the formalization of the 'Elimination of Repeated Factors Algorithm' in the AFP (see https://www.isa-afp.org/entries/Elimination_Of_Repeated_Factors.html)

```
fun find- $\alpha$  :: 'a  $\Rightarrow$  'e mod-ring poly  $\Rightarrow$  'e mod-ring where
  find- $\alpha$  g-pow- $\alpha$   $\varphi$  = find- $\alpha$ -square-free g-pow- $\alpha$  (radical  $\varphi$ )
```

The reduction: An adversary for the KZG polynomial binding can output two polynomials ϕ and ϕ' that have the same commitment, i.e $g^{\phi(\alpha)} = g^{\phi'(\alpha)}$, which is equivalent to $\phi(\alpha) = \phi'(\alpha)$ (same argument as in the function above). Hence $(\phi - \phi')(\alpha) = 0$, so $(\phi - \phi')$ has a root at α . Furthermore we have g^α in the public key at position 1. Hence we can use the *find- α* function to compute α in polynomial time. Given α we can easily compute a c and a g' such that $g^{1/(\alpha+c)} = g'$. E.g. c=0 and $g' = g^{1/\alpha}$ Hence we can break the t-DL assumption, as we have a polynomial-time algorithm to compute (c,g).

```
fun bind-reduction
  :: ('a ck, 'e mod-ring poly, 'a commit, unit) bind-adversary  $\Rightarrow$  ('a, 'e) t-DL.adversary
```

where

```
bind-reduction  $\mathcal{A}$  PK = do {
  (C,  $\varphi$ , -,  $\varphi'$ , -)  $\leftarrow$   $\mathcal{A}$  PK;
  let  $\alpha$  = find- $\alpha$  (PK!1) ( $\varphi$  -  $\varphi'$ );
  return-spmf  $\alpha$ }
```

This function captures the `bind_reduction` with the asserts from the binding game, essentially allowing us to prove the equivalence of binding game to t-DL game with `stronger_bind_reduction`. From this equivalence it is easy to proof the theorem winnnng the binding game is less than or equal to breaking the t-DL assumption

fun *stronger-bind-reduction*
 $:: ('a\ ck, 'e\ mod\text{-}ring\ poly, 'a\ commit, unit)\ bind\text{-}adversary \Rightarrow ('a, 'e)\ t\text{-}DL.adversary$

where

stronger-bind-reduction $\mathcal{A}\ PK = do \{$
 $(C, \varphi, -, \varphi', -) \leftarrow \mathcal{A}\ PK;$
 $- :: unit \leftarrow assert\text{-}spmf(\varphi \neq \varphi'$
 $\wedge valid\text{-}poly\ \varphi$
 $\wedge valid\text{-}poly\ \varphi'$
 $\wedge (C = g\text{-}pow\text{-}PK\text{-}Prod\ PK\ \varphi)$
 $\wedge (C = g\text{-}pow\text{-}PK\text{-}Prod\ PK\ \varphi'));$
 $let\ \alpha = find\text{-}\alpha\ (PK!1)\ (\varphi - \varphi');$
 $return\text{-}spmf\ \alpha\}$

13.2 Helping lemmas

$g^{\phi(\alpha)} = g^{\phi'(\alpha)}$ is equivalent to $(\phi - \phi')(\alpha) = 0$

lemma *commit-eq-is-poly-diff- α -eq-0*:

assumes $degree\ \varphi \leq max\text{-}deg\ degree\ \varphi' \leq max\text{-}deg$
shows $g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (\alpha \hat{t}))\ [0..<max\text{-}deg+1])\ \varphi$
 $= g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (\alpha \hat{t}))\ [0..<max\text{-}deg+1])\ \varphi'$
 $\longleftrightarrow poly\ (\varphi - \varphi')\ \alpha = 0$

proof

assume *commit-eq*:

$g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} \alpha \hat{t})\ [0..<max\text{-}deg + 1])\ \varphi$
 $= g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} \alpha \hat{t})\ [0..<max\text{-}deg + 1])\ \varphi'$
have $acc\text{-}\varphi$: $g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} \alpha \hat{t})\ [0..<max\text{-}deg + 1])\ \varphi$
 $= \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (poly\ \varphi\ \alpha)$

by (*metis g-pow-PK-Prod-correct assms(1)*)

moreover have $acc\text{-}\varphi'$:

$g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} \alpha \hat{t})\ [0..<max\text{-}deg + 1])\ \varphi'$
 $= \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (poly\ \varphi'\ \alpha)$

by (*metis g-pow-PK-Prod-correct assms(2)*)

ultimately show $(poly\ (\varphi - \varphi')\ \alpha = 0)$

using *pow-on-eq-card commit-eq by fastforce*

next

assume *poly-eq-0*: $poly\ (\varphi - \varphi')\ \alpha = 0$

have $acc\text{-}\varphi$: $g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} \alpha \hat{t})\ [0..<max\text{-}deg + 1])\ \varphi$
 $= \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (poly\ \varphi\ \alpha)$

by (*metis g-pow-PK-Prod-correct assms(1)*)

moreover have $acc\text{-}\varphi'$:

$g\text{-}pow\text{-}PK\text{-}Prod\ (map\ (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} \alpha \hat{t})\ [0..<max\text{-}deg + 1])\ \varphi'$

```

    =  $\mathbf{g}_{G_p} \hat{G_p} (\text{poly } \varphi' \alpha)$ 
    by (metis g-pow-PK-Prod-correct assms(2))
    ultimately show g-pow-PK-Prod (map ( $\lambda t. \mathbf{g} \hat{G_p} \alpha \hat{t}$ ) [ $0..<\text{max-deg} + 1$ ])
 $\varphi$ 
        = g-pow-PK-Prod (map ( $\lambda t. \mathbf{g} \hat{G_p} \alpha \hat{t}$ ) [ $0..<\text{max-deg} + 1$ ])  $\varphi'$ 
    using poly-eq-0 by fastforce
qed

```

```

lemma finite-field-factorization-roots:
fixes  $\varphi::'e \text{ mod-ring poly}$ 
  assumes sf-f: square-free  $\varphi$ 
    and us: finite-field-factorization  $\varphi = (c, us)$ 
    shows  $\text{poly } \varphi \alpha = 0 \longleftrightarrow c=0 \vee (\exists u \in \text{set } us. \text{poly } u \alpha = 0)$ 
proof
  assume asm:  $\text{poly } \varphi \alpha = 0$ 
  have smult:  $\varphi = \text{Polynomial.smult } c (\text{prod-list } us)$ 
    using finite-field-factorization-explicit[OF assms] ..
  then show  $c = 0 \vee (\exists u \in \text{set } us. \text{poly } u \alpha = 0)$ 
  proof (cases  $c=0$ )
    case True
    then show ?thesis ..
  next
    case False
    then show ?thesis
      using asm
      by (simp add: poly-prod-list-zero-iff smult)
  qed
next
  assume asm:  $c = 0 \vee (\exists u \in \text{set } us. \text{poly } u \alpha = 0)$ 
  have smult:  $\varphi = \text{Polynomial.smult } c (\text{prod-list } us)$ 
    using finite-field-factorization-explicit[OF assms] ..
  show  $\text{poly } \varphi \alpha = 0$ 
  proof (cases  $c=0$ )
    case True
    then show ?thesis using smult by force
  next
    case False
    then have  $(\exists u \in \text{set } us. \text{poly } u \alpha = 0)$  using asm by blast
    then show ?thesis
      using smult poly-prod-list-zero-iff poly-smult-zero-iff by blast
  qed
qed

```

```

lemma root-imp-deg-1:
  assumes monic ( $u::'e \text{ mod-ring poly}$ )  $\wedge$  irreducible  $u$ 
  shows  $(\exists x. \text{poly } u x = 0) \longleftrightarrow \text{degree } u = 1$ 
proof
  assume asm:  $\exists x. \text{poly } u x = 0$ 
  show  $\text{degree } u = 1$ 

```

```

proof (rule ccontr)
  assume deg-ne-1: degree u  $\neq$  1
  obtain c where c: poly u c = 0 using asm by blast
  with synthetic-div-correct' [of c u] have split-u: u =  $[-c, 1:] * \text{synthetic-div } u$ 
c by simp
  from c deg-ne-1 have deg-u-pos: degree u  $\geq$  2
  by (metis One-nat-def assms leading-coeff-0-iff less-2-cases not-le poly-zero
rel-simps(93))
  then have degree (synthetic-div u c)  $\geq$  1 using degree-synthetic-div[of u c] by
linarith
  then have  $\neg(\text{synthetic-div } u \text{ c}) \text{ dvd } 1$  by auto
  moreover have  $\neg[-c, 1:] \text{ dvd } 1$  by auto
  ultimately show False
  using irreducible-def[of u] split-u assms by blast
qed
next
assume asm: degree u = 1
have poly-deg-1:  $\forall x. \text{poly } u \ x = \text{poly.coeff } u \ 0 + x$ 
proof
  fix x
  have poly u x =  $(\sum i \leq \text{degree } u. \text{poly.coeff } u \ i * x^i)$ 
  using poly-altdef by fast
  also have ... = poly.coeff u 0 + poly.coeff u 1 * x
  using asm by force
  also have ... = poly.coeff u 0 + x
  proof -
    have poly.coeff u 1 = 1
    using asm assms by force
    then show ?thesis by fastforce
  qed
  finally show poly u x = poly.coeff u 0 + x .
qed
show  $\exists x. \text{poly } u \ x = 0$ 
proof
  show poly u ( $-\text{poly.coeff } u \ 0$ ) = 0
  using poly-deg-1 by fastforce
qed
qed

lemma poly-eq0-is-find- $\alpha$ -sf-eq- $\alpha$ :
  assumes  $\varphi \neq 0 \wedge \text{square-free } \varphi$ 
  shows poly  $\varphi \ \alpha = 0 \implies \text{find-}\alpha\text{-square-free } (\mathbf{g}_{G_p} \hat{\ }_{G_p} \alpha) \ \varphi = \alpha$ 
proof -
  assume asm: poly  $\varphi \ \alpha = 0$ 
  obtain c polys where c-polys: (c, polys) = finite-field-factorization  $\varphi$ 
  by (metis prod.exhaust)
  then have c $\neq$ 0 using assms
  by (metis finite-field-factorization-explicit smult-0-left)

```

```

then have  $\exists u \in \text{set polys. } \text{poly } u \ \alpha = 0$  using c-polys asm
  by (metis assms finite-field-factorization-roots)
then obtain u where  $u \in \text{set polys} \wedge \text{poly } u \ \alpha = 0$  by blast
then have  $\text{degree } u = 1$  using root-imp-deg-1
  by (metis (mono-tags, lifting) assms c-polys finite-field-factorization-explicit)
moreover have monic u using u c-polys
  by (metis assms finite-field-factorization-explicit)
ultimately have u-coeff0:  $\text{poly.coeff } u \ 0 = -\alpha$  using u
  by (metis (no-types, lifting) One-nat-def add-0-right coeff-pCons-0 coeff-pCons-Suc
degree1-coeffs degree-1 mpoly-base-conv(2) mult-cancel-left1 one-pCons pCons-0-hom.hom-zero
synthetic-div-correct' synthetic-div-eq-0-iff synthetic-div-pCons)
  then show find- $\alpha$ -square-free ( $\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha$ )  $\varphi = \alpha$ 
    proof -
      have find- $\alpha$ -square-free ( $\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha$ )  $\varphi$ 
        = ( $-\text{filter } (\lambda r. \mathbf{g} \hat{\phantom{g}}_{G_p} \alpha = \mathbf{g} \hat{\phantom{g}}_{G_p} -r) \text{ (map } (\lambda p. \text{poly.coeff } p \ 0) \text{ (filter } (\lambda f. \text{degree } f = 1) \text{ (snd (finite-field-factorization } \varphi)))) \text{ ! } 0$ )
      unfolding find- $\alpha$ -square-free.simps by (simp add: split-def)
      also have  $\dots = (\text{filter } (\lambda r. \mathbf{g} \hat{\phantom{g}}_{G_p} \alpha = \mathbf{g} \hat{\phantom{g}}_{G_p} -r) \text{ (map } (\lambda p. \text{poly.coeff } p \ 0) \text{ (filter } (\lambda f. \text{degree } f = 1) \text{ (polys)))) \text{ ! } 0$ 
        using c-polys by (smt (verit, best) snd-conv)
      also have  $\dots = \alpha$ 
    proof -
      have  $u \in \text{set (filter } (\lambda f. \text{degree } f = 1) \text{ polys)}$ 
        by (simp add:  $\langle \text{degree } u = 1 \rangle u$ )
      then have  $-\alpha \in \text{set (map } (\lambda p. \text{poly.coeff } p \ 0) \text{ (filter } (\lambda f. \text{degree } f = 1) \text{ polys))}$ 
        using u-coeff0  $\langle \text{degree } u = 1 \rangle$  by force
      then have  $-\alpha \in \text{set (filter } (\lambda r. \mathbf{g} \hat{\phantom{g}}_{G_p} \alpha = \mathbf{g} \hat{\phantom{g}}_{G_p} -r) \text{ (map } (\lambda p. \text{poly.coeff } p \ 0) \text{ (filter } (\lambda f. \text{degree } f = 1) \text{ polys))}$ 
        by fastforce
      moreover have  $\forall xs. -\alpha \in \text{set } xs \longrightarrow \text{set (filter } (\lambda r. \mathbf{g} \hat{\phantom{g}}_{G_p} \alpha = \mathbf{g} \hat{\phantom{g}}_{G_p} -r) \text{ xs}) = \{-\alpha\}$ 
        by (auto simp: pow-on-eq-card)
      ultimately have  $\text{set (filter } (\lambda r. \mathbf{g} \hat{\phantom{g}}_{G_p} \alpha = \mathbf{g} \hat{\phantom{g}}_{G_p} -r) \text{ (map } (\lambda p. \text{poly.coeff } p \ 0) \text{ (filter } (\lambda f. \text{degree } f = 1) \text{ polys))}) = \{-\alpha\}$ 
        by fastforce
      then have  $\text{filter } (\lambda r. \mathbf{g} \hat{\phantom{g}}_{G_p} \alpha = \mathbf{g} \hat{\phantom{g}}_{G_p} -r) \text{ (map } (\lambda p. \text{poly.coeff } p \ 0) \text{ (filter } (\lambda f. \text{degree } f = 1) \text{ polys)) \text{ ! } 0 = -\alpha$ 
        by (metis length-pos-if-in-set nth-mem singleton-iff)
      then show ?thesis by force
    qed
  finally show ?thesis .
qed
qed

```

show *find- α* correctly finds(factorizes) α , if α is a root and ϕ is not a zero-polynomial.

lemma *poly-eq0-imp-find- α -eq- α* : $\varphi \neq 0 \implies \text{poly } \varphi \ \alpha = 0 \implies \text{find-}\alpha \ (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha)$

$\alpha) \varphi = \alpha$
proof –
 assume $\varphi\text{-ne-0}$: $\varphi \neq 0$
 assume $\alpha\text{-root}$: $\text{poly } \varphi \alpha = 0$
 have $(\text{radical } \varphi) \neq 0$
 by $(\text{simp add: } \varphi\text{-ne-0})$
 moreover have $\text{poly } (\text{radical } \varphi) \alpha = 0$
 using $\alpha\text{-root same-zeros-radical}$ by blast
 moreover have $\text{square-free } (\text{radical } \varphi)$
 using $\langle (\text{radical } \varphi) \neq 0 \rangle \varphi\text{-ne-0 squarefree-radical squarefree-square-free}$ by blast
 ultimately show $\text{find-}\alpha (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha) \varphi = \alpha$
 unfolding $\text{find-}\alpha.\text{sims}$ using $\text{poly-eq0-is-find-}\alpha\text{-sf-eq-}\alpha$ by blast
qed

13.2.1 Literal helping lemmas

From here on we define literal helping lemmas, with name-pattern: `helping_*number*_*content-reference*`. These lemmas are literal logic blocks that are used in the actual equivalence proof. They all capture one step in the transition (from `poly_bind` game to `t-DL` reduction game logic), that is either too complicated for Isabelle to prove in the monadic/game-based form or requires some additional proving steps that would complicate the equivalence proof. Basically extracted logical reasoning.

The logical addition of $(\phi - \phi')(\alpha) = 0$, which is implied by $\phi(\alpha) = \phi'(\alpha)$, which we derive from $C = g^{\phi(\alpha)} \wedge C = g^{\phi'(\alpha)}$

lemma *helping-1-add-poly- φ -m- φ'* : $(\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi)$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi')$
 $= (\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi)$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi')$
 $\wedge (\text{poly } (\varphi - (\varphi')) (\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) = 0))$
unfolding *valid-poly-def*
using *commit-eq-is-poly-diff- α -eq-0* by fast

lemma *helping-2-factorize- α* : $\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi)$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi')$
 $\wedge (\text{poly } (\varphi - \varphi') (\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) = 0)$
 $\longleftrightarrow \varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$

$\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi)$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi')$
 $\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}))) (\varphi - \varphi') =$
 $(\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}))$
 $(\text{is } ?lhs = ?rhs)$

proof

assume $?lhs$

then show $?rhs$ **using** $\text{poly-eq0-imp-find-}\alpha\text{-eq-}\alpha$

by $(\text{meson right-minus-eq})$

next

assume $?rhs$

then show $?lhs$

unfolding valid-poly-def

using $\text{commit-eq-is-poly-diff-}\alpha\text{-eq-0}$ **by** fast

qed

lemma $\text{helping-}\beta\text{-}\alpha\text{-is-found: } \varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$

$\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi) \wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi')$

$\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}))) (\varphi - \varphi') =$
 $(\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}))$

$\longleftrightarrow \varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$

$\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi) \wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1]) \varphi')$

$\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}))) (\varphi - \varphi') =$
 $(\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}))$

$\wedge \text{of-int-mod-ring } (\text{int } \alpha) = \text{find-}\alpha ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{} t)) [0..<\text{max-deg}+1])!1) (\varphi - \varphi')$

$(\text{is } ?lhs = ?rhs)$

proof –

have $\text{map } (\lambda t::\text{nat. } \mathbf{g}_{G_p} \hat{}_{G_p} \text{of-int-mod-ring } (\text{int } \alpha) \hat{} t) [0::\text{nat}..<\text{max-deg} +$
 $(1::\text{nat})] ! (1::\text{nat}) = \mathbf{g}_{G_p} \hat{}_{G_p} \text{of-int-mod-ring } (\text{int } \alpha)$

using PK-i d-pos **by** force

then show $?thesis$ **by** metis

qed

13.3 KZG poly bind game to strong reduction game - equivalence theorem

showing the equivalence of the KZG poly bind game to the stronger bind_reduction game, which we show to be at least as hard as the real reduction game in the next subsection.

theorem *poly-bind-game-eq-t-DL-strong-red:*
shows *cs.bind-game* $\mathcal{A} = t\text{-DL-}G_p.\text{game}$ (*stronger-bind-reduction* $\mathcal{A}$)
proof –
note $[simp] = \text{Let-def split-def}$

abbreviations for the `mod_ring` version of `sample uniform nat` and the `public key`

let $? \alpha = \lambda \alpha. (\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring})$
let $?PK = \lambda \alpha. (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((? \alpha \alpha) \hat{t})) [0..<\text{max-deg}+1])$

We start with the poly bind game and perform logical transitions until we obtain the t-DL game with the (stronger-)reduction

have *cs.bind-game* $\mathcal{A} = \text{TRY do } \{$
 $(ck, vk) \leftarrow \text{key-gen};$
 $(c, m, d, m', d') \leftarrow \mathcal{A} \text{ } ck;$
 $- :: \text{unit} \leftarrow \text{assert-spmf}(m \neq m' \wedge \text{valid-poly } m \wedge \text{valid-poly } m');$
 $\text{let } b = \text{verify-poly } vk \text{ } c \text{ } d;$
 $\text{let } b' = \text{verify-poly } vk \text{ } m' \text{ } c \text{ } d';$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (b \wedge b');$
 $\text{return-spmf True} \} \text{ ELSE return-spmf False}$
by (*simp add: abstract-commitment.bind-game-alt-def*)
also have $\dots = \text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$
 $\text{let } PK = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring}) \hat{t})) [0..<\text{max-deg}+1];$
 $(C, \varphi, d, \varphi', d') \leftarrow \mathcal{A} \text{ } PK;$
 $- :: \text{unit} \leftarrow \text{assert-spmf}(\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi');$
 $- :: \text{unit} \leftarrow \text{assert-spmf } ((C = g\text{-pow-PK-Prod } PK \text{ } \varphi) \wedge (C = g\text{-pow-PK-Prod } PK \text{ } \varphi'));$
 $\text{return-spmf True} \} \text{ ELSE return-spmf False}$
unfolding *key-gen-def verify-poly-def Setup-def* **by** *simp*
also have $\dots = \text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$
 $\text{TRY do } \{$
 $(C, \varphi, d, \varphi', d') \leftarrow \mathcal{A} \text{ } (?PK \alpha);$
 $\text{TRY do } \{$
 $- :: \text{unit} \leftarrow \text{assert-spmf}(\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi');$
 $- :: \text{unit} \leftarrow \text{assert-spmf } ((C = g\text{-pow-PK-Prod } (?PK \alpha) \text{ } \varphi) \wedge (C = g\text{-pow-PK-Prod } (?PK \alpha) \text{ } \varphi'));$
 return-spmf True
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$
unfolding *split-def*
by(*fold try-bind-spmf-lossless2[OF lossless-return-spmf]*)*simp*
also have $\dots = \text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$
 $\text{TRY do } \{$
 $(C, \varphi, d, \varphi', d') \leftarrow \mathcal{A} \text{ } (?PK \alpha);$
 $\text{TRY do } \{$

```

- :: unit ← assert-spmf ( $\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
 $\wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi'))$ ;
return-spmf True
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
by (presburger add: assert-collapse)
also have ... = TRY do {
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p)$ ;
TRY do {
 $(C, \varphi, d, \varphi', d') \leftarrow \mathcal{A} (?PK \ \alpha)$ ;
TRY do {
- :: unit ← assert-spmf ( $\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
 $\wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi')$ 
 $\wedge (\text{poly } (\varphi - \varphi') (? \alpha \ \alpha) = 0))$ ;
return-spmf True
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
using helping-1-add-poly- $\varphi$ -m- $\varphi'$  by presburger
also have ... = TRY do {
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p)$ ;
TRY do {
 $(C, \varphi, d, \varphi', d') \leftarrow \mathcal{A} (?PK \ \alpha)$ ;
TRY do {
- :: unit ← assert-spmf ( $\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
 $\wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi')$ 
 $\wedge (\text{find-}\alpha \ (\mathbf{g}_{G_p} \hat{\ }_{G_p} ((? \alpha \ \alpha))) (\varphi - \varphi') = (? \alpha \ \alpha))$ ;
return-spmf True
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
using helping-2-factorize- $\alpha$  by presburger
also have ... = TRY do {
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p)$ ;
TRY do {
 $(C, \varphi, d, \varphi', d') \leftarrow \mathcal{A} (?PK \ \alpha)$ ;
TRY do {
- :: unit ← assert-spmf ( $\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
 $\wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \ \alpha) \ \varphi')$ 
 $\wedge (\text{find-}\alpha \ (\mathbf{g}_{G_p} \hat{\ }_{G_p} ((? \alpha \ \alpha))) (\varphi - \varphi') = (? \alpha \ \alpha))$ 
 $\wedge ((? \alpha \ \alpha) = \text{find-}\alpha \ ((?PK \ \alpha)!1) (\varphi - \varphi'))$ ;
return-spmf True
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
using helping-3- $\alpha$ -is-found
by presburger
also have ... = TRY do {

```

```

 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
TRY do {
   $(C, \varphi, -, \varphi', -) \leftarrow \mathcal{A} (?PK \alpha);$ 
  TRY do {
    - ::  $\text{unit} \leftarrow \text{assert-spmf } (\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
       $\wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi')$ 
       $\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((? \alpha \alpha))) (\varphi - \varphi') = (? \alpha \alpha));$ 
    -:: $\text{unit} \leftarrow \text{assert-spmf } ((? \alpha \alpha) = \text{find-}\alpha ((?PK \alpha)!1) (\varphi - \varphi'));$ 
     $\text{return-spmf True } \}$  ELSE  $\text{return-spmf False}$ 
  } ELSE  $\text{return-spmf False}$ 
by (presburger add: assert-collapse)
also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, \varphi, -, \varphi', -) \leftarrow \mathcal{A} (?PK \alpha);$ 
  - ::  $\text{unit} \leftarrow \text{assert-spmf } (\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
     $\wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi')$ 
     $\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((? \alpha \alpha))) (\varphi - \varphi') = (? \alpha \alpha));$ 
  -:: $\text{unit} \leftarrow \text{assert-spmf } ((? \alpha \alpha) = \text{find-}\alpha ((?PK \alpha)!1) (\varphi - \varphi'));$ 
   $\text{return-spmf True } \}$  ELSE  $\text{return-spmf False}$ 
unfolding split-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf])simp
also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, \varphi, -, \varphi', -) \leftarrow \mathcal{A} (?PK \alpha);$ 
  - ::  $\text{unit} \leftarrow \text{assert-spmf } (\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
     $\wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi')$ 
     $\wedge (\text{poly } (\varphi - \varphi') (? \alpha \alpha) = 0));$ 
  -:: $\text{unit} \leftarrow \text{assert-spmf } ((? \alpha \alpha) = \text{find-}\alpha ((?PK \alpha)!1) (\varphi - \varphi'));$ 
   $\text{return-spmf True } \}$  ELSE  $\text{return-spmf False}$ 
using helping-2-factorize- $\alpha$  by presburger
also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, \varphi, -, \varphi', -) \leftarrow \mathcal{A} (?PK \alpha);$ 
  - ::  $\text{unit} \leftarrow \text{assert-spmf } (\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
     $\wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi'));$ 
  -:: $\text{unit} \leftarrow \text{assert-spmf } ((? \alpha \alpha) = \text{find-}\alpha ((?PK \alpha)!1) (\varphi - \varphi'));$ 
   $\text{return-spmf True } \}$  ELSE  $\text{return-spmf False}$ 
using helping-1-add-poly- $\varphi$ -m- $\varphi'$  by presburger
also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $\alpha' \leftarrow \text{do } \{$ 
     $(C, \varphi, -, \varphi', -) \leftarrow \mathcal{A} (?PK \alpha);$ 
    - ::  $\text{unit} \leftarrow \text{assert-spmf } (\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$ 
       $\wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi) \wedge (C = \text{g-pow-PK-Prod } (?PK \alpha) \varphi'));$ 
     $\text{let } \alpha = \text{find-}\alpha ((?PK \alpha)!1) (\varphi - \varphi');$ 
     $\text{return-spmf } \alpha \}$ ;
  -:: $\text{unit} \leftarrow \text{assert-spmf } ((? \alpha \alpha) = \alpha');$ 
   $\text{return-spmf True } \}$  ELSE  $\text{return-spmf False}$ 

```

```

    by fastforce
  also have ... = TRY do {
     $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
     $\alpha' \leftarrow (\text{stronger-bind-reduction } \mathcal{A}) \text{ } (?PK \ \alpha);$ 
     $\text{--::unit} \leftarrow \text{assert-spmf } ((? \alpha \ \alpha) = \alpha');$ 
    return-spmf True } ELSE return-spmf False
  unfolding stronger-bind-reduction.simps
  using g-pow-to-int-mod-ring-of-int-mod-ring-pow-t by presburger
  also have ... = t-DL- $G_p$ .game (stronger-bind-reduction  $\mathcal{A}$ )
  using t-DL- $G_p$ .game-alt-def[of (stronger-bind-reduction  $\mathcal{A}$ )] by simp
  finally show ?thesis .
qed

lemma t-DL-advantage-stronger-red-le-red:
  t-DL- $G_p$ .advantage (stronger-bind-reduction  $\mathcal{A}$ )  $\leq$  t-DL- $G_p$ .advantage (bind-reduction
 $\mathcal{A}$ )
  apply (simp only: t-DL- $G_p$ .advantage-def t-DL- $G_p$ .game-def
    bind-reduction.simps stronger-bind-reduction.simps)
  apply (rule try-spmf-le)
  apply (simp only: bind-spmf-assoc)
  apply (rule bind-spmf-le)+
  apply (simp only: split-def Let-def bind-spmf-assoc bind-return-spmf)
  apply (rule del-assert)
  done

theorem polynomial-binding:
  poly-bind-advantage  $\mathcal{A} \leq$  t-DL- $G_p$ .advantage (bind-reduction  $\mathcal{A}$ )
  apply (simp add: poly-bind-advantage-def cs.bind-advantage-def)
  apply (simp add: poly-bind-game-eq-t-DL-strong-red)
  apply (insert t-DL-advantage-stronger-red-le-red)
  apply (simp add: t-DL- $G_p$ .advantage-def)
  done

end

end

theory KZG-eval-bind

imports KZG-correct tSDH-assumption CryptHOL-ext

begin

```

14 Evaluation Binding of the KZG

In this theory we prove that the KZG is evaluation binding. The proof is a reduction to the t-strong Diffie-Hellman assumption and follows the paper proof by Kate, Zaverucha, and Goldberg in the extended version of “Constant-Size Commitments to Polynomials and Their Applications” [8].

We prove evaluation binding from the abstract PCS for the KZG instantiated as a PCS

locale *KZG-PCS-binding* = *KZG-PCS-correct*
begin

14.1 t-SDH game

We instantiate the t-SDH game for the group G_p

sublocale *t-SDH- G_p* : *t-SDH G_p max-deg of-int-mod-ring $\circ$ int pow-mod-ring G_p*
unfolding *t-SDH-def*
by (*rule G_p .cyclic-group-axioms*)

14.2 Defining a reduction adversary from evaluation binding to t-SDH

The reduction function takes a adversary for the evaluation binding game and returns an adversary for the t-SDH game. Specifically, the reduction uses the evaluation binding adversary to construct a winning strategy for the t-SDH game (i.e. to win it every time). Essentially, it uses the fact that the values supplied by the adversary already break the t-SDH assumption.

fun *eval-bind-reduction*
 $:: ('a\ ck, 'a\ commit, 'e\ mod\text{-}ring, 'e\ mod\text{-}ring, 'a\ witness)\ eval\text{-}bind\text{-}adversary \Rightarrow ('a, 'e)\ t\text{-}SDH.adversary$
where
eval-bind-reduction $\mathcal{A}\ PK$ = *do* {
 $(C, i, \varphi\text{-}of\text{-}i, w\text{-}i, \varphi'\text{-}of\text{-}i, w'\text{-}i) \leftarrow \mathcal{A}\ PK;$
 $return\text{-}spmf\ (-i::'e\ mod\text{-}ring, (w\text{-}i \div_{G_p} w'\text{-}i) \wedge_{G_p} (1 / (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i)))$ }

14.3 Helping definitions

The *eval_bind* reduction adversary extended for asserts that are present in the evaluation binding game. We use this definition to show equivalence to the evaluation binding game. Later on we can then easily over-estimate the probability from this extended version to the normal reduction.

fun *ext-eval-bind-reduction*
 $:: ('a\ ck, 'a\ commit, 'e\ mod\text{-}ring, 'e\ mod\text{-}ring, 'a\ witness)\ eval\text{-}bind\text{-}adversary \Rightarrow ('a, 'e)\ t\text{-}SDH.adversary$
where
ext-eval-bind-reduction $\mathcal{A}\ PK$ = *do* {
 $(C, i, v, w, v', w') \leftarrow \mathcal{A}\ PK;$
 $- :: unit \leftarrow assert\text{-}spmf\ (v \neq v'$
 $\quad \wedge\ valid\text{-}argument\ i$
 $\quad \wedge\ valid\text{-}eval\ (v, w)$
 $\quad \wedge\ valid\text{-}eval\ (v', w')$
 $\quad \wedge\ verify\text{-}eval\ PK\ C\ i\ (v, w)$
 $\quad \wedge\ verify\text{-}eval\ PK\ C\ i\ (v', w'));$

return-spmf $(-i::'e \text{ mod-ring}, (w \div_{G_p} w') \hat{}_{G_p} (1 / (v' - v))) \})$

show that *VerifyEval* on two evaluations, φ -of- i and φ' -of- i , for the same point i , implies that the t-SDH is broken. This lemma captures that the adversaries messages already break the t-SDH assumption.

lemma *two-eval-verify-imp-tSDH-break*:

assumes φ -of- $i \neq \varphi'$ -of- $i \wedge w$ - $i \neq w'$ - i
 $\wedge w$ - $i \in \text{carrier } G_p \wedge w'$ - $i \in \text{carrier } G_p$
 $\wedge \text{verify-eval} ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\alpha) \hat{} t)) [0..<\text{max-deg}+1])) C i (\varphi$ -of- i, w - $i)$
 $\wedge \text{verify-eval} ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\alpha) \hat{} t)) [0..<\text{max-deg}+1])) C i (\varphi'$ -of- i, w' - $i)$
shows $\mathbf{g}_{G_p} \hat{}_{G_p} (1/(\alpha + (-i)))$
 $= (w$ - $i \div_{G_p} w'$ - $i) \hat{}_{G_p} (1 / (\varphi'$ -of- $i - \varphi$ -of- $i))$

proof –

let $?PK = \lambda \alpha. (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\alpha) \hat{} t)) [0..<\text{max-deg}+1])$
obtain ψ_i **where** $\psi_i: w$ - $i = \mathbf{g}_{G_p} \hat{}_{G_p} \psi_i$
by (*metis* G_p .*generatorE* *assms* *g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int*)
obtain ψ_i' **where** $\psi_i': w'$ - $i = \mathbf{g}_{G_p} \hat{}_{G_p} \psi_i'$
by (*metis* G_p .*generatorE* *assms* *g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int*)

the proof is essentially rearranging equations, an outline can be found in the evaluation binding proof section in the thesis paper.

have $e w$ - $i ((\mathbf{g}_{G_p} \hat{}_{G_p} \alpha) \div_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} i)) \otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi$ -of- $i)$
 $= e w'$ - $i ((\mathbf{g}_{G_p} \hat{}_{G_p} \alpha) \div_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} i)) \otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi'$ -of- $i)$

proof –

have $\text{map } (\lambda t. \mathbf{g} \hat{} \alpha \hat{} t) [0..<\text{max-deg} + 1] ! 1 = \mathbf{g} \hat{} \alpha$
by (*metis* PK -*i* *Suc-le-eq* *d-pos numeral-nat*(7) *power-one-right*)
then show *?thesis* **using** *assms* **unfolding** *verify-eval-def* **by** *simp*
qed

then have $e w$ - $i (\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi$ -of- $i)$
 $= e w'$ - $i (\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi'$ -of- $i)$
using *mod-ring-pow-mult-inv- G_p* **by** *presburger*

then have $e (\mathbf{g}_{G_p} \hat{}_{G_p} \psi_i) (\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi$ -of- $i)$
 $= e (\mathbf{g}_{G_p} \hat{}_{G_p} \psi_i') (\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi'$ -of- $i)$

using $\psi_i \psi_i'$ **by** *fast*

then have $(e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} (\psi_i * (\alpha - i) + \varphi$ -of- $i)$
 $= (e \mathbf{g}_{G_p} \mathbf{g}_{G_p}) \hat{}_{G_T} (\psi_i' * (\alpha - i) + \varphi'$ -of- $i)$

using *e-bilinear*

by (*metis* G_p .*generator-closed* *addition-in-exponents-on-e*)

then have $\psi_i * (\alpha - i) + \varphi$ -of- $i = \psi_i' * (\alpha - i) + \varphi'$ -of- i

using *pow-on-eq-card-GT-carrier-ext'*

by *blast*

then have $(\psi_i - \psi_i') * (\alpha - i) = \varphi'$ -of- $i - \varphi$ -of- i

by (*simp* *add: algebra-simps*)

then have $(\psi_i - \psi_i') / (\varphi'$ -of- $i - \varphi$ -of- $i) = 1 / (\alpha - i)$

by (*metis* $\psi_i \psi_i'$ *assms* *divide-divide-eq-left* *divide-self-if eq-iff-diff-eq-0*)

moreover have $(w-i \div_{G_p} w'-i) \hat{G}_p (1 / (\varphi'-of-i - \varphi-of-i)) = \mathbf{g}_{G_p} \hat{G}_p ((\psi_i - \psi_i') / (\varphi'-of-i - \varphi-of-i))$
proof –
have $(w-i \div_{G_p} w'-i) \hat{G}_p (1 / (\varphi'-of-i - \varphi-of-i))$
 $= ((\mathbf{g}_{G_p} \hat{G}_p \psi_i) \div_{G_p} (\mathbf{g}_{G_p} \hat{G}_p \psi_i')) \hat{G}_p (1 / (\varphi'-of-i - \varphi-of-i))$
using $\psi_i \psi_i'$ **by fast**
also have $\dots = (\mathbf{g}_{G_p} \hat{G}_p (\psi_i - \psi_i')) \hat{G}_p (1 / (\varphi'-of-i - \varphi-of-i))$
using *mod-ring-pow-mult-inv- G_p* **by presburger**
also have $\dots = \mathbf{g}_{G_p} \hat{G}_p ((\psi_i - \psi_i') / (\varphi'-of-i - \varphi-of-i))$
by (*metis mod-ring-pow-pow- G_p times-divide-eq-right verit-prod-simplify(2)*)
finally show *?thesis* .
qed
ultimately show *?thesis* **by fastforce**
qed

14.3.1 Literal helping lemmas

CryptHOL has some difficulties with simplifying, thus we need to use literal helping lemmas, that state the equalities we want to exchange literally.

lemma *add-witness-neq-if-eval-neq*: $v \neq v'$
 $\wedge \text{valid-argument } i$
 $\wedge \text{valid-eval } (v, w)$
 $\wedge \text{valid-eval } (v', w')$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..<max-deg+1])$
 $C \ i \ (v, w)$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..<max-deg+1])$
 $C \ i \ (v', w')$
 $\longleftrightarrow$
 $v \neq v' \wedge w \neq w'$
 $\wedge \text{valid-argument } i$
 $\wedge \text{valid-eval } (v, w)$
 $\wedge \text{valid-eval } (v', w')$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..<max-deg+1])$
 $C \ i \ (v, w)$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..<max-deg+1])$
 $C \ i \ (v', w')$
proof
assume *asm*: $v \neq v'$
 $\wedge \text{valid-argument } i$
 $\wedge \text{valid-eval } (v, w)$
 $\wedge \text{valid-eval } (v', w')$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..<max-deg+1]) \ C \ i \ (v, w)$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..<max-deg+1]) \ C \ i \ (v', w')$
have $w \neq w'$
proof –
obtain *w-pow* **where** *w-i-pow*: $w = \mathbf{g}_{G_p} \hat{G}_p \ w\text{-pow}$
proof –

```

have w ∈ carrier Gp
  using asm by (simp add: valid-eval-def)
then show (⋀ w-pow. w = g ^ w-pow ⇒ thesis) ⇒ thesis
  by (metis Gp.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int)
qed
obtain w'-pow where w'-i-pow: w' = g ^Gp w'-pow
proof -
  have w' ∈ carrier Gp
    using asm by (simp add: valid-eval-def)
  then show (⋀ w-pow. w' = g ^ w-pow ⇒ thesis) ⇒ thesis
    by (metis Gp.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int)
  qed

from asm
have e w ((map (λt. gGp ^Gp (α ^ t)) [0..^Gp i))
  ⊗GT e g g ^GT v
=e w' ((map (λt. gGp ^Gp (α ^ t)) [0..^Gp i))
  ⊗GT e g g ^GT v' unfolding verify-eval-def by force
then have e (g ^Gp w-pow) ((gGp ^Gp α) ⊗ inv (g ^Gp i))
  ⊗GT e g g ^GT v
=e (g ^Gp w'-pow) ((gGp ^Gp α) ⊗ inv (g ^Gp i))
  ⊗GT e g g ^GT v'
using PK-i w-i-pow w'-i-pow
using add.commute add-diff-cancel-right' d-pos landau-product-preprocess(52)
length-upt less-diff-conv nth-map nth-upt power-one-right
by auto
then have e g g ^GT (w-pow * (α - i))
  ⊗GT e g g ^GT v
=e g g ^GT (w'-pow * (α - i))
  ⊗GT e g g ^GT v'
by (metis Gp.generator-closed e-bilinear mod-ring-pow-mult-inv-Gp)
then have e g g ^GT (w-pow * (α - i) + v)
  =e g g ^GT (w'-pow * (α - i) + v')
using mod-ring-pow-mult-GT by auto
then have w-pow * (α - i) + v = w'-pow * (α - i) + v'
  using pow-on-eq-card-GT-carrier-ext' by blast
then have w-pow ≠ w'-pow using asm by force

then show ?thesis
  using w-i-pow w'-i-pow pow-on-eq-card by presburger
qed
then show v ≠ v' ∧ w ≠ w'
  ∧ valid-argument i
  ∧ valid-eval (v, w)
  ∧ valid-eval (v', w')
  ∧ verify-eval (map (λt. gGp ^Gp (α ^ t)) [0..Gp ^Gp (α ^ t)) [0..

```

using *asm* **by** *fast*
qed *fast*

lemma *helping-1*: $\varphi\text{-of-}i \neq \varphi'\text{-of-}i \wedge w\text{-}i \neq w'\text{-}i$
 $\wedge \text{valid-argument } i$
 $\wedge w\text{-}i \in \text{carrier } G_p \wedge w'\text{-}i \in \text{carrier } G_p$
 $\wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((\alpha) \hat{t})) [0..<\text{max-deg}+1])) C i (\varphi\text{-of-}i,$
 $w\text{-}i)$
 $\wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((\alpha) \hat{t})) [0..<\text{max-deg}+1])) C i (\varphi'\text{-of-}i,$
 $w'\text{-}i)$
 $\wedge \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} (1/(\alpha + (-i)))$
 $= (w\text{-}i \div_{G_p} w'\text{-}i) \hat{\mathbf{g}}_{G_p} (1 / (\varphi'\text{-of-}i - \varphi\text{-of-}i))$
 $\longleftrightarrow$
 $\varphi\text{-of-}i \neq \varphi'\text{-of-}i \wedge w\text{-}i \neq w'\text{-}i$
 $\wedge \text{valid-argument } i$
 $\wedge w\text{-}i \in \text{carrier } G_p \wedge w'\text{-}i \in \text{carrier } G_p$
 $\wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((\alpha) \hat{t})) [0..<\text{max-deg}+1])) C i (\varphi\text{-of-}i,$
 $w\text{-}i)$
 $\wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((\alpha) \hat{t})) [0..<\text{max-deg}+1])) C i (\varphi'\text{-of-}i,$
 $w'\text{-}i)$
using *two-eval-verify-imp-tSDH-break* **unfolding** *valid-eval-def* **by** *meson*

14.4 Proof

lemma *valid-eval-in-carrier[simp]*: $\text{valid-eval } (v,w) \equiv w \in \text{carrier } G_p$
unfolding *valid-eval-def* **by** *force*

theorem *eval-bind-game-eq-tSDH-strong-ext-red*:
shows *eval-bind-game* $\mathcal{A} = t\text{-SDH-}G_p.\text{game } (\text{ext-eval-bind-reduction } \mathcal{A})$
proof –
note *[simp]* = *Let-def split-def*

abbreviations for the `mod_ring` version of sample uniform nat and the public key

let $? \alpha = \lambda \alpha. (\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring})$
let $?PK = \lambda \alpha. (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} ((? \alpha) \hat{t})) [0..<\text{max-deg}+1])$

have $t\text{-SDH-}G_p.\text{game } (\text{ext-eval-bind-reduction } \mathcal{A}) = \text{TRY do } \{$
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$
 $(C, i, v, w, v', w') \leftarrow \mathcal{A} (?PK \alpha);$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (v \neq v'$
 $\wedge \text{valid-argument } i$
 $\wedge \text{valid-eval } (v,w)$
 $\wedge \text{valid-eval } (v',w')$
 $\wedge \text{verify-eval } (?PK \alpha) C i (v,w)$
 $\wedge \text{verify-eval } (?PK \alpha) C i (v',w'));$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (\mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} (1/(? \alpha \alpha + (-i))) = (w \div_{G_p} w') \hat{\mathbf{g}}_{G_p} (1 /$
 $(v' - v)));$

```

    return-spmf True
  } ELSE return-spmf False
  by (force simp add: t-SDH- $G_p$ .game-alt-def[of (ext-eval-bind-reduction  $\mathcal{A}$ )])

```

Add the fact that witnesses have to be unequal if evaluations are unequal for a easier proof.

```

also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, i, v, w, v', w') \leftarrow \mathcal{A} (?PK \alpha);$ 
  - :: unit  $\leftarrow \text{assert-spmf } (v \neq v' \wedge w \neq w'$ 
     $\wedge \text{valid-argument } i$ 
     $\wedge \text{valid-eval } (v, w)$ 
     $\wedge \text{valid-eval } (v', w')$ 
     $\wedge \text{verify-eval } (?PK \alpha) C i (v, w)$ 
     $\wedge \text{verify-eval } (?PK \alpha) C i (v', w'));$ 
  -::unit  $\leftarrow \text{assert-spmf } (\mathbf{g}_{G_p} \wedge_{G_p} (1 / (? \alpha \alpha + (-i))) = (w \div_{G_p} w') \wedge_{G_p} (1 /$ 
     $(v' - v)))$ ;
  return-spmf True
} ELSE return-spmf False
using add-witness-neq-if-eval-neq by presburger

```

Goal is to erase the second assert statement, such that we just end up with the evaluation_game. To do that, we first merge the two asserts and show that the first assert's statement implies the second one's statement, hence we can leave the second assert's statement out and are left with only the first assert statement.

```

also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, i, v, w, v', w') \leftarrow \mathcal{A} (?PK \alpha);$ 
  - :: unit  $\leftarrow \text{assert-spmf } (v \neq v' \wedge w \neq w'$ 
     $\wedge \text{valid-argument } i$ 
     $\wedge \text{valid-eval } (v, w)$ 
     $\wedge \text{valid-eval } (v', w')$ 
     $\wedge \text{verify-eval } (?PK \alpha) C i (v, w)$ 
     $\wedge \text{verify-eval } (?PK \alpha) C i (v', w')$ 
     $\wedge \mathbf{g}_{G_p} \wedge_{G_p} (1 / (? \alpha \alpha + (-i)))$ 
     $= (w \div_{G_p} w') \wedge_{G_p} (1 / (v' - v)))$ ;
  return-spmf True
} ELSE return-spmf False
by (simp add: assert-collapse)

```

We use the equivalence to erase the assert-term that t-SDH is broken, as it is not contained in the evaluation binding game.

```

also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, i, v, w, v', w') \leftarrow \mathcal{A} (?PK \alpha);$ 
  - :: unit  $\leftarrow \text{assert-spmf } (v \neq v' \wedge w \neq w'$ 
     $\wedge \text{valid-argument } i$ 

```

```

       $\wedge$  valid-eval ( $v, w$ )
       $\wedge$  valid-eval ( $v', w'$ )
       $\wedge$  verify-eval ( $?PK \ \alpha$ )  $C \ i \ (v, w)$ 
       $\wedge$  verify-eval ( $?PK \ \alpha$ )  $C \ i \ (v', w')$ ;
    return-spmf True
  } ELSE return-spmf False
  using helping-1 unfolding valid-eval-in-carrier
  by algebra

```

remove additional assert-term about the witnesses inequality

```

also have ... = TRY do {
   $\alpha \leftarrow$  sample-uniform (order  $G_p$ );
   $(C, i, v, w, v', w') \leftarrow \mathcal{A} \ ?PK \ \alpha$ ;
  - :: unit  $\leftarrow$  assert-spmf ( $v \neq v'$ )
     $\wedge$  valid-argument  $i$ 
     $\wedge$  valid-eval ( $v, w$ )
     $\wedge$  valid-eval ( $v', w'$ )
     $\wedge$  verify-eval ( $?PK \ \alpha$ )  $C \ i \ (v, w)$ 
     $\wedge$  verify-eval ( $?PK \ \alpha$ )  $C \ i \ (v', w')$ ;
  return-spmf True
} ELSE return-spmf False
using add-witness-neq-if-eval-neq[symmetric] by algebra

```

form the KeyGen function from the uniformly sampled alpha

```

also have ... = TRY do {
   $(PK, PK') \leftarrow$  key-gen;
   $(C, i, v, w, v', w') \leftarrow \mathcal{A} \ PK$ ;
  - :: unit  $\leftarrow$  assert-spmf ( $v \neq v'$ )
     $\wedge$  valid-argument  $i$ 
     $\wedge$  valid-eval ( $v, w$ )
     $\wedge$  valid-eval ( $v', w'$ )
     $\wedge$  verify-eval  $PK \ C \ i \ (v, w)$ 
     $\wedge$  verify-eval  $PK \ C \ i \ (v', w')$ ;
  return-spmf True
} ELSE return-spmf False
unfolding key-gen-def Setup-def by simp

```

split the accumulated assert, to obtain the sequence in the evaluation binding game

```

also have ... = TRY do {
   $(PK, -) \leftarrow$  key-gen;
  TRY do {
     $(C, i, v, w, v', w') \leftarrow \mathcal{A} \ PK$ ;
    TRY do {
      - :: unit  $\leftarrow$  assert-spmf ( $v \neq v'$ )
         $\wedge$  valid-argument  $i$ 
         $\wedge$  valid-eval ( $v, w$ )
         $\wedge$  valid-eval ( $v', w'$ )
         $\wedge$  verify-eval  $PK \ C \ i \ (v, w)$ 

```

```

       $\wedge$  verify-eval  $PK\ C\ i\ (v', w')$ );
    return-spmf True
  } ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
unfolding split-def Let-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
also have ... = TRY do {
  ( $PK, -$ )  $\leftarrow$  key-gen;
  TRY do {
    ( $C, i, v, w, v', w'$ )  $\leftarrow$   $\mathcal{A}\ PK$ ;
    TRY do {
      - :: unit  $\leftarrow$  assert-spmf ( $v \neq v' \wedge$  valid-argument  $i \wedge$  valid-eval ( $v, w$ )  $\wedge$ 
valid-eval ( $v', w'$ ));
      - :: unit  $\leftarrow$  assert-spmf (verify-eval  $PK\ C\ i\ (v, w) \wedge$  verify-eval  $PK\ C\ i\ (v', w')$ );
    }
    return-spmf True
  } ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
by (simp add: assert-collapse)
also have ... = TRY do {
  ( $PK, -$ )  $\leftarrow$  key-gen;
  TRY do {
    ( $C, i, v, w, v', w'$ )  $\leftarrow$   $\mathcal{A}\ PK$ ;
    TRY do {
      - :: unit  $\leftarrow$  assert-spmf ( $v \neq v' \wedge$  valid-argument  $i \wedge$  valid-eval ( $v, w$ )  $\wedge$ 
valid-eval ( $v', w'$ ));
      TRY do {
        let  $b =$  verify-eval  $PK\ C\ i\ (v, w)$ ;
        let  $b' =$  verify-eval  $PK\ C\ i\ (v', w')$ ;
        return-spmf ( $b \wedge b'$ )
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False
} ELSE return-spmf False
by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong
bind-spmf-cong)
also have ... = TRY do {
  ( $PK, -$ )  $\leftarrow$  key-gen;
  ( $C, i, v, w, v', w'$ )  $\leftarrow$   $\mathcal{A}\ PK$ ;
  - :: unit  $\leftarrow$  assert-spmf ( $v \neq v' \wedge$  valid-argument  $i \wedge$  valid-eval ( $v, w$ )  $\wedge$  valid-eval
( $v', w'$ ));
  let  $b =$  verify-eval  $PK\ C\ i\ (v, w)$ ;
  let  $b' =$  verify-eval  $PK\ C\ i\ (v', w')$ ;
  return-spmf ( $b \wedge b'$ ) } ELSE return-spmf False
unfolding split-def Let-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
also have ... = eval-bind-game  $\mathcal{A}$ 

```

```

    using eval-bind-game-def unfolding key-gen-def Setup-def by auto
    finally show ?thesis ..
qed

```

```

lemma overestimate-reductions:  $\text{spmf } (t\text{-SDH-}G_p.\text{game } (\text{ext-eval-bind-reduction } \mathcal{A}))$ 
 $\text{True}$ 
 $\leq \text{spmf } (t\text{-SDH-}G_p.\text{game } (\text{eval-bind-reduction } \mathcal{A})) \text{ True}$ 
 $(\text{is\_spmf\_?lhgame True} \leq \text{spmf\_?rhgame True})$ 
proof –
  note [simp] = Let-def split-def

```

abbreviations for the mod_ring version of sample uniform nat and the public key

```

let ? $\alpha$  =  $\lambda\alpha. (\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring})$ 
let ?PK =  $\lambda\alpha. (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} ((? \alpha \text{ } \alpha) \hat{t})) [0..<\text{max-deg}+1])$ 

```

We extend the t-SDH game with reduction adversary to a complete game.

```

have bind-red-ext:  $t\text{-SDH-}G_p.\text{game } (\text{ext-eval-bind-reduction } \mathcal{A}) = \text{TRY do } \{$ 
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
 $(C, i, v, w, v', w') \leftarrow \mathcal{A} (?PK \alpha);$ 
 $- :: \text{unit} \leftarrow \text{assert-spmf } (v \neq v'$ 
 $\quad \wedge \text{valid-argument } i$ 
 $\quad \wedge \text{valid-eval } (v, w)$ 
 $\quad \wedge \text{valid-eval } (v', w')$ 
 $\quad \wedge \text{verify-eval } (?PK \alpha) C i (v, w)$ 
 $\quad \wedge \text{verify-eval } (?PK \alpha) C i (v', w'));$ 
 $--:: \text{unit} \leftarrow \text{assert-spmf } (\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (1 / (? \alpha \alpha + (-i))) = (w \div_{G_p} w') \hat{\phantom{x}}_{G_p} (1 /$ 
 $(v' - v))));$ 
 $\text{return-spmf True}$ 
 $\}$  ELSE  $\text{return-spmf False}$ 
by (force simp add:  $t\text{-SDH-}G_p.\text{game-alt-def}[of \text{ } (\text{ext-eval-bind-reduction } \mathcal{A})])$ 

```

We extend the t-SDH game with reduction adversary to a complete game.

```

have eval-bind-red-ext:  $t\text{-SDH-}G_p.\text{game } (\text{eval-bind-reduction } \mathcal{A}) = \text{TRY do } \{$ 
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
 $(C, i, \varphi\text{-of-}i, w\text{-}i, \varphi'\text{-of-}i, w'\text{-}i) \leftarrow \mathcal{A} (?PK \alpha);$ 
 $--:: \text{unit} \leftarrow \text{assert-spmf } (\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (1 / (? \alpha \alpha + (-i))) = (w\text{-}i \div_{G_p} w'\text{-}i) \hat{\phantom{x}}_{G_p} (1$ 
 $/ (\varphi'\text{-of-}i - \varphi\text{-of-}i))));$ 
 $\text{return-spmf True}$ 
 $\}$  ELSE  $\text{return-spmf False}$ 
by (force simp add:  $t\text{-SDH-}G_p.\text{game-alt-def}[of \text{ } (\text{eval-bind-reduction } \mathcal{A})])$ 

```

We show the thesis in ennreal, which implies the plain thesis

```

have ennreal ( $\text{spmf } (t\text{-SDH-}G_p.\text{game } (\text{ext-eval-bind-reduction } \mathcal{A})) \text{ True}$ )
 $\leq \text{ennreal } (\text{spmf } (t\text{-SDH-}G_p.\text{game } (\text{eval-bind-reduction } \mathcal{A})) \text{ True})$ 
unfolding bind-red-ext eval-bind-red-ext
apply (simp add:  $\text{spmf-try-spmf ennreal-spmf-bind}$ )
apply (rule nn-integral-mono)+

```

```

    apply (simp add: assert-spmf-def)
    apply (simp add: measure-spmf.emmeasure-eq-measure)
  done

  then show ?thesis by simp
qed

Finally we put everything together: we conclude that for every efficient adversary the advantage of winning the evaluation binding game is less equal to breaking the t-SDH assumption.

theorem evaluation-binding: eval-bind-advantage  $\mathcal{A} \leq t\text{-SDH-}G_p.\text{advantage}$  (eval-bind-reduction  $\mathcal{A}$ )
  using overestimate-reductions
  unfolding t-SDH- $G_p.\text{advantage-def}$  eval-bind-advantage-def eval-bind-game-eq-t-SDH-strong-ext-red
  by fast

end

end

theory KZG-knowledge-sound

imports KZG-eval-bind Algebraic-Group-Model

begin

```

15 Knowledge Soundness of the KZG

In this theory we prove knowledge soundness for the KZG, concretely the knowledge soundness as defined in the abstract polynomial commitment scheme. The proof is a reduction to the evaluation binding game which has been reduced to the t-strong Diffie-Hellman problem in the KZG_eval_bind theory.

```

hide-const restrict

locale KZG-PCS-knowledge-sound = KZG-PCS-binding
begin

the AGM adversary types that are useful in defining reductions (i.e. the reduction to the evaluation binding game)

lift-to-algebraicT ('a ck, 'a commit, 'state) knowledge-soundness-adversary1  $G_p$ 

=> AGM-knowledge-soundness-adversary1
lift-to-algebraicT ('state, 'a ck, 'e mod-ring, 'e evaluation, 'a witness) knowledge-soundness-adversary2
 $G_p$  => AGM-knowledge-soundness-adversary2

type-synonym ('e, 'state, 'a) knowledge-soundness-adversary2-AGM

```

$$= ('a' \text{ ck} \times 'state') \Rightarrow ('e' \text{ argument} \times ('e' \text{ evaluation} \times ('a' \text{ witness} \times \text{int list}))) \\ \text{spmf}$$

The extractor is an algorithm that plays against the adversary. It is granted access to the adversaries messages and state (which we neglect in this case as we do not need it because the calculation vector is enough to create sensible values) and has to come up with a polynomial such that the adversary cannot create valid opening points that are not part of the polynomial.

type-synonym ('a', 'e') *extractor* =
 ('a' *commit* $\times$ *int list*) $\Rightarrow$
 ('e' *mod-ring poly* $\times$ *unit*) *spmf*

restrict for AGM adversaries 1 and 2

realized by the following two interpretations:

interpretation *AGM1: Algebraic-Algorithm* G_p *listS* $G_p.\text{groupS}$ *prodC* $G_p.\text{groupC}$ *noConstrain*
by (*unfold-locals*)

interpretation *AGM2: Algebraic-Algorithm* G_p *prodS* (*listS* $G_p.\text{groupS}$) *noSelect*

prodC noConstrain (*prodC noConstrain* $G_p.\text{groupC}$)
by (*unfold-locals*)

definition *knowledge-soundness-game-AGM* :: ('state', 'a') *AGM-knowledge-soundness-adversary1*

$\Rightarrow ('e, 'state, 'a) \text{ knowledge-soundness-adversary2-AGM} \Rightarrow ('a, 'e) \text{ extractor} \Rightarrow$
bool spmf
where *knowledge-soundness-game-AGM* $\mathcal{A}1 \mathcal{A}2 \mathcal{E} = \text{TRY do } \{$
 (*ck, vk*) $\leftarrow$ *key-gen*;
 (*c, cvec*), $\sigma \leftarrow$ *AGM1.restrict* $\mathcal{A}1$ *ck*;
 (*p, td*) $\leftarrow$ $\mathcal{E}$ (*c, cvec*);
 (*i, p-i, w, wvec*) $\leftarrow$ *AGM2.restrict* $\mathcal{A}2$ (*ck, \sigma*);
 let (*p-i', w'*) = *Eval ck td p i*;
 return-spmf (*verify-eval vk c i (p-i, w) $\wedge$ p-i $\neq$ p-i' $\wedge$ valid-argument i $\wedge$*
 valid-eval (p-i, w))
 } *ELSE return-spmf False*

reduction to the evaluation bind game The main idea is that if the adversary can break knowledge soundness, i.e. give an evaluation (+ proof) that differs from the evaluation of the polynomial provided by the extractor, the evaluation of the polynomial provided by the extractor will still yield a valid evaluation (+ proof). Hence, one obtains two distinct valid evaluations of the same value, thus breaking evaluation binding.

definition *knowledge-soundness-reduction*

:: ('a', 'e') *extractor* $\Rightarrow ('state, 'a) \text{ AGM-knowledge-soundness-adversary1}$
 $\Rightarrow ('e, 'state, 'a) \text{ knowledge-soundness-adversary2-AGM}$

$\Rightarrow ('a\ ck, 'a\ commit, 'e\ argument, 'e\ evaluation, 'a\ witness)\ \text{eval-bind-adversary}$

where

```
knowledge-soundness-reduction  $\mathcal{E}\ \mathcal{A}1\ \mathcal{A}2\ ck = do \{$ 
   $((c, cvec), \sigma) \leftarrow AGM1.restrict\ \mathcal{A}1\ ck;$ 
   $(p, td) \leftarrow \mathcal{E}\ (c, cvec);$ 
   $(i, p-i, w, wvec) \leftarrow AGM2.restrict\ \mathcal{A}2\ (ck, \sigma);$ 
   $let\ (p-i', w') = Eval\ ck\ td\ p\ i;$ 
   $return-spmf\ (c, i, p-i, w, p-i', w')\}$ 
```

Extractor definition

fun $E :: ('a, 'e)\ \text{extractor}\ \text{where}$

$E\ (c, cvec) = return-spmf\ (Poly\ (map\ (of-int-mod-ring::int \Rightarrow 'e\ mod-ring)\ cvec), ())$

15.1 Helping definitions

The knowledge soundness reduction adversary extended for asserts that are present in the evaluation binding game. We use this definition to show equivalence to the evaluation binding game. Later on we can then easily over-estimate the probability from this extended version to the normal reduction.

definition *knowledge-soundness-reduction-ext*

```
:: ('a, 'e)\ extractor  $\Rightarrow ('state, 'a)\ AGM\text{-}knowledge\text{-}soundness\text{-}adversary1$ 
 $\Rightarrow ('e, 'state, 'a)\ knowledge\text{-}soundness\text{-}adversary2\text{-}AGM$ 
 $\Rightarrow ('a\ ck, 'a\ commit, 'e\ argument, 'e\ evaluation, 'a\ witness)\ \text{eval-bind-adversary}$ 
```

where

```
knowledge-soundness-reduction-ext  $\mathcal{E}\ \mathcal{A}1\ \mathcal{A}2\ ck = do \{$ 
   $((c, cvec), \sigma) \leftarrow AGM1.restrict\ \mathcal{A}1\ ck;$ 
   $(p, td) \leftarrow \mathcal{E}\ (c, cvec);$ 
   $(i, p-i, w, wvec) \leftarrow AGM2.restrict\ \mathcal{A}2\ (ck, \sigma);$ 
   $- :: unit \leftarrow assert-spmf\ (valid-eval\ (p-i, w));$ 
   $let\ (p-i', w') = Eval\ ck\ td\ p\ i;$ 
   $return-spmf\ (c, i, p-i, w, p-i', w')\}$ 
```

15.2 Helping lemmas

proof related helping lemmas

lemma *ks-imp-eval-bind-asserts:*

```
let  $ck = map\ (\lambda t. \mathbf{g}_{G_p} \widehat{\alpha} t) [0..<max-deg+1];$ 
 $vk = map\ (\lambda t. \mathbf{g}_{G_p} \widehat{\alpha} t) [0..<max-deg+1];$ 
 $(p-i', w') = Eval\ ck\ td\ (Poly\ (map\ of-int-mod-ring\ cvec))\ i$ 
in
 $length\ ck = length\ (cvec::int\ list)$ 
 $\wedge\ c = fold\ (\lambda\ i\ acc.\ acc \otimes ck!i\ [\neg]\ (cvec!i))\ [0..<length\ ck]\ \mathbf{1}$ 
 $\wedge\ verify-eval\ vk\ c\ i\ (p-i, w)$ 
 $\wedge\ p-i \neq p-i'$ 
```

```

    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
  ↔
    length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
    ∧ valid-eval (p-i', w')
    ∧ verify-eval vk c i (p-i, w)
    ∧ verify-eval vk c i (p-i', w')
proof -
  define ck where ck-def: ck = map (λt. gGp ⌈Gp (α⌈t)) [0..<max-deg+1]
  define vk where vk-def: vk = map (λt. gGp ⌈Gp (α⌈t)) [0..<max-deg+1]
  define p-i' where p-i'-def: p-i' = fst (Eval ck td (Poly (map of-int-mod-ring
cvec)) i)
  define w' where w'-def: w' = snd (Eval ck td (Poly (map of-int-mod-ring cvec))
i)

  have length ck = length (cvec::int list)
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ verify-eval vk c i (p-i,w)
    ∧ p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
  ↔
    length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
    ∧ valid-eval (p-i', w')
    ∧ verify-eval vk c i (p-i, w)
    ∧ verify-eval vk c i (p-i', w')
(is ?lhs ↔ ?rhs)
proof
  assume asm: ?lhs
  show ?rhs
proof(intro conjI)
  from asm show valid-eval-adv: valid-eval (p-i, w) by force
  from asm show verify-eval vk c i (p-i, w) by force

  show valid-eval-gen: valid-eval (p-i', w')
proof -
  have g-pow-PK-Prod ck (ψ-of (Poly (map of-int-mod-ring cvec)) i)
  = gGp ⌈Gp (poly (ψ-of (Poly (map of-int-mod-ring cvec)) i) α)
  unfolding ck-def
proof (rule g-pow-PK-Prod-correct)
  show degree (ψ-of (Poly (map of-int-mod-ring cvec)) i) ≤ max-deg

```

```

proof (rule le-trans[OF degree-q-le-φ])
  have length (map of-int-mod-ring cvec) = max-deg + 1
  using asm unfolding ck-def by force
  moreover have length (coeffs (Poly (map of-int-mod-ring cvec))) ≤
length (map of-int-mod-ring cvec)
  by (metis coeffs-Poly length-map length-strip-while-le)
  ultimately show degree (Poly (map of-int-mod-ring cvec)) ≤ max-deg
  using degree-eq-length-coeffs[of Poly (map of-int-mod-ring cvec)]
  by (metis le-diff-conv)
qed
qed
then show ?thesis
  unfolding valid-eval-def
  by (simp add: Eval-def p-i'-def w'-def)
qed

show verify-eval-gen: verify-eval vk c i (p-i', w')
proof -
  let ?cvec = (map of-int-mod-ring cvec::'e mod-ring list)

  have length-cvec: length ?cvec = max-deg + 1
  using asm unfolding ck-def by force
  moreover have length (coeffs (Poly ?cvec)) ≤ length ?cvec
  by (metis coeffs-Poly length-strip-while-le)
ultimately have deg-poly-calc-vec-le-max-deg: degree (Poly ?cvec) ≤ max-deg
  using degree-eq-length-coeffs[of Poly ?cvec]
  by (metis coeffs-Poly le-diff-conv length-strip-while-le)

  have 1: (g-pow-PK-Prod (map (λt. g ^Gp α ^ t) [0..Gp poly (ψ-of (Poly ?cvec) i) α)
  proof(rule g-pow-PK-Prod-correct)
    show degree (ψ-of (Poly ?cvec) i) ≤ max-deg
    by (rule le-trans[OF degree-q-le-φ])(fact deg-poly-calc-vec-le-max-deg)
  qed

  have 2: map (λt. g ^Gp α ^ t) [0..Gp α
    by (metis (no-types, lifting) One-nat-def add.commute d-pos diff-zero
le-add-same-cancel1 le-zero-eq length-upt nth-map nth-upt plus-1-eq-Suc power-one-right
zero-compare-simps(1))

  have 3: (g ^Gp poly (Poly ?cvec) α) = c
  proof -
    have (g ^Gp poly (Poly ?cvec) α)
      = g-pow-PK-Prod (map (λt. g ^Gp α ^ t) [0..by (rule g-pow-PK-Prod-correct[symmetric])(fact deg-poly-calc-vec-le-max-deg)
    also have g-pow-to-fold: ... = fold (λi acc. acc ⊗Gp (gGp ^Gp (α ^ i))
^Gp (poly.coeff (Poly ?cvec) i))

```

```

[0..Suc (degree (Poly ?cvec))] 1Gp
  by (rule g-pow-PK-Prod-to-fold)(fact deg-poly-calc-vec-le-max-deg)
  also have ...
= fold (λ i acc. acc ⊗Gp (gGp ^Gp (α ^ i)) ^Gp (?cvec!i)) [0..max-deg+1]
1Gp
  proof -
    have fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp ?cvec ! i) [0..max-deg +
1] 1
      = fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp ?cvec ! i)
      ([0..Suc (degree (Poly ?cvec))] @ [Suc (degree (Poly ?cvec))..max-deg
+ 1])
      1
    proof -
      have Suc (degree (Poly ?cvec)) ≤ max-deg + 1
      by (simp add: deg-poly-calc-vec-le-max-deg)
      then show ?thesis
      by (metis (lifting) nat-le-iff-add upt-add-eq-append zero-order(1))
    qed
    also have ... = fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp ?cvec ! i)
      [Suc (degree (Poly ?cvec))..max-deg + 1]
      (fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp ?cvec ! i)
      [0..Suc (degree (Poly ?cvec))] 1)
      by fastforce
    also have ... = fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp poly.coeff (Poly
?cvec) i)
      [0..Suc (degree (Poly ?cvec))]
      1
    proof -
      have fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp ?cvec ! i) [0..Suc (degree
(Poly ?cvec))] 1
      = fold (λ i acc. acc ⊗ (g ^Gp α ^ i) ^Gp poly.coeff (Poly ?cvec) i)
[0..Suc (degree (Poly ?cvec))] 1
    proof (rule List.fold-cong)
      show ∧ x. x ∈ set [0..Suc (degree (Poly ?cvec))] ⇒
      (λ acc. acc ⊗ (g ^Gp α ^ x) ^Gp ?cvec ! x) =
      (λ acc. acc ⊗ (g ^Gp α ^ x) ^Gp poly.coeff (Poly ?cvec) x)
    proof
      fix x::nat
      fix acc::'a
      assume asm: x ∈ set [0..Suc (degree (Poly ?cvec))]
      then have ?cvec ! x = poly.coeff (Poly ?cvec) x
      by (metis ⟨length ?cvec = max-deg + 1⟩ atLeastLessThan-iff co-
eff-Poly deg-poly-calc-vec-le-max-deg dual-order.trans less-Suc-eq-le nth-default-nth
semiring-norm(174) set-upt)
      then show acc ⊗ (g ^Gp α ^ x) ^Gp ?cvec ! x = acc ⊗ (g ^Gp α
^ x) ^Gp poly.coeff (Poly ?cvec) x
      by presburger
    qed
  qed

```

```

qed simp+
moreover have  $\forall \text{init} \in \text{carrier } G_p.$ 
   $\text{fold } (\lambda i \text{ acc. acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} i) \hat{\wedge}_{G_p} ?cvec ! i)$ 
     $[\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} + 1]$ 
     $\text{init}$ 
   $= \text{init}$ 
proof
  fix  $\text{init} :: 'a$ 
  assume  $\text{init-in-carrier: init} \in \text{carrier } G_p$ 
  have  $\text{fold } (\lambda i \text{ acc. acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} i) \hat{\wedge}_{G_p} ?cvec ! i)$ 
     $[\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} + 1]$ 
     $\text{init} = \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1})$ 
     $[\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} + 1]$ 
     $\text{init}$ 
  proof (rule List.fold-cong)
    show  $\bigwedge x. x \in \text{set } [\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} + 1] \implies$ 
       $(\lambda \text{acc. acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} x) \hat{\wedge}_{G_p} ?cvec ! x) = (\lambda \text{acc. acc} \otimes \mathbf{1})$ 
    proof
      fix  $x::\text{nat}$ 
      fix  $\text{acc} :: 'a$ 
      assume  $\text{asm: } x \in \text{set } [\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} + 1]$ 
      show  $\text{acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} x) \hat{\wedge}_{G_p} ?cvec ! x = \text{acc} \otimes \mathbf{1}$ 
      proof -
        have  $?cvec ! x = 0$  using asm length-cvec
        by (smt (verit) add.commute coeff-Poly-eq in-set-conv-nth
le-degree length-upt less-diff-conv not-less-eq-eq nth-default-eq-dflt-iff nth-upt or-
der.refl trans-le-add2)
        then have  $(\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} x) \hat{\wedge}_{G_p} ?cvec ! x = \mathbf{1}$  by simp
        then show ?thesis by argo
      qed
    qed
  qed simp+
  also have  $\dots = \text{init}$ 
  proof (induction max-deg)
    case 0
    then show ?case by fastforce
  next
    case (Suc max-deg)
    have  $\text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1}) [\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{Suc max-deg}$ 
     $+ 1] \text{ init}$ 
     $= \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1}) ([\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} +$ 
     $1] @ [\text{Suc max-deg}]) \text{ init}$ 
    by (simp add: init-in-carrier)
    also have  $\dots = \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1}) [\text{Suc max-deg}] (\text{fold } (\lambda i \text{ acc.}$ 
     $\text{acc} \otimes \mathbf{1}) [\text{Suc } (\text{degree } (\text{Poly } ?cvec))..<\text{max-deg} + 1] \text{ init})$ 
    by force
    also have  $\dots = \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1}) [\text{Suc max-deg}] \text{init}$  using
Suc.IH by argo
    also have  $\dots = \text{init} \otimes \mathbf{1}$  by force

```

```

    also have ... = init by (simp add: init-in-carrier)
    finally show ?case .
  qed
  finally show fold (λi acc. acc ⊗ (g  $\hat{\phantom{x}}$ Gp α  $\hat{\phantom{x}}$  i)  $\hat{\phantom{x}}$ Gp ?cvec ! i)
    [Suc (degree (Poly ?cvec))..<max-deg + 1]
    init
  = init .
  qed
  ultimately show ?thesis
    by (metis (no-types, lifting) Gp.generator-closed Gp.int-pow-closed ⟨g
 $\hat{\phantom{x}}$ Gp poly (Poly ?cvec) α = g-pow-PK-Prod (map (λt. g  $\hat{\phantom{x}}$ Gp α  $\hat{\phantom{x}}$  t) [0..<max-deg
+ 1]) (Poly ?cvec)⟩ g-pow-to-fold)
  qed
  finally show ?thesis by presburger
  qed
  also have ...
    = fold (λ i acc. acc ⊗Gp (map (λt. gGp  $\hat{\phantom{x}}$ Gp (α  $\hat{\phantom{x}}$  t)) [0..<max-deg+1])!i
 $\hat{\phantom{x}}$ Gp (?cvec!i)) [0..<max-deg+1] 1Gp
  proof(rule List.fold-cong)
    show 1 = 1 by simp
    show [0..<max-deg + 1] = [0..<max-deg + 1] by simp
    show  $\bigwedge x. x \in \text{set } [0..<max-deg + 1] \implies$ 
      (λacc. acc ⊗ (g  $\hat{\phantom{x}}$ Gp α  $\hat{\phantom{x}}$  x)  $\hat{\phantom{x}}$ Gp ?cvec ! x) =
      (λacc. acc ⊗ map (λt. g  $\hat{\phantom{x}}$ Gp α  $\hat{\phantom{x}}$  t) [0..<max-deg + 1] ! x  $\hat{\phantom{x}}$ Gp ?cvec !
x)
  proof
    fix x::nat
    fix acc :: 'a
    assume asm: x ∈ set [0..<max-deg + 1]
    show acc ⊗ (g  $\hat{\phantom{x}}$ Gp α  $\hat{\phantom{x}}$  x)  $\hat{\phantom{x}}$ Gp ?cvec ! x
      = acc ⊗ map (λt. g  $\hat{\phantom{x}}$ Gp α  $\hat{\phantom{x}}$  t) [0..<max-deg + 1] ! x  $\hat{\phantom{x}}$ Gp ?cvec ! x
    using PK-i[symmetric] asm
    by (metis Suc-eq-plus1 atLeastLessThan-iff less-Suc-eq-le set-upt)
  qed
  qed
  also have ...
    = fold (λ i acc. acc ⊗Gp (map (λt. gGp  $\hat{\phantom{x}}$ Gp (α  $\hat{\phantom{x}}$  t)) [0..<max-deg+1])!i
 $\hat{\phantom{x}}$ Gp (of-int-mod-ring (cvec!i))) [0..<max-deg+1] 1Gp
  proof(rule List.fold-cong)
    fix x
    assume x ∈ set [0..<max-deg + 1]
    then have x < length cvec
      using asm unfolding ck-def
      by fastforce
    then show (λacc. acc ⊗ map (λt. g  $\hat{\phantom{x}}$  α  $\hat{\phantom{x}}$  t) [0..<max-deg + 1] ! x  $\hat{\phantom{x}}$ 
map of-int-mod-ring cvec ! x) =
      (λacc. acc ⊗ map (λt. g  $\hat{\phantom{x}}$  α  $\hat{\phantom{x}}$  t) [0..<max-deg + 1] ! x  $\hat{\phantom{x}}$  of-int-mod-ring
(cvec ! x))

```

```

      by force
    qed simp+
    also have ... = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    proof -
      have length-eq-max-deg: length (map (λt. g ^ α ^ t) [0..<max-deg + 1])
= max-deg + 1
      by force
      have mod-ring-trnsf-eq-plain: ⋀g x. g ∈ carrier Gp ⇒ g [⌈Gp
(to-int-mod-ring (of-int-mod-ring x::'e mod-ring)) = g [⌈Gp x
    proof -
      fix g x
      assume g-in-carrier: g ∈ carrier Gp
      have mod-red: to-int-mod-ring (of-int-mod-ring x::'e mod-ring) = x mod
p
      unfolding of-int-mod-ring-def to-int-mod-ring-def
      by (metis CARD-q of-int-mod-ring.rep-eq of-int-mod-ring-def
to-int-mod-ring.rep-eq to-int-mod-ring-def)
      then show g [⌈Gp (to-int-mod-ring (of-int-mod-ring x::'e mod-ring))
= g [⌈Gp x
      using carrier-pow-mod-order-Gp g-in-carrier mod-red by metis
    qed
    show ?thesis
    proof(rule List.fold-cong)
      fix x
      assume x ∈ set [0..<max-deg + 1]
      then show (λacc. acc ⊗ map (λt. g ^ α ^ t) [0..<max-deg + 1] ! x ^
of-int-mod-ring (cvec ! x)) = (λacc. acc ⊗ ck ! x [⌈ cvec ! x)
      unfolding ck-def length-eq-max-deg using mod-ring-trnsf-eq-plain
      by (metis (no-types, lifting) Gp.generator-closed Gp.int-pow-closed
atLeastLessThan-iff length-upt nth-map set-upt verit-minus-simplify(2))
    qed (simp add: ck-def)+
    qed
    also have ... = c
      using asm unfolding ck-def by fast
    finally show ?thesis .
  qed
  show ?thesis
  unfolding verify-eval-def Eval-def Let-def split-def g-pow-PK-Prod-correct
  using eq-on-e[of (Poly ?cvec) i α]
  by (metis 1 2 3 Eval-def ck-def vk-def p-i'-def w'-def eq-on-e fst-conv
snd-conv)
  qed
  qed (force simp add: asm)+
next
  assume asm: ?rhs
  show ?lhs
  proof(intro conjI)
    from asm show valid-eval (p-i, w) by force
    from asm show verify-eval vk c i (p-i, w) by force

```

```

    qed (simp add: asm)+
  qed
  then show ?thesis
    unfolding ck-def vk-def p-i'-def w'-def Let-def split-def by fast
  qed

lemma knowledge-soundness-game-alt-def:
  knowledge-soundness-game-AGM  $\mathcal{A}1$   $\mathcal{A}2$   $E$  =
  eval-bind-game (knowledge-soundness-reduction-ext  $E$   $\mathcal{A}1$   $\mathcal{A}2$ )
proof -
  note [simp] = Let-def split-def

  have knowledge-soundness-game-AGM  $\mathcal{A}1$   $\mathcal{A}2$   $E$  =
    TRY do {
      (ck,vk)  $\leftarrow$  key-gen;
      ((c,cvec), $\sigma$ )  $\leftarrow$  AGM1.restrict  $\mathcal{A}1$  ck;
      (p,td)  $\leftarrow$   $E$  (c,cvec);
      (i, p-i, w, wvec)  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
      let (p-i',w') = Eval ck td p i;
      return-spmf (verify-eval vk c i (p-i,w)  $\wedge$  p-i  $\neq$  p-i'  $\wedge$  valid-argument i  $\wedge$ 
        valid-eval (p-i,w))
    } ELSE return-spmf False
  by (simp add: knowledge-soundness-game-AGM-def del: Let-def split-def)
  also have ... =
    TRY do {
      (ck,vk)  $\leftarrow$  key-gen;
      ((c,cvec), $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  ck;
      - :: unit  $\leftarrow$  assert-spmf (length ck = length cvec
         $\wedge$  c = fold ( $\lambda$  i acc. acc  $\otimes$  ck!i [ $\top$ ] (cvec!i)) [0..1);
      (p,td)  $\leftarrow$   $E$  (c,cvec);
      (i, p-i, (w, wvec))  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
      let (p-i',w') = Eval ck td p i;
      return-spmf (verify-eval vk c i (p-i,w)  $\wedge$  p-i  $\neq$  p-i'  $\wedge$  valid-argument i  $\wedge$ 
        valid-eval (p-i,w))
    } ELSE return-spmf False
  unfolding AGM1.restrict-def listS-def  $G_p$ .groupS-def noSelect-def
    Restrictive-Comp.restrict-def prodC-def  $G_p$ .groupC-def  $G_p$ .constrain-grp-def
    noConstrain-def Let-def split-def
  by simp
  also have ... =
    TRY do {
      (ck,vk)  $\leftarrow$  key-gen;
      ((c,cvec), $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  ck;
      (p,td)  $\leftarrow$   $E$  (c,cvec);
      (i, p-i, (w, wvec))  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
      let (p-i',w') = Eval ck td p i;
      - :: unit  $\leftarrow$  assert-spmf (length ck = length cvec
         $\wedge$  c = fold ( $\lambda$  i acc. acc  $\otimes$  ck!i [ $\top$ ] (cvec!i)) [0..1);
      return-spmf (verify-eval vk c i (p-i,w)  $\wedge$  p-i  $\neq$  p-i'  $\wedge$  valid-argument i  $\wedge$ 

```

```

valid-eval (p-i,w))
} ELSE return-spmf False
by (rule try-spmf-cong)(simp add: assert-commute)+
also have ... =
TRY do {
  (ck,vk) ← key-gen;
  TRY do {
    ((c,cvec),σ) ← A1 ck;
    TRY do {
      (p,td) ← E (c,cvec);
      TRY do {
        (i, p-i, (w, wvec)) ← AGM2.restrict A2 (ck,σ);
        TRY do {
          let (p-i',w') = Eval ck td p i;
          - :: unit ← assert-spmf (length ck = length cvec
            ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..

```

```

bind-spmf-cong)
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    ((c,cvec),σ) ← A1 ck;
    (p,td) ← E (c,cvec);
    (i, p-i, (w, wvec)) ← AGM2.restrict A2 (ck,σ);
    let (p-i',w') = Eval ck td p i;
    - :: unit ← assert-spmf (length ck = length cvec
      ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1];
    - :: unit ← assert-spmf (verify-eval vk c i (p-i,w) ∧ p-i ≠ p-i' ∧ valid-argument
i ∧ valid-eval (p-i,w));
    return-spmf True
  } ELSE return-spmf False
unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])simp
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    ((c,cvec),σ) ← A1 ck;
    (p,td) ← E (c,cvec);
    (i, p-i, (w, wvec)) ← AGM2.restrict A2 (ck,σ);
    let (p-i',w') = Eval ck td p i;
    - :: unit ← assert-spmf (length ck = length cvec
      ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
      ∧ verify-eval vk c i (p-i,w)
      ∧ p-i ≠ p-i'
      ∧ valid-argument i
      ∧ valid-eval (p-i,w));
    return-spmf True
  } ELSE return-spmf False
by (simp add: assert-collapse)
also have ... = TRY do {
  x :: nat ← sample-uniform (order Gp);
  let (α::'e mod-ring) = of-int-mod-ring (int x);
  let ck = map (λt. gGp ⌈Gp (α ⌈t)) [0..<max-deg+1];
  let vk = map (λt. gGp ⌈Gp (α ⌈t)) [0..<max-deg+1];
  ((c,cvec),σ) ← A1 ck;
  let (p,td) = (Poly (map (of-int-mod-ring::int ⇒ 'e mod-ring) cvec),());
  (i, p-i, (w, wvec)) ← AGM2.restrict A2 (ck,σ);
  let (p-i',w') = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ verify-eval vk c i (p-i,w)
    ∧ p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w));
  return-spmf True
} ELSE return-spmf False

```

```

unfolding key-gen-def Setup-def by auto
also have ... =
  TRY do {
     $x :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
     $\text{let } (\alpha :: 'e \text{ mod-ring}) = \text{of-int-mod-ring } (\text{int } x);$ 
     $\text{let } ck = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1];$ 
     $\text{let } vk = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1];$ 
     $((c, cvec), \sigma) \leftarrow \mathcal{A1} \text{ } ck;$ 
     $\text{let } (p, td) = (\text{Poly } (\text{map } (\text{of-int-mod-ring} :: \text{int} \Rightarrow 'e \text{ mod-ring}) \text{ } cvec), ());$ 
     $(i, p-i, (w, wvec)) \leftarrow \text{AGM2.restrict } \mathcal{A2} \text{ } (ck, \sigma);$ 
     $\text{let } (p-i', w') = \text{Eval } ck \text{ } td \text{ } p \text{ } i;$ 
    - :: unit  $\leftarrow \text{assert-spmf } (\text{length } ck = \text{length } cvec$ 
       $\wedge c = \text{fold } (\lambda i \text{ } acc. \text{ } acc \otimes ck!i \text{ } [\neg] (cvec!i)) [0..<\text{length } ck] \text{ } \mathbf{1}$ 
       $\wedge p-i \neq p-i'$ 
       $\wedge \text{valid-argument } i$ 
       $\wedge \text{valid-eval } (p-i, w)$ 
       $\wedge \text{valid-eval } (p-i', w')$ 
       $\wedge \text{verify-eval } vk \text{ } c \text{ } i \text{ } (p-i, w)$ 
       $\wedge \text{verify-eval } vk \text{ } c \text{ } i \text{ } (p-i', w'));$ 
     $\text{return-spmf True}$ 
  } ELSE  $\text{return-spmf False}$ 
  apply(unfold spmf-rel-eq[symmetric])
  apply (rule rel-spmf-try-spmf)
  apply(unfold Let-def split-def)
  apply(rule rel-spmf-bindI[of (=)] | force)+
  apply(rule assert-cong)
  apply(insert ks-imp-eval-bind-asserts)
  apply(unfold Let-def split-def)
  apply simp+
done
also have ... =
  TRY do {
     $(ck, vk) \leftarrow \text{key-gen};$ 
     $((c, cvec), \sigma) \leftarrow \mathcal{A1} \text{ } ck;$ 
     $(p, td) \leftarrow E \text{ } (c, cvec);$ 
     $(i, p-i, (w, wvec)) \leftarrow \text{AGM2.restrict } \mathcal{A2} \text{ } (ck, \sigma);$ 
     $\text{let } (p-i', w') = \text{Eval } ck \text{ } td \text{ } p \text{ } i;$ 
    - :: unit  $\leftarrow \text{assert-spmf } (\text{length } ck = \text{length } cvec$ 
       $\wedge c = \text{fold } (\lambda i \text{ } acc. \text{ } acc \otimes ck!i \text{ } [\neg] (cvec!i)) [0..<\text{length } ck] \text{ } \mathbf{1}$ 
       $\wedge p-i \neq p-i'$ 
       $\wedge \text{valid-argument } i$ 
       $\wedge \text{valid-eval } (p-i, w)$ 
       $\wedge \text{valid-eval } (p-i', w')$ 
       $\wedge \text{verify-eval } vk \text{ } c \text{ } i \text{ } (p-i, w)$ 
       $\wedge \text{verify-eval } vk \text{ } c \text{ } i \text{ } (p-i', w'));$ 
     $\text{return-spmf True}$ 
  } ELSE  $\text{return-spmf False}$ 
  unfolding key-gen-def Setup-def by force
also have ... =

```

```

TRY do {
  (ck,vk) ← key-gen;
  ((c,cvec),σ) ←  $\mathcal{A}1$  ck;
  (p,td) ← E (c,cvec);
  (i, p-i, (w, wvec)) ←  $AGM2.restrict\ \mathcal{A}2\ (ck,\sigma)$ ;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈] (cvec!i)) [0..length ck] 1);
  let (p-i',w') = Eval ck td p i;
  - :: unit ← assert-spmf (
    p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
    ∧ valid-eval (p-i', w')
    ∧ verify-eval vk c i (p-i, w)
    ∧ verify-eval vk c i (p-i', w'));
  return-spmf True
} ELSE return-spmf False
by (simp add: assert-collapse)
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    ((c,cvec),σ) ←  $\mathcal{A}1$  ck;
    - :: unit ← assert-spmf (length ck = length cvec
      ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈] (cvec!i)) [0..length ck] 1);
    (p,td) ← E (c,cvec);
    (i, p-i, (w, wvec)) ←  $AGM2.restrict\ \mathcal{A}2\ (ck,\sigma)$ ;
    let (p-i',w') = Eval ck td p i;
    - :: unit ← assert-spmf (
      p-i ≠ p-i'
      ∧ valid-argument i
      ∧ valid-eval (p-i,w)
      ∧ valid-eval (p-i', w')
      ∧ verify-eval vk c i (p-i, w)
      ∧ verify-eval vk c i (p-i', w'));
    return-spmf True
  } ELSE return-spmf False
apply (rule try-spmf-cong)
apply simp
apply (subst assert-commute)
apply blast+
done
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    ((c,cvec),σ) ←  $AGM1.restrict\ \mathcal{A}1\ ck$ ;
    (p,td) ← E (c,cvec);
    (i, p-i, (w, wvec)) ←  $AGM2.restrict\ \mathcal{A}2\ (ck,\sigma)$ ;
    let (p-i',w') = Eval ck td p i;
    - :: unit ← assert-spmf (

```

```

       $p-i \neq p-i'$ 
       $\wedge$  valid-argument  $i$ 
       $\wedge$  valid-eval  $(p-i, w)$ 
       $\wedge$  valid-eval  $(p-i', w')$ 
       $\wedge$  verify-eval  $vk\ c\ i\ (p-i, w)$ 
       $\wedge$  verify-eval  $vk\ c\ i\ (p-i', w')$ );
    return-spmf True
  } ELSE return-spmf False
  unfolding AGM1.restrict-def listS-def  $G_p$ .groupS-def noSelect-def
    Restrictive-Comp.restrict-def prodC-def  $G_p$ .groupC-def  $G_p$ .constrain-grp-def
    noConstrain-def Let-def split-def
  by fastforce
also have ... =
  TRY do {
     $(ck, vk) \leftarrow$  key-gen;
     $((c, cvec), \sigma) \leftarrow$  AGM1.restrict  $\mathcal{A}1\ ck$ ;
     $(p, td) \leftarrow$   $E\ (c, cvec)$ ;
     $(i, p-i, (w, wvec)) \leftarrow$  AGM2.restrict  $\mathcal{A}2\ (ck, \sigma)$ ;
    let  $(p-i', w') =$  Eval  $ck\ td\ p\ i$ ;
    - :: unit  $\leftarrow$  assert-spmf (
      valid-eval  $(p-i, w)$ 
       $\wedge$   $p-i \neq p-i'$ 
       $\wedge$  valid-argument  $i$ 
       $\wedge$  valid-eval  $(p-i, w)$ 
       $\wedge$  valid-eval  $(p-i', w')$ 
       $\wedge$  verify-eval  $vk\ c\ i\ (p-i, w)$ 
       $\wedge$  verify-eval  $vk\ c\ i\ (p-i', w')$ );
    return-spmf True
  } ELSE return-spmf False
apply(unfold spmf-rel-eq[symmetric])
apply (rule rel-spmf-try-spmf)
apply(unfold Let-def split-def)
apply(rule rel-spmf-bindI[of (=)] | force)+
apply(rule assert-cong)
apply force+
done
also have ... =
  TRY do {
     $(ck, vk) \leftarrow$  key-gen;
     $((c, cvec), \sigma) \leftarrow$  AGM1.restrict  $\mathcal{A}1\ ck$ ;
     $(p, td) \leftarrow$   $E\ (c, cvec)$ ;
     $(i, p-i, (w, wvec)) \leftarrow$  AGM2.restrict  $\mathcal{A}2\ (ck, \sigma)$ ;
    - :: unit  $\leftarrow$  assert-spmf ( valid-eval  $(p-i, w)$ );
    let  $(p-i', w') =$  Eval  $ck\ td\ p\ i$ ;
    - :: unit  $\leftarrow$  assert-spmf (
       $p-i \neq p-i'$ 
       $\wedge$  valid-argument  $i$ 
       $\wedge$  valid-eval  $(p-i, w)$ 
       $\wedge$  valid-eval  $(p-i', w')$ );

```

```

- :: unit ← assert-spmf(
  verify-eval vk c i (p-i, w)
  ∧ verify-eval vk c i (p-i', w'));
return-spmf True
} ELSE return-spmf False
by (simp add: assert-collapse)
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    (c, i, v, w, v', w') ← knowledge-soundness-reduction-ext E A1 A2 ck;
    - :: unit ← assert-spmf (
      v ≠ v'
      ∧ valid-argument i
      ∧ valid-eval (v,w)
      ∧ valid-eval (v', w'));
    - :: unit ← assert-spmf(
      verify-eval vk c i (v, w)
      ∧ verify-eval vk c i (v', w'));
    return-spmf True
  } ELSE return-spmf False
unfolding knowledge-soundness-reduction-ext-def by force
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    TRY do {
      (c, i, v, w, v', w') ← knowledge-soundness-reduction-ext E A1 A2 ck;
      TRY do {
        - :: unit ← assert-spmf (
          v ≠ v'
          ∧ valid-argument i
          ∧ valid-eval (v,w)
          ∧ valid-eval (v', w'));
        TRY do {
          - :: unit ← assert-spmf(
            verify-eval vk c i (v, w)
            ∧ verify-eval vk c i (v', w'));
          return-spmf True
        } ELSE return-spmf False
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False
unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
also have ... =
  TRY do {
    (ck,vk) ← key-gen;
    TRY do {
      (c, i, v, w, v', w') ← knowledge-soundness-reduction-ext E A1 A2 ck;
      TRY do {

```

```

- :: unit ← assert-spmf (
  v ≠ v'
  ∧ valid-argument i
  ∧ valid-eval (v,w)
  ∧ valid-eval (v', w'));
TRY do {
  return-spmf (verify-eval vk c i (v, w) ∧ verify-eval vk c i (v', w'))
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong
bind-spmf-cong)
also have ... =
TRY do {
  (ck,vk) ← key-gen;
  (c, i, v, w, v', w') ← knowledge-soundness-reduction-ext E A1 A2 ck;
  - :: unit ← assert-spmf (
    v ≠ v'
    ∧ valid-argument i
    ∧ valid-eval (v,w)
    ∧ valid-eval (v', w'));
  return-spmf( verify-eval vk c i (v, w) ∧ verify-eval vk c i (v', w'))
} ELSE return-spmf False
unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])simp
also have ... = eval-bind-game (knowledge-soundness-reduction-ext E A1 A2)
unfolding eval-bind-game-def by presburger
finally show ?thesis .
qed

```

We overestimate the probability of winning the evaluation binding game with the extended adversary by winning it with the normal adversary.

lemma *overestimate-reductions: spmf (eval-bind-game (knowledge-soundness-reduction-ext E A A')) True*

≤ spmf (eval-bind-game (knowledge-soundness-reduction E A A')) True

proof –

note $[simp] = \text{Let-def split-def}$

We extend the evaluation binding game with the extended reduction adversary to a complete game.

have *w-assert-ext: eval-bind-game (knowledge-soundness-reduction-ext E A A')*

=

```

TRY do {
  (ck, vk) ← key-gen;
  ((c,cvec),σ) ← AGM1.restrict A ck;
  (p,td) ← E (c,cvec);
  (i, v, (w, wvec)) ← AGM2.restrict A' (ck,σ);
  - :: unit ← assert-spmf (valid-eval (v, w));

```

```

    let (v',w') = Eval ck td p i;
    - :: unit ← assert-spmf (v ≠ v' ∧ valid-argument i ∧ valid-eval (v, w) ∧
valid-eval (v', w'));
    let b = verify-eval vk c i (v,w);
    let b' = verify-eval vk c i (v',w');
    return-spmf (b ∧ b')} ELSE return-spmf False
unfolding eval-bind-game-def knowledge-soundness-reduction-ext-def
by simp

```

We extend the evaluation binding game with the normal reduction adversary to a complete game.

```

have wo-assert-ext: eval-bind-game (knowledge-soundness-reduction E A A') =
TRY do {
  (ck, vk) ← key-gen;
  ((c,cvec),σ) ← AGM1.restrict A ck;
  (p,td) ← E (c,cvec);
  (i, v, (w, wvec)) ← AGM2.restrict A' (ck,σ);
  let (v',w') = Eval ck td p i;
  - :: unit ← assert-spmf (v ≠ v' ∧ valid-argument i ∧ valid-eval (v, w) ∧
valid-eval (v', w'));
  let b = verify-eval vk c i (v,w);
  let b' = verify-eval vk c i (v',w');
  return-spmf (b ∧ b')} ELSE return-spmf False
unfolding eval-bind-game-def knowledge-soundness-reduction-def
by simp

```

We show the thesis in ennreal, which implies the plain thesis

```

have ennreal (spm (eval-bind-game (knowledge-soundness-reduction-ext E A
A')) True)
  ≤ ennreal (spm (eval-bind-game (knowledge-soundness-reduction E A A'))
True)
unfolding w-assert-ext wo-assert-ext
apply (simp add: spmf-try-spmf ennreal-spmf-bind)
apply (rule nn-integral-mono)+
apply (simp add: assert-spmf-def)
apply (simp add: measure-spmf.emmeasure-eq-measure)
done
then show ?thesis by simp
qed

```

Finally we put everything together: we conclude that for every efficient adversary in the AGM the advantage over winning the knowledge soundness game is less than or equal to breaking the t-SDH assumption.

theorem knowledge-soundness:

```

  spmf (knowledge-soundness-game-AGM A1 A2 E) True
  ≤ t-SDH-Gp.advantage (eval-bind-reduction (knowledge-soundness-reduction E
A1 A2))
using evaluation-binding[of knowledge-soundness-reduction E A1 A2]

```

```

    overestimate-reductions[of A1 A2]
unfolding eval-bind-advantage-def knowledge-soundness-game-alt-def
by linarith

end

end
theory KZG-hiding

imports KZG-correct DL-assumption tDL-assumption CryptHOL-ext
begin

locale KZG-PCS-weak-hiding = KZG-PCS-correct
begin

```

16 Hiding of the KZG

16.1 Definitions for the weak hiding game

The difference between the weak hiding game and the hiding game is that in the weak hiding game the polynomial is uniform random, not arbitrary. Since arbitrary values can be modelled as uninitialized parameters in Isabelle, like the polynomial in the hiding game, we can make the hiding weaker by instantiating a part of it.

The polynomial is chosen uniform random. We also enforce two additional properties on the adversary outputs. With this construction we can use the predefined eval hiding game from Polynomial Commitment Scheme.

definition *weak-eval-hiding-adversary1* :: (*'a vk, 'e argument, 'state*) *eval-hiding-adversary1* $\Rightarrow$ (*'a vk, 'e argument, 'state*) *eval-hiding-adversary1*
where
weak-eval-hiding-adversary1 A vk = do {
 (*I, σ*) $\leftarrow$ *A vk*;
 - :: *unit* $\leftarrow$ *assert-spmf (length I = max-deg $\wedge$ distinct I)*;
 return-spmf (I, σ)
}

eval_hiding_game except the polynomial is chosen at uniform random and not by the adversary.

definition *weak-eval-hiding-game* :: (*'a vk, 'e argument, 'state*) *eval-hiding-adversary1* $\Rightarrow$
(*'e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state*) *eval-hiding-adversary2* $\Rightarrow$ *bool spmf*
where *weak-eval-hiding-game A1 A2 =*
TRY do {
 p :: 'e mod-ring poly $\leftarrow$ *sample-uniform-poly max-deg*;
 eval-hiding-game p (weak-eval-hiding-adversary1 A1) A2
}

} *ELSE* return-spmf *False*

The advantage of the adversary over the weak hiding game is the advantage of eval hiding with the weak adversary.

definition *weak-eval-hiding-advantage* :: ('a vk, 'e argument, 'state) *eval-hiding-adversary1* $\Rightarrow$
 ('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ *real*
where *weak-eval-hiding-advantage* *A1 A2*
 $\equiv$ *spm*f (*weak-eval-hiding-game* *A1 A2*) *True*

16.1.1 DL game

We instantiate the DL game for the group G_p

sublocale *DL-G_p*: *DL* G_p of-int-mod-ring $\circ$ int pow-mod-ring G_p
unfolding *DL-def*
by (*rule* *G_p.cyclic-group-axioms*)

We instantiate the t-DL game for the group G_p

sublocale *t-DL-G_p*: *t-DL* G_p max-deg of-int-mod-ring $\circ$ int pow-mod-ring G_p
unfolding *t-DL-def*
by (*rule* *G_p.cyclic-group-axioms*)

16.1.2 Reduction

interpolate a polynomial over group points i.e. points of the form (i, g^i) .
 Furthermore, return the polynomial evaluated at a value α

definition *interpolate-on* :: ('e mod-ring $\times$ 'a) list $\Rightarrow$ 'e mod-ring $\Rightarrow$ 'a **where**
interpolate-on *xs-ys* α = *do* {
let *xs* = *map fst xs-ys*;
let *lagrange-exp* = *map* ($\lambda(xj, yj). yj \hat{=}_{G_p} (\text{poly } (\text{lagrange-basis-poly } xs \ xj) \ \alpha)$)
xs-ys;
fold ($\lambda x \text{ prod. prod } \otimes_{G_p} x$) *lagrange-exp* **1**}

filter for the elements that are in *xs* but not in *ys*

fun *filter-distinct* :: 'e mod-ring list $\Rightarrow$ 'e argument list $\Rightarrow$ 'e argument list
where *filter-distinct* *xs ys* = *filter* ($\lambda x. \text{find } ((=) \ x) \ ys = \text{None}$) *xs*

lemma *set-filter-distinct*: *set* (*filter-distinct* *xs ys*) = {*x*. *x* $\in$ *set xs* $\wedge$ *x* $\notin$ *set ys*}
by (*simp add: find-None-iff*)

lemma *length-filter-distinct*:

assumes *card* (*set xs*) > *card* (*set ys*)
shows *length* (*filter-distinct* *xs ys*) > 0

proof –

have {*x*. *x* $\in$ *set xs* $\wedge$ *x* $\notin$ *set ys*} = *set xs* – *set ys*
by *blast*

```

moreover have card (set xs - set ys) ≥ card (set xs) - card (set ys)
using diff-card-le-card-Diff by blast
moreover have ... > 0
using assms calculation(2) by linarith
ultimately show ?thesis
using set-filter-distinct
by (metis bot-nat-0.extremum-uniqueI card-length not-gr-zero)
qed

```

function to pick a field element that is not in I. Intuitively, this function returns the lowest value not contained in I (note, finite fields do not have a ordering by default).

```

fun PickDistinct :: 'e argument list ⇒ 'e argument
where PickDistinct I = hd (filter-distinct (map (of-int-mod-ring::int ⇒ 'e mod-ring)
[0..

```

```

lemma card-map-card: card (set (map (of-int-mod-ring::int ⇒ 'e mod-ring) [0..proof -
have card ((of-int-mod-ring ∘ int::nat ⇒ 'e mod-ring) ‘ {0..apply (rule card-image)
apply (rule comp-inj-on)
apply simp
apply (metis of-int-mod-inj-on-ff atLeast0LessThan inj-on-imageI)
done
then show ?thesis by force
qed

```

helping lemma to show that PickDistinct I prepended to I is distinct if I is distinct and smaller than the finite field of its elements.

```

lemma PickDistinct:
assumes length I < CARD('e)
and distinct I
shows distinct ((PickDistinct I)#I)
proof -
have length (filter-distinct (map of-int-mod-ring [0..apply (rule length-filter-distinct)
apply (insert card-map-card)
apply (simp add: distinct-card assms)
done
then have filter-distinct (map of-int-mod-ring [0..by blast
then have PickDistinct I ∉ set I
using set-filter-distinct[of - I]
by (metis (no-types, lifting) PickDistinct.elims list.set-sel(1) mem-Collect-eq)
then show ?thesis
by (meson assms(2) distinct.simps(2))
qed

```

The reduction function takes an adversary for the hiding game and returns an adversary for the DL game. Specifically, the reduction uses the hiding adversary to construct a winning strategy for the DL game (i.e. to win it every time). Essentially, it encodes the DL-instance in a polynomial evaluation on group elements and asks the hiding adversary to return the plain polynomial, which contains the solution to the DL-instance.

```

fun reduction
  :: ('a vk, 'e argument, 'state) eval-hiding-adversary1  $\Rightarrow$ 
    ('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
    eval-hiding-adversary2  $\Rightarrow$  ('a, 'e) DL.adversary
where
  reduction  $\mathcal{A}1$   $\mathcal{A}2$  g-pow-a = do {
    ( $\alpha$ , PK)  $\leftarrow$  Setup;
    ( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$  PK;
    let  $i$  = PickDistinct  $I$ ;
    plain-evals  $\leftarrow$  map-spmf (map ((of-int-mod-ring::int  $\Rightarrow$  'e mod-ring)  $\circ$  int))
    (sample-uniform-list max-deg (order  $G_p$ ));
    let evals = g-pow-a # map ( $\lambda i. \mathbf{g} \hat{\phantom{g}}_{G_p} i$ ) plain-evals ;
    let  $C$  = interpolate-on (zip ( $i \# I$ ) evals)  $\alpha$ ;
    let  $W$  = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{\phantom{g}}_{G_p} y) \hat{\phantom{g}}_{G_p} ((1::'e \text{ mod-ring})/(\alpha-x))))$ 
    (zip  $I$  plain-evals);
     $\varphi' \leftarrow \mathcal{A}2$  PK  $\sigma$   $C$   $I$   $W$ ;
    return-spmf (poly  $\varphi' i$ )
  }

fun reduction-tDL :: ('a vk, 'e argument, 'state) eval-hiding-adversary1
 $\Rightarrow$  ('a, 'e) t-DL.adversary
where reduction-tDL  $\mathcal{A}1$  PK = do {
  ( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$  PK;
  let ( $\alpha'::'e \text{ mod-ring}$ ) = (case (find ( $\lambda x. \mathbf{g} \hat{\phantom{g}}_{G_p} x = PK!1$ ) (PickDistinct  $I \# I$ ))
of Some  $x \Rightarrow x$  | None  $\Rightarrow 1$ );
  return-spmf  $\alpha'$ 
}

```

16.2 Hiding proof

16.2.1 Helping lemmas

lemma PK1[simp]: map ($\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{} t)$) [$0..<\text{max-deg}+1$] $!1 = \mathbf{g}_{G_p} \hat{}_{G_p} \alpha$
by (metis (no-types) Suc-eq-plus1 add.commute d-pos diff-zero landau-product-preprocess(53))
 less-Suc-eq-0-disj nth-map-upt power-one-right)

The "find" in the tDL reduction is successful.

lemma in-set-impl-find: $x \in \text{set } xs \implies \text{Some}(x) = \text{find } (\lambda y. \mathbf{g} \hat{}_{G_p} y = \mathbf{g} \hat{}_{G_p} x)$
 xs

proof –

assume asm: $x \in \text{set } xs$

```

then have  $\exists y. y \in \text{set } xs \wedge y=x$ 
by blast
then have  $\text{find } (\lambda y. y = x) \text{ } xs = \text{Some } (x)$ 
by (smt (verit, ccfv-threshold) find-None-iff find-Some-iff2 not-None-eq)
then show  $\text{Some}(x) = \text{find } (\lambda y. \mathbf{g} \hat{G_p} y = \mathbf{g} \hat{G_p} x) \text{ } xs$ 
by (simp add: pow-on-eq-card)
qed

lemma exchange-lem: length I = max-deg  $\wedge$  distinct I  $\wedge$   $\alpha \in \text{set } (\text{PickDistinct } I \# I)$ 
 $\longleftrightarrow \text{length } I = \text{max-deg} \wedge \text{distinct } I \wedge \alpha \in \text{set } (\text{PickDistinct } I \# I)$ 
 $\wedge \alpha = (\text{case } (\text{find } (\lambda x. \mathbf{g} \hat{G_p} x = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1]!1)$ 
 $(\text{PickDistinct } I \# I)) \text{ of } \text{Some } x \Rightarrow x \mid \text{None} \Rightarrow 1)$ 
proof
assume asm: length I = max-deg  $\wedge$  distinct I  $\wedge$   $\alpha \in \text{set } (\text{PickDistinct } I \# I)$ 
then have some: Some  $\alpha =$ 
 $\text{find } (\lambda x. \mathbf{g} \hat{x} = \text{map } (\lambda t. \mathbf{g} \hat{\alpha \hat{t}}) [0..<\text{max-deg} + 1] ! 1) (\text{PickDistinct } I$ 
 $\# I)$ 
apply (simp only: PK1)
apply (rule in-set-impl-find)
apply simp
done
show  $\text{length } I = \text{max-deg} \wedge$ 
 $\text{distinct } I \wedge$ 
 $\alpha \in \text{set } (\text{PickDistinct } I \# I) \wedge$ 
 $\alpha = (\text{case } (\text{find } (\lambda x. \mathbf{g} \hat{G_p} x = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1]!1)$ 
 $(\text{PickDistinct } I \# I)) \text{ of } \text{Some } x \Rightarrow x \mid \text{None} \Rightarrow 1)$ 
apply (subst some[symmetric])
apply (insert asm)
apply fastforce
done
qed simp

```

Note, the following 3 lemmas are build up for the 4th, which states that `interpolate_on` is equivalent to computing `Commit`.

restate the interpolation of a polynomial as the sum of Lagrange basis polynomials.

```

lemma eval-on-lagrange-basis: poly (lagrange-interpolation-poly xs-ys) x  $\equiv$  (let
 $xs = \text{map } \text{fst } xs\text{-}ys$ 
 $\text{in } \text{sum-list } (\text{map } (\lambda (xj,yj). yj * (\text{poly } (\text{lagrange-basis-poly } xs \text{ } xj) \text{ } x)) \text{ } xs\text{-}ys))$ 
 $(\text{is } ?lhs \equiv ?rhs)$ 
proof –
have ?lhs  $\equiv$  let
 $xs = \text{map } \text{fst } xs\text{-}ys$ 
 $\text{in } \text{sum-list } (\text{map } (\lambda p. \text{poly } p \text{ } x) (\text{map } (\lambda (xj,yj). \text{Polynomial.smult } yj (\text{lagrange-basis-poly}$ 
 $\text{ } xs \text{ } xj)) \text{ } xs\text{-}ys))$ 
unfolding lagrange-interpolation-poly-def Let-def
by (simp add: poly-sum-list poly-hom.hom-sum-list)

```

```

also have ... $\equiv$ let
  xs = map fst xs-ys
  in sum-list (map (( $\lambda$  (xj,yj). poly (Polynomial.smult yj (lagrange-basis-poly xs
xj)) x)) xs-ys)
unfolding split-def Let-def
by (smt (verit, ccfv-SIG) length-map nth-equalityI nth-map)
also have ... $\equiv$ let
  xs = map fst xs-ys
  in sum-list (map (( $\lambda$  (xj,yj). yj * (poly (lagrange-basis-poly xs xj) x))) xs-ys)
by fastforce
finally show ?lhs  $\equiv$  ?rhs .
qed

```

This lemma is used to restate the folding operation used in the Commit function as a summing operation. Together with the prior lemma (i.e. interpolation is summation of Lagrange basis polynomials), this allows to restate Commit as the Lagrange interpolation over some points, evaluated at α .

```

lemma fold-on- $G_p$ -is-sum-list: fold ( $\lambda$ x prod. prod  $\otimes_{G_p}$  x) (map ( $\lambda$ x.  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} f$ 
x) xs) ( $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} z$ )
=  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} z \otimes_{G_p} \mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (\text{sum-list (map f xs)})$ 
proof (induction xs arbitrary: z)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have fold ( $\lambda$ x prod. prod  $\otimes$  x) (map ( $\lambda$ x.  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} f$  x) (a # xs)) ( $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} z$ )
    = fold ( $\lambda$ x prod. prod  $\otimes$  x) (map ( $\lambda$ x.  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} f$  x) xs) ( $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (z + f a)$ )
    by (simp add: mod-ring-pow-mult- $G_p$ )
  also have ... =  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (z + f a) \otimes \mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} \text{sum-list (map f xs)}$ 
    using Cons.IH by blast
  also have ... =  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} z \otimes \mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (f a) \otimes \mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} \text{sum-list (map f xs)}$ 
    by (simp add: mod-ring-pow-mult- $G_p$ )
  also have ... =  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} z \otimes \mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} \text{sum-list (map f (a # xs))}$ 
    using  $G_p$ .cyclic-group-assoc mod-ring-pow-mult- $G_p$  by force
  finally show ?case .
qed

```

The two prior lemmas are pulled together to show that `interpolation_on` is the evaluation of a polynomial on α as a group value i.e. $g^{\phi(\alpha)}$. Note $g^{\phi(\alpha)}$ is also the result of a correctly computed Commit.

```

lemma interpolate-on- $\alpha$ -is- $\varphi$ -of- $\alpha$ :
assumes dist: distinct (map fst xs-ys)
  and length-xs-ys: length xs-ys  $\leq$  max-deg+1
shows interpolate-on (map ( $\lambda$ (x,y).(x, $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} y$ )) xs-ys)  $\alpha$ 
=  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (\text{poly (lagrange-interpolation-poly xs-ys) } \alpha)$ 
proof -
  have interpolate-on (map ( $\lambda$ (x,y).(x, $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} y$ )) xs-ys)  $\alpha$  =
    (let xs = map fst (map ( $\lambda$ (x, y). (x,  $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} y$ )) xs-ys);

```

```

    lagrange-exp =
      map (λ(xj, y). (g  $\hat{\phantom{g}}$ Gp y)  $\hat{\phantom{g}}$ Gp poly (lagrange-basis-poly xs xj) α) xs-ys
    in fold (λx prod. prod  $\otimes$  x) lagrange-exp 1)
    by (smt (verit) case-prod-unfold interpolate-on-def length-map nth-equalityI
nth-map prod.simps(2))
  also have ... = (let xs = map fst (map (λ(x, y). (x, g  $\hat{\phantom{g}}$ Gp y)) xs-ys);
    lagrange-exp =
      map (λ(xj, y). g  $\hat{\phantom{g}}$ Gp (y * poly (lagrange-basis-poly xs xj) α)) xs-ys
    in fold (λx prod. prod  $\otimes$  x) lagrange-exp 1)
  using mod-ring-pow-pow-Gp by presburger
  also have ... = g  $\hat{\phantom{g}}$ Gp
    (∑ (xj,
      y) ← xs-ys. y * poly (lagrange-basis-poly
        (map fst (map (λ(x, y). (x, g  $\hat{\phantom{g}}$ Gp y)) xs-ys)) xj)
        α)
    using fold-on-Gp-is-sum-list[of (λ(xj, y). (y *
      poly
        (lagrange-basis-poly (map fst (map (λ(x, y). (x, g  $\hat{\phantom{g}}$ Gp y)) xs-ys))
        xj)
        α)) xs-ys 0]
    unfolding Let-def split-def by force
  also have ... = g  $\hat{\phantom{g}}$ Gp (poly (lagrange-interpolation-poly xs-ys) α)
    using eval-on-lagrange-basis
    by (smt (verit, ccfv-SIG) fst-conv length-map nth-equalityI nth-map split-def)
  finally show ?thesis by fast
qed

```

corollary *interpolate-on-α-is-φ-of-α-helper:*

```

  assumes length I = max-deg  $\wedge$  distinct I
  and α  $\notin$  set I
shows interpolate-on (zip (PickDistinct I # I) (map (( $\hat{\phantom{g}}$ Gp g) evals)) α)
  = g  $\hat{\phantom{g}}$ Gp (poly (lagrange-interpolation-poly (zip (PickDistinct I # I) evals)) α)
proof -
  obtain xs-ys where xs-ys: xs-ys = zip (PickDistinct I # I) evals by fast
  then have length xs-ys  $\leq$  max-deg + 1 using assms(1) by simp
  moreover have distinct (map fst xs-ys) unfolding xs-ys using assms (1)
    by (metis CARD-q PickDistinct d-l-p distinct-take map-fst-zip-take of-nat-less-imp-less)
  ultimately have interpolate-on (map (λ(x,y).(x,g  $\hat{\phantom{g}}$ Gp y)) xs-ys) α
  = g  $\hat{\phantom{g}}$ Gp (poly (lagrange-interpolation-poly xs-ys) α) using interpolate-on-α-is-φ-of-α
    by blast
  then show ?thesis unfolding xs-ys
    by (simp add: zip-map2)
qed

```

Finally, conclude the statement that `interpolate_on` is equivalent to computing `Commit`. This is what the prior 3 lemmas were building up to.

lemma *interpolate-on-is-Commit:*

```

  assumes dist: distinct (map fst xs-ys)

```

and *length-xs-ys*: $\text{length } xs\text{-}ys \leq \text{max-deg} + 1$
shows *interpolate-on* ($\text{map } (\lambda(x,y).(x, \mathbf{g}_{G_p} \hat{}_{G_p} y)) \text{ } xs\text{-}ys$) $\alpha = g\text{-pow-PK-Prod}$
 $(\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{} t)) [0..<\text{max-deg} + 1]) (\text{lagrange-interpolation-poly } xs\text{-}ys)$
proof –
have *interpolate-on* ($\text{map } (\lambda(x,y).(x, \mathbf{g}_{G_p} \hat{}_{G_p} y)) \text{ } xs\text{-}ys$) α
 $= \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly } (\text{lagrange-interpolation-poly } xs\text{-}ys) \alpha)$
by (*rule interpolate-on- α -is- φ -of- α* [*OF assms*])
also have $\dots = g\text{-pow-PK-Prod}$
 $(\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} t) [0..<\text{max-deg} + 1]) (\text{lagrange-interpolation-poly } xs\text{-}ys)$
proof –
have *degree* ($\text{lagrange-interpolation-poly } xs\text{-}ys$) $\leq \text{max-deg}$
by (*meson assms*(2) *degree-lagrange-interpolation-poly le-diff-conv le-trans nat-le-iff-add*)
then show $\mathbf{g}_{G_p} \hat{}_{G_p} \text{poly } (\text{lagrange-interpolation-poly } xs\text{-}ys) \alpha = g\text{-pow-PK-Prod}$
 $(\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} t) [0..<\text{max-deg} + 1]) (\text{lagrange-interpolation-poly } xs\text{-}ys)$
using *g-pow-PK-Prod-correct* **by** *presburger*
qed
finally show *?thesis* **by** *fast*
qed

lemma *split-pow-div- G_p* :

shows $\mathbf{g}_{G_p} \hat{}_{G_p} (y/x) = (\mathbf{g}_{G_p} \hat{}_{G_p} y) \hat{}_{G_p} (1/x)$
by (*metis mod-ring-pow-pow- G_p mult-cancel-left2 times-divide-eq-right*)

Show that the witness emulation from the reduction adversary, is equivalent to computing Eval if $\alpha \notin I$.

lemma *witness-calc-correct*:

assumes *dist*: *distinct* ($\text{map fst } xs\text{-}ys$)
and *length-xs-ys*: $\text{length } xs\text{-}ys \leq \text{max-deg} + 1$
and *α -nin-xs*: $\alpha \notin \text{set } (\text{map fst } (tl \text{ } xs\text{-}ys))$
shows $\text{map } (\lambda i. (\text{Eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{} t)) [0..<\text{max-deg} + 1]) () (\text{lagrange-interpolation-poly } xs\text{-}ys) i)) (\text{map fst } (tl \text{ } xs\text{-}ys))$
 $= \text{map } (\lambda(x,y). (y, (\mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly } (\text{lagrange-interpolation-poly } xs\text{-}ys) \alpha) \div_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} y) \hat{}_{G_p} (1/(\alpha - x)))) (tl \text{ } xs\text{-}ys)$
proof –
have *?thesis* $= (\text{map } (\lambda(x,y). (\text{Eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{} t)) [0..<\text{max-deg} + 1]) () (\text{lagrange-interpolation-poly } xs\text{-}ys) x)) (tl \text{ } xs\text{-}ys))$
 $= \text{map } (\lambda(x,y). (y, (\mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly } (\text{lagrange-interpolation-poly } xs\text{-}ys) \alpha) \div_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} y) \hat{}_{G_p} (1/(\alpha - x)))) (tl \text{ } xs\text{-}ys)$
by *auto*
also have $\dots$
proof –
have $\bigwedge x y. (x,y) \in \text{set } (tl \text{ } xs\text{-}ys) \implies$
 $\text{Eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} \alpha \hat{} t) [0..<\text{max-deg} + 1]) () (\text{lagrange-interpolation-poly } xs\text{-}ys) x$
 $= (y, (\mathbf{g}_{G_p} \hat{}_{G_p} \text{poly } (\text{lagrange-interpolation-poly } xs\text{-}ys) \alpha \otimes \text{inv } (\mathbf{g}_{G_p} \hat{}_{G_p} y)) \hat{}_{G_p} (1 / (\alpha - x)))$

```

proof –
  fix  $x\ y$ 
  assume  $asm: (x, y) \in \text{set } (tl\ xs\ ys)$ 
  let  $? \varphi = \text{lagrange-interpolation-poly } xs\ ys$ 

  have  $y: \text{poly } (\text{lagrange-interpolation-poly } xs\ ys) \ x = y$ 
    by (rule lagrange-interpolation-poly[OF assms(1)])(simp add: asm calculation in-set-tlD)+

  have  $\text{deg-}\psi\text{-le: degree } (\psi\text{-of } (? \varphi) \ x) \leq \text{max-deg}$ 
    by (meson degree-lagrange-interpolation-poly degree-q-le- $\varphi$  le-diff-conv le-trans length-xs-ys)

  have  $\alpha\text{-neg-}x: \alpha \neq x$ 
    by (metis (mono-tags, lifting) asm assms(1,3) distinct-tl entries-keysD entries-of-alist keys-of-alist map-tl prod.sel(1))

  then have  $\psi\text{-unfold: poly } (\psi\text{-of } ? \varphi \ x) \ \alpha = (\text{poly } ? \varphi \ \alpha - \text{poly } ? \varphi \ x) / (\alpha - x)$ 
    by (simp add:  $\varphi x\text{-m-}\varphi u\text{-eq-}x\mu u\text{-}\psi x$ )

  show  $\text{Eval } (\text{map } (\lambda t. \mathbf{g} \wedge \alpha \wedge t) [0..\text{max-deg} + 1]) \ () \ (\text{lagrange-interpolation-poly } xs\ ys) \ x$ 
    =  $(y, (\mathbf{g} \wedge \text{poly } (\text{lagrange-interpolation-poly } xs\ ys) \ \alpha \otimes \text{inv } (\mathbf{g} \wedge y)) \wedge (1 / (\alpha - x)))$ 
    unfolding Eval-def Let-def
    apply (rule prod-eqI)
    apply (metis asm dist in-set-tlD lagrange-interpolation-poly prod.sel(1))
    apply (simp only: prod.sel(2) g-pow-PK-Prod-correct[OF deg- $\psi$ -le])
    apply (simp only:  $\psi$ -unfold)
    apply (simp only:  $y$ )
    apply (simp only: split-pow-div- $G_p$ )
    apply (simp only: mod-ring-pow-mult-inv- $G_p$ )
    done
  qed
  then show  $?thesis$  by auto
  qed
  finally show  $?thesis$  by fast
  qed

corollary witness-calc-helper:
  assumes  $\text{length } I = \text{max-deg} \wedge \text{distinct } I$ 
  and  $\alpha \notin \text{set } I$ 
  and  $\text{evals}$ 
     $\in \text{set-spmf}$ 
     $(\text{map-spmf } (\text{map } (\lambda x. \text{of-int-mod-ring } (\text{int } x)))$ 
       $(\text{sample-uniform-list } (\text{max-deg} + 1) (\text{order } G_p)))$ 
  shows  $\text{map } (\lambda i. (\text{Eval } (\text{map } (\lambda t. \mathbf{g} \wedge_{G_p} (\alpha \wedge t)) [0..\text{max-deg} + 1]) \ () \ (\text{lagrange-interpolation-poly } (\text{zip } (\text{PickDistinct } I \ \# \ I) \ \text{evals})) \ i)) \ I$ 

```

```

    = map2 (λx y. (y, (gGp ^Gp (poly (lagrange-interpolation-poly (zip (PickDistinct
I # I) evals)) α) ÷Gp gGp ^Gp y) ^Gp (1/(α-x)))) I (tl evals)
proof –
  obtain xs-ys where xs-ys: xs-ys = (zip (PickDistinct I # I) evals) by fast
  have hlength: length evals = length (PickDistinct I # I)
    using assms(3,1) by force
  then have h1: map fst (tl (zip (PickDistinct I # I) evals)) = I
    by (metis list.sel(3) map-fst-zip map-tl)
  have h2: (tl (zip (PickDistinct I # I) evals)) = zip I (tl evals)
    by (metis hlength h1 map-snd-zip map-tl xs-ys zip-map-fst-snd)
  have asm1: distinct (map fst xs-ys) unfolding xs-ys using assms(1)
    by (metis CARD-q PickDistinct d-l-p distinct-take map-fst-zip-take of-nat-less-imp-less)
  moreover have asm2: length xs-ys ≤ max-deg + 1 unfolding xs-ys using
assms(1) by force
  moreover have asm3: α ∉ set (map fst (tl xs-ys))
proof (rule ccontr)
  have tl-move: tl xs-ys = zip I (tl evals) unfolding xs-ys
    by (metis list.sel(2,3) neq-Nil-conv zip.simps(1) zip-Cons-Cons)
  assume ¬ α ∉ set (map fst (tl xs-ys))
  then have α ∈ set (map fst (tl xs-ys)) by linarith
  then obtain y where y: (α, y) ∈ set (tl xs-ys) by fastforce
  then have α ∈ set I by (simp only: tl-move set-zip-leftD)
  then show False using assms(2) by blast
qed

  ultimately have map (λi. (Eval (map (λt. gGp ^Gp (α ^ t)) [0..max-deg + 1])
  () (lagrange-interpolation-poly xs-ys i)) (map fst (tl xs-ys))
    = map (λ(x,y). (y, (gGp ^Gp (poly (lagrange-interpolation-poly xs-ys) α) ÷Gp
gGp ^Gp y) ^Gp (1/(α-x)))) (tl xs-ys)
    using witness-calc-correct by blast
  then show ?thesis unfolding xs-ys
    using h1 h2 by argo
qed

```

show that converting a nat to a finite field element is injective on the natural numbers that are less than the cardinality of the finite field.

lemma *of-int-mod-inj-on-ff*: *inj-on* (*of-int-mod-ring* ∘ *int*:: *nat* ⇒ 'e *mod-ring*) {0..*CARD* ('e)}

proof

```

  fix x
  fix y
  assume x: x ∈ {0..CARD ('e)}
  assume y: y ∈ {0..CARD ('e)}
  assume app-x-eq-y: (of-int-mod-ring ∘ int:: nat ⇒ 'e mod-ring) x = (of-int-mod-ring
  ∘ int:: nat ⇒ 'e mod-ring) y
  show x = y
    using x y app-x-eq-y
    by (metis atLeastLessThan-iff nat-int o-apply of-nat-0-le-iff of-nat-less-iff to-int-mod-ring of-int-mod-ring)
qed

```

The proof goal is to show that the probability of winning the hiding game is less than or equal to winning the DL game. We start with the hiding game transform it via game-based methods into the DL game (with some reduction adversary).

We define 4 new games as subgoals in our proof: - game1 - game2 - game2_w_assert - game2_wo_assert

The proof is structured into 5 steps, each step targets one subgoal, except the fifth one, where the target is DL game:

1. Transform the hiding game into game 1. We exchange Commit with interpolate_on and prepare the game to easily replace CreateWitness with the witness emulation of the reduction adversary. We use only game hops based on bridging steps in this step.
2. We use a game-hop based on a failure event to transform game1 into game2. We extract the negligible probability that α is contained in I, as in that case, CreateWitness is not the same as the emulation in the reduction adversary.
3. We use game-hops based on bridging steps to explicitly restate game2 as game2_w_assert. We extract the assert that the adversary guessed polynomial φ' equals φ , which is part of the hiding game but not of the DL reduction.
4. We use over-estimation to drop the $\varphi' = \varphi$ -assert. We obtain game2_wo_assert from game2_w_assert.
5. We transform game2_wo_assert into the DL game using game hops based on bridging steps.

For a more intuitive understanding of the steps go to the final theorem at the end of the file.

The hiding game with interpolate_on instead of Commit and the random sampling of φ split up into sampling random points for interpolate_on.

definition *game1* :: ('a vk, 'e argument, 'state) eval-hiding-adversary1 $\Rightarrow$ ('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state) eval-hiding-adversary2 $\Rightarrow$ bool spmf **where**
game1 *A1* *A2* = TRY do {
 (α , PK) $\leftarrow$ Setup;
 (*I*, σ) $\leftarrow$ *A1* PK;
 - :: unit $\leftarrow$ assert-spmf (length *I* = max-deg $\wedge$ distinct *I*);
 let *i* = PickDistinct *I*;
 evals $\leftarrow$ map-spmf (map (of-int-mod-ring $\circ$ int::nat $\Rightarrow$ 'e mod-ring)) (sample-uniform-list (max-deg+1) (order *G_p*));
 let φ = lagrange-interpolation-poly (zip (*i*#*I*) evals);
 let exp-evals = map ($\lambda i. \mathbf{g}_{G_p} \widehat{G_p} i$) evals;
 let *C* = interpolate-on (zip (*i*#*I*) exp-evals) α ;
 let *W* = map ($\lambda j. \text{Eval } PK () \varphi j$) *I*;
 $\varphi' \leftarrow$ *A2* PK σ *C* *I* *W*;

return-spmf ($\varphi = \varphi'$) *ELSE* *return-spmf* *False*

game1 with the reduction adversaries emulation instead of CreateWitness

definition *game2* :: ('a vk, 'e argument, 'state) *eval-hiding-adversary1* $\Rightarrow$
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ bool *spmf* **where**
game2 *A1 A2* = *TRY* do {
(α , *PK*) $\leftarrow$ *Setup*;
(*I*, σ) $\leftarrow$ *A1 PK*;
- :: unit $\leftarrow$ *assert-spmf* (length *I* = max-deg $\wedge$ distinct *I*);
let *i* = *PickDistinct I*;
evals $\leftarrow$ *map-spmf* (map (of-int-mod-ring $\circ$ int)) (sample-uniform-list (max-deg+1)
(order *G_p*));
let φ = *lagrange-interpolation-poly* (zip (*i*#*I*) *evals*);
let *exp-evals* = map ($\lambda i. \mathbf{g} \hat{\ } i$) *evals*;
let *C* = *interpolate-on* (zip (*i*#*I*) *exp-evals*) α ;
let *witn-tupel* = map ($\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{\ }_{G_p} y) \hat{\ }_{G_p} (1/(\alpha-x)))$) (zip *I*
(*tl evals*));
 $\varphi' \leftarrow$ *A2 PK* σ *C I witn-tupel*;
return-spmf ($\varphi = \varphi'$) *ELSE* *return-spmf* *False*

game 2 with extracted $\varphi = \varphi'$ -assert

definition *game2-w-assert* :: ('a vk, 'e argument, 'state) *eval-hiding-adversary1* $\Rightarrow$
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ bool *spmf* **where**
game2-w-assert *A1 A2* = *TRY* do {
(α , *PK*) $\leftarrow$ *Setup*;
(*I*, σ) $\leftarrow$ *A1 PK*;
- :: unit $\leftarrow$ *assert-spmf* (length *I* = max-deg $\wedge$ distinct *I*);
let *i* = *PickDistinct I*;
evals $\leftarrow$ *map-spmf* (map (of-int-mod-ring $\circ$ int)) (sample-uniform-list (max-deg+1)
(order *G_p*));
let φ = *lagrange-interpolation-poly* (zip (*i*#*I*) *evals*);
let *C* = *interpolate-on* (zip (*i*#*I*) (map ($\lambda i. \mathbf{g} \hat{\ } i$) *evals*)) α ;
let *witn-tupel* = map ($\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{\ }_{G_p} y) \hat{\ }_{G_p} (1/(\alpha-x)))$) (zip *I*
(*tl evals*));
 $\varphi' \leftarrow$ *A2 PK* σ *C I witn-tupel*;
-::unit $\leftarrow$ *assert-spmf* ($\varphi = \varphi'$);
return-spmf (hd *evals* = poly $\varphi' i$) *ELSE* *return-spmf* *False*

game2_w_assert without the assert.

definition *game2-wo-assert* :: ('a vk, 'e argument, 'state) *eval-hiding-adversary1*
 $\Rightarrow$
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ bool *spmf* **where**
game2-wo-assert *A1 A2* = *TRY* do {
(α , *PK*) $\leftarrow$ *Setup*;
(*I*, σ) $\leftarrow$ *A1 PK*;

```

let i = PickDistinct I;
evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list (max-deg+1)
(order Gp));
let φ = lagrange-interpolation-poly (zip (i#I) evals);
(α, PK) ← Setup;
let C = interpolate-on (zip (i#I) (map (λi. g ^ i) evals)) α;
let withn-tupel = map (λ(x,y). (y, (C ÷Gp gGp ^Gp y) ^Gp (1/(α-x)))) (zip I
(tl evals));
φ' ← A2 PK σ C I withn-tupel;
return-spmf (hd evals = poly φ' i) } ELSE return-spmf False

```

lemma *PickDistinct-Prop*: length a = max-deg ∧ distinct a
 $\implies$ distinct (PickDistinct a#a) ∧ length (PickDistinct a#a) = max-deg + 1
by (metis CARD-q PickDistinct Suc-eq-plus1 d-l-p length-Cons
of-nat-less-iff)

corollary *PickDistinct-Prop'*: length a = max-deg ∧ distinct a ∧ valid-poly φ
 $\implies$ distinct (PickDistinct a#a) ∧ length (PickDistinct a#a) = max-deg + 1
using *PickDistinct-Prop* **by** presburger

lemma *distinct-map-fst-zip*: distinct a $\implies$ distinct (map fst (zip a b))
by (simp add: map-fst-zip-take)

lemma *card-eq-ord*: CARD('e) = order G_p
using CARD-q CARD-G_p **by** force

Step 1 of the proof.

lemma *hiding-game-to-game1*:
shows spmf (weak-eval-hiding-game A1 A2) True = spmf (game1 A1 A2) True
(is ?lhs = ?rhs)
proof –

We unfold the weak eval hiding game

```

have ?lhs = spmf (TRY do {
  p::'e mod-ring poly ← sample-uniform-poly max-deg;
  (ck, vk) ← key-gen;
  (I,σ) ← do {
    (I,σ) ← A1 vk;
    - :: unit ← assert-spmf (length I = max-deg ∧ distinct I);
    return-spmf (I,σ)
  };
  (c,d) ← commit ck p;
  let W = map (λi. Eval ck d p i) I;
  p' ← A2 vk σ c I W;
  return-spmf (p = p') } ELSE return-spmf False) True
unfolding weak-eval-hiding-game-def eval-hiding-game-def weak-eval-hiding-adversary1-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])(simp)
also have ... = spmf (TRY do {
  p::'e mod-ring poly ← sample-uniform-poly max-deg;

```

```

(ck, vk) ← key-gen;
(I,σ) ← A1 vk;
- :: unit ← assert-spmf (length I = max-deg ∧ distinct I);
(c,d) ← commit ck p;
let W = map (λi. Eval ck d p i) I;
p' ← A2 vk σ c I W;
return-spmf (p = p')} ELSE return-spmf False) True
  by (simp add: key-gen-def bind-map-spmf o-def Setup-def split-def Let-def
      del: PickDistinct.simps)

```

sample phi uniform random is sampling evaluation points for the adversaries
 I uniform random

```

also have ... = spmf (TRY do {
  (ck, vk) ← key-gen;
  (I,σ) ← A1 vk;
  p::'e mod-ring poly ← do {
    - :: unit ← assert-spmf (length I = max-deg ∧ distinct I);
    evals::'e mod-ring list ← map-spmf (map (of-int-mod-ring ∘ int:: nat ⇒ 'e
mod-ring)) (sample-uniform-list (max-deg+1) (CARD ('e)));
    let i = PickDistinct I;
    return-spmf (lagrange-interpolation-poly (zip (i#I) evals))
  };
  (c,d) ← commit ck p;
  let W = map (λi. Eval ck d p i) I;
  p' ← A2 vk σ c I W;
  return-spmf (p = p')} ELSE return-spmf False) True
  apply (rule spmf-eqI')
  apply (rule unpack-try-spmf)
  apply (subst bind-commute-spmf; rule unpack-bind-spmf'; rule ext;
    simp
    split: prod.splits
    add: split-paired-all
    del: return-bind-spmf bind-spmf-assoc PickDistinct.simps)+
  apply (subst bind-commute-spmf)
  apply (subst bind-spmf-assoc)
  apply (rule assert-based-eq)
  apply (drule PickDistinct-Prop)
  subgoal for ck vk I σ
    apply (subst sample-uniform-evals-is-sample-poly[of (PickDistinct I # I)
max-deg])
    apply simp+
  done
done

```

Now we bring the game in a nicer form and expose the Setup from key_gen

```

also have ... = spmf(TRY do {
  α::'e mod-ring ← map-spmf (λx. of-int-mod-ring (int x)) (sample-uniform (order
Gp));
  let PK = map (λt. gGp ^Gp (α ^t)) [0..<max-deg+1];

```

```

(I,σ) ← A1 PK;
- :: unit ← assert-spmf (length I = max-deg ∧ distinct I);
  evals::'e mod-ring list ← map-spmf (map (of-int-mod-ring ∘ int:: nat ⇒ 'e
mod-ring)) (sample-uniform-list (max-deg+1) (CARD ('e)));
  let i = PickDistinct I;
  let p = (lagrange-interpolation-poly (zip (i#I) evals));
  (c,d) ← commit PK p;
  let W = map (λi. Eval PK d p i) I;
  p' ← A2 PK σ c I W;
  return-spmf (p = p')} ELSE return-spmf False) True
  by (simp add: key-gen-def bind-map-spmf o-def Setup-def split-def Let-def
    del: PickDistinct.simps)

```

Now we replace the Commit computed with a valid public key PK by interpolate_on

```

also have ... = spmf(TRY do {
  α::'e mod-ring ← map-spmf (λx. of-int-mod-ring (int x)) (sample-uniform (order
Gp));
  let PK = map (λt. gGp ^Gp (α ^t)) [0..<max-deg+1];
  (I,σ) ← A1 PK;
  - :: unit ← assert-spmf (length I = max-deg ∧ distinct I);
    evals::'e mod-ring list ← map-spmf (map (of-int-mod-ring ∘ int:: nat ⇒ 'e
mod-ring)) (sample-uniform-list (max-deg+1) (CARD ('e)));
    let i = PickDistinct I;
    let p = (lagrange-interpolation-poly (zip (i#I) evals));
    let exp-evals = map (λi. gGp ^Gp i) evals;
    let C = interpolate-on (zip (i#I) exp-evals) α;
    let W = map (λi. Eval PK () p i) I;
    p' ← A2 PK σ C I W;
    return-spmf (p = p')} ELSE return-spmf False) True
  apply (rule spmf-eqI')
  apply (simp only: Let-def split-def)
  apply (rule unpack-try-spmf; rule unpack-bind-spmf')+
  apply (rule ext; simp split: prod.splits add: split-paired-all
    del: return-bind-spmf bind-spmf-assoc PickDistinct.simps One-nat-def;
    rule unpack-bind-spmf'; rule ext)
  apply (rule assert-based-eq)
  apply (simp split: prod.splits add: split-paired-all
    del: return-bind-spmf bind-spmf-assoc PickDistinct.simps One-nat-def)
  apply (drule PickDistinct-Prop)
  apply (simp only: commit-def return-bind-spmf bind-return-spmf)
  apply (subst interpolate-on-is-Commit[symmetric, of ])
  apply (simp add: map-fst-zip-take)
  apply simp
  apply (simp add: zip-map2)
  done
also have ... = ?rhs
  by (simp add: game1-def Setup-def Let-def split-def card-eq-ord bind-map-spmf
o-def

```

$\text{del: } \text{PickDistinct.simps}$
finally show $?thesis$.
qed

lemma *lossless-mod-ring-list*: $\text{lossless-spmf } (\text{map-spmf } (\text{map } (\lambda x. \text{of-int-mod-ring } (\text{int } x)::'e \text{ mod-ring}))$
 $(\text{sample-uniform-list } (\text{max-deg} + 1) (\text{order } G_p)))$
by $(\text{simp add: } G_p.\text{order-gt-0})$

Step 2 of the proof:

lemma *fundamental-lemma-game1-game2*:
assumes $\text{lossless-}\mathcal{A}1$: $\bigwedge PK . \text{lossless-spmf } (\mathcal{A}1 \text{ } PK)$
and $\text{lossless-}\mathcal{A}2$: $\bigwedge PK \sigma \ C \ I \ W . \text{lossless-spmf } (\mathcal{A}2 \text{ } PK \ \sigma \ C \ I \ W)$
shows $\text{spm}f (\text{game1 } \mathcal{A}1 \ \mathcal{A}2) \text{ True} \leq \text{spm}f (\text{game2 } \mathcal{A}1 \ \mathcal{A}2) \text{ True} + t\text{-DL-}G_p.\text{advantage}$
 $(\text{reduction-tDL } \mathcal{A}1)$
proof –
note $[\text{simp}] = \text{map-lift-spmf } \text{gpv.map-id } \text{lossless-weight-spmfD } \text{map-spmf-bind-spmf}$
 $\text{bind-spmf-const } \text{Let-def } \text{o-def}$

Step 3 of the proof:

lemma *game2-le-DL*: $\text{spm}f (\text{game2 } \mathcal{A}1 \ \mathcal{A}2) \text{ True} \leq \text{DL-}G_p.\text{advantage } (\text{reduction}$
 $\mathcal{A}1 \ \mathcal{A}2)$
(is $?lhs \leq ?rhs)$
proof –
have $?lhs = \text{spm}f (\text{TRY do } \{$
 $(\alpha, PK) \leftarrow \text{Setup};$
 $\text{TRY do}\{$
 $(I, \sigma) \leftarrow \mathcal{A}1 \text{ } PK;$
 $\text{TRY do}\{$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (\text{length } I = \text{max-deg} \wedge \text{distinct } I);$
 $\text{let } i = \text{PickDistinct } I;$
 $\text{TRY do}\{$
 $\text{evals} \leftarrow \text{map-spmf } (\text{map } (\text{of-int-mod-ring} \circ \text{int})) (\text{sample-uniform-list } (\text{max-deg}+1)$
 $(\text{order } G_p));$
 $\text{let } \varphi = \text{lagrange-interpolation-poly } (\text{zip } (i\#I) \text{ evals});$
 $\text{let } \text{exp-evals} = \text{map } (\lambda i. \mathbf{g} \wedge i) \text{ evals};$
 $\text{let } C = \text{interpolate-on } (\text{zip } (i\#I) \text{ exp-evals}) \ \alpha;$
 $\text{let } \text{witn-tupel} = \text{map } (\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \wedge_{G_p} y) \wedge_{G_p} (1/(\alpha-x)))) (\text{zip } I$
 $(\text{tl evals}));$
 $\text{TRY do}\{$
 $\varphi' \leftarrow \mathcal{A}2 \text{ } PK \ \sigma \ C \ I \ \text{witn-tupel};$
 $\text{TRY do}\{$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (\varphi = \varphi');$
 return-spmf True
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$
 $\} \text{ ELSE return-spmf False}$

```

} ELSE return-spmf False
} ELSE return-spmf False) True
unfolding game2-def split-def Let-def split-def
apply (simp only: return-spmf-assert[symmetric])
apply (fold try-bind-spmf-lossless2[OF lossless-return-spmf])
apply (simp del: PickDistinct.simps)
done
also have ... = spmf (TRY do {
  ( $\alpha$ , PK)  $\leftarrow$  Setup;
  ( $I, \sigma$ )  $\leftarrow$  A1 PK;
  - :: unit  $\leftarrow$  assert-spmf (length I = max-deg  $\wedge$  distinct I);
  let i = PickDistinct I;
  evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
  (order  $G_p$ ));
  let  $\varphi$  = lagrange-interpolation-poly (zip (i#I) evals);
  let exp-evals = map ( $\lambda i. \mathbf{g} \hat{\ } i$ ) evals;
  let C = interpolate-on (zip (i#I) exp-evals)  $\alpha$ ;
  let witn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{\ }_{G_p} y) \hat{\ }_{G_p} (1/(\alpha-x))))$  (zip I
  (tl evals));
   $\varphi' \leftarrow$  A2 PK  $\sigma$  C I witn-tupel;
  - :: unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi'$ );
  return-spmf True
} ELSE return-spmf False) True
unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])
  (simp del: PickDistinct.simps One-nat-def)

```

next we show that the assert $\varphi = \varphi'$ implies $hd\ evals = poly\ \varphi' (PickDistinct\ I)$, which is asserted by the DI reduction game. Hence we add $hd\ evals = poly\ \varphi' (PickDistinct\ I)$ to the assert without changing the game.

```

also have ... = spmf (TRY do {
  ( $\alpha$ , PK)  $\leftarrow$  Setup;
  ( $I, \sigma$ )  $\leftarrow$  A1 PK;
  - :: unit  $\leftarrow$  assert-spmf (length I = max-deg  $\wedge$  distinct I);
  let i = PickDistinct I;
  evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
  (order  $G_p$ ));
  let  $\varphi$  = lagrange-interpolation-poly (zip (i#I) evals);
  let exp-evals = map ( $\lambda i. \mathbf{g} \hat{\ } i$ ) evals;
  let C = interpolate-on (zip (i#I) exp-evals)  $\alpha$ ;
  let witn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{\ }_{G_p} y) \hat{\ }_{G_p} (1/(\alpha-x))))$  (zip I
  (tl evals));
   $\varphi' \leftarrow$  A2 PK  $\sigma$  C I witn-tupel;
  - :: unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi' \wedge hd\ evals = poly\ \varphi' i$ );
  return-spmf True
} ELSE return-spmf False) True
proof -
have equi:  $\bigwedge I\ evals\ \varphi'.\ length\ I = max-deg \wedge distinct\ I$ 
   $\implies evals \in set-spmf\ (map-spmf\ (map\ (of-int-mod-ring \circ int)))$ 

```

```

      (sample-uniform-list (max-deg+1) (order  $G_p$ )))
     $\implies$  ((lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =  $\varphi'$ 
     $\longleftrightarrow$  (lagrange-interpolation-poly (zip (PickDistinct I#I) evals)) =  $\varphi'$ 
       $\wedge$  hd evals = poly  $\varphi'$  (PickDistinct I)))
  proof -
    fix I evals :: 'e mod-ring list
    fix  $\varphi'$ 
    assume asm1: length I = max-deg  $\wedge$  distinct I
    assume asm2: evals  $\in$  set-spmf (map-spmf (map (of-int-mod-ring  $\circ$  int))
    (sample-uniform-list (max-deg+1) (order  $G_p$ )))
    then have evals-length: length evals = max-deg+1
      by (force simp add: bind-map-spmf o-def)
    show (lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =  $\varphi'$ 
     $\longleftrightarrow$  lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =  $\varphi'$   $\wedge$  hd evals
    = poly  $\varphi'$  (PickDistinct I))
  proof
    show lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =  $\varphi'$   $\implies$ 
    lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =  $\varphi'$   $\wedge$  hd evals = poly
     $\varphi'$  (PickDistinct I)
  proof
    assume asm: lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =
     $\varphi'$ 
    show hd evals = poly  $\varphi'$  (PickDistinct I)
  proof(rule lagrange-interpolation-poly[symmetric, of zip (PickDistinct I#I)
    evals])
    show distinct (map fst (zip (PickDistinct I#I) evals))
      using PickDistinct-Prop asm1 distinct-map-fst-zip by blast
    show  $\varphi'$  = lagrange-interpolation-poly (zip (PickDistinct I#I) evals)
      using asm[symmetric] .
    show (PickDistinct I, hd evals)  $\in$  set (zip (PickDistinct I#I) evals)
      using asm1 evals-length
    by (metis Nil-eq-zip-iff add-is-0 hd-in-set hd-zip length-0-conv list.discI
    list.sel(1) rel-simps(92))
  qed
  qed
  qed simp
  qed
  show ?thesis
    unfolding split-def Let-def
    apply (rule spmf-eqI')
    apply (rule unpack-try-spmf)
    apply (rule unpack-bind-spmf'; rule ext)
    apply (rule unpack-bind-spmf'; rule ext)
    apply (rule assert-based-eq)
    apply (rule bind-spmf-cong)
    apply simp
    apply (rule unpack-bind-spmf'; rule ext)
    apply (rule unpack-bind-spmf)
    apply (subst equi)

```

```

    apply simp-all
  done
qed

```

In the next part we split the conjuncture $\varphi = \varphi' \wedge \text{hd evals} = \text{poly } \varphi'$ (*PickDistinct* I) into two separate assert statements, so we can use overestimation at a later step to get closer to the DL game.

```

also have ... = spmf (TRY do {
  ( $\alpha$ ,  $PK$ )  $\leftarrow$  Setup;
  ( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$   $PK$ ;
  - :: unit  $\leftarrow$  assert-spmf (length  $I$  = max-deg  $\wedge$  distinct  $I$ );
  let  $i$  = PickDistinct  $I$ ;
  evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
(order  $G_p$ ));
  let  $\varphi$  = lagrange-interpolation-poly (zip (i# $I$ ) evals);
  let exp-evals = map ( $\lambda i. \mathbf{g} \hat{=} i$ ) evals;
  let  $C$  = interpolate-on (zip (i# $I$ ) exp-evals)  $\alpha$ ;
  let witn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{=}_{G_p} y) \hat{=}_{G_p} (1/(\alpha-x))))$  (zip  $I$ 
(tl evals));
   $\varphi' \leftarrow \mathcal{A}2$   $PK$   $\sigma$   $C$   $I$  witn-tupel;
  - :: unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi'$ );
  - :: unit  $\leftarrow$  assert-spmf (hd evals = poly  $\varphi' i$ );
  return-spmf True
} ELSE return-spmf False) True
by (simp only: assert-collapse)
also have ... = spmf (TRY do {
  ( $\alpha$ ,  $PK$ )  $\leftarrow$  Setup;
  TRY do {
    ( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$   $PK$ ;
    TRY do {
      - :: unit  $\leftarrow$  assert-spmf (length  $I$  = max-deg  $\wedge$  distinct  $I$ );
      let  $i$  = PickDistinct  $I$ ;
      TRY do {
        evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
(order  $G_p$ ));
        let  $\varphi$  = lagrange-interpolation-poly (zip (i# $I$ ) evals);
        let exp-evals = map ( $\lambda i. \mathbf{g} \hat{=} i$ ) evals;
        let  $C$  = interpolate-on (zip (i# $I$ ) exp-evals)  $\alpha$ ;
        let witn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{=}_{G_p} y) \hat{=}_{G_p} (1/(\alpha-x))))$  (zip  $I$ 
(tl evals));
        TRY do {
           $\varphi' \leftarrow \mathcal{A}2$   $PK$   $\sigma$   $C$   $I$  witn-tupel;
          TRY do {
            - :: unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi'$ );
            TRY do {
              return-spmf (hd evals = poly  $\varphi' i$ )
            } ELSE return-spmf False
          } ELSE return-spmf False
        } ELSE return-spmf False
      }
    }
  }
} ELSE return-spmf False

```

```

} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False) True
apply (simp only: split-def Let-def)
apply (subst return-spmf-assert)
apply (fold try-bind-spmf-lossless2[OF lossless-return-spmf])
apply (simp del: PickDistinct.simps)
done
also have ... = spmf (TRY do {
  ( $\alpha$ , PK)  $\leftarrow$  Setup;
  ( $I, \sigma$ )  $\leftarrow$  A1 PK;
  - :: unit  $\leftarrow$  assert-spmf (length I = max-deg  $\wedge$  distinct I);
  let i = PickDistinct I;
  evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
  (order  $G_p$ ));
  let  $\varphi$  = lagrange-interpolation-poly (zip (i#I) evals);
  let exp-evals = map ( $\lambda i. \mathbf{g} \wedge i$ ) evals;
  let C = interpolate-on (zip (i#I) exp-evals)  $\alpha$ ;
  let witn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \wedge_{G_p} y) \wedge_{G_p} (1/(\alpha-x)))$ ) (zip I
  (tl evals));
   $\varphi' \leftarrow$  A2 PK  $\sigma$  C I witn-tupel;
  - :: unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi'$ );
  return-spmf (hd evals = poly  $\varphi'$  i)
} ELSE return-spmf False) True
unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])(simp del: PickDis-
tinct.simps)
also have ...  $\leq$  spmf (TRY do {
  ( $\alpha$ , PK)  $\leftarrow$  Setup;
  ( $I, \sigma$ )  $\leftarrow$  A1 PK;
  let i = PickDistinct I;
  evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
  (order  $G_p$ ));
  let  $\varphi$  = lagrange-interpolation-poly (zip (i#I) evals);
  let exp-evals = map ( $\lambda i. \mathbf{g} \wedge i$ ) evals;
  let C = interpolate-on (zip (i#I) exp-evals)  $\alpha$ ;
  let witn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \wedge_{G_p} y) \wedge_{G_p} (1/(\alpha-x)))$ ) (zip I
  (tl evals));
   $\varphi' \leftarrow$  A2 PK  $\sigma$  C I witn-tupel;
  - :: unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi'$ );
  return-spmf (hd evals = poly  $\varphi'$  i)
} ELSE return-spmf False) True
apply (simp only: Let-def split-def)
apply (rule try-spmf-le)
apply (rule bind-spmf-le)+
apply (simp only: del-assert)
done
also have ...  $\leq$  spmf (TRY do {

```

```

( $\alpha$ ,  $PK$ )  $\leftarrow$  Setup;
( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$   $PK$ ;
let  $i = \text{PickDistinct } I$ ;
evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list ( $\text{max-deg}+1$ )
(order  $G_p$ ));
let exp-evals = map ( $\lambda i. \mathbf{g} \wedge i$ ) evals;
let  $C = \text{interpolate-on } (\text{zip } (i \# I) \text{ exp-evals}) \alpha$ ;
let withn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \wedge_{G_p} y) \wedge_{G_p} (1/(\alpha-x)))) (\text{zip } I$ 
(tl evals));
 $\varphi' \leftarrow \mathcal{A}2$   $PK$   $\sigma$   $C$   $I$  withn-tupel;
return-spmf (hd evals = poly  $\varphi' i$ )
} ELSE return-spmf False) True
apply (simp only: Let-def split-def)
apply (rule try-spmf-le)
apply (rule bind-spmf-le) +
apply (simp only: del-assert)
done

```

Next, we split the random sampled list up into a random point concatenated with a one element shorter random list. The random point represents the DL value which creates later the DL instance.

```

also have ... = spmf (TRY do {
( $\alpha$ ,  $PK$ )  $\leftarrow$  Setup;
( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$   $PK$ ;
let  $i = \text{PickDistinct } I$ ;
 $a \leftarrow$  map-spmf (of-int-mod-ring  $\circ$  int) (sample-uniform (order  $G_p$ ));
plain-evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list  $\text{max-deg}$ 
(order  $G_p$ ));
let evals =  $\mathbf{g} \wedge a \# \text{map } (\lambda i. \mathbf{g} \wedge i) \text{ plain-evals}$ ;
let  $C = \text{interpolate-on } (\text{zip } (i \# I) \text{ evals}) \alpha$ ;
let withn-tupel = map ( $\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \wedge_{G_p} y) \wedge_{G_p} (1/(\alpha-x)))) (\text{zip } I$ 
plain-evals);
 $\varphi' \leftarrow \mathcal{A}2$   $PK$   $\sigma$   $C$   $I$  withn-tupel;
return-spmf ( $a = \text{poly } \varphi' i$ )
} ELSE return-spmf False) True
using pretty-Cons-random-list-split p-gr-two CARD- $G_p$ 
by(simp add: bind-map-spmf o-def Let-def del: PickDistinct.simps)

```

Lastly, we extract the functions from the reduction adversary into an *do*-block to mirror the DL game.

```

also have ... = spmf (TRY do {
 $a \leftarrow$  map-spmf (of-int-mod-ring  $\circ$  int) (sample-uniform (order  $G_p$ ));
( $\alpha$ ,  $PK$ )  $\leftarrow$  Setup;
( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$   $PK$ ;
let  $i = \text{PickDistinct } I$ ;
plain-evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list  $\text{max-deg}$ 
(order  $G_p$ ));
let evals =  $\mathbf{g} \wedge a \# \text{map } (\lambda i. \mathbf{g} \wedge i) \text{ plain-evals}$ ;

```

```

let C = interpolate-on (zip (i#I) evals) α;
let witn-tupel = map (λ(x,y). (y,(C ÷Gp gGp ^Gp y) ^Gp (1/(α-x)))) (zip I
plain-evals);
φ' ← A2 PK σ C I witn-tupel;
return-spmf (a = poly φ' i)
} ELSE return-spmf False) True
  apply (simp only: Let-def split-def)
  apply (subst bind-commute-spmf)
  apply (subst bind-commute-spmf)
  apply (blast)
done
also have ... = spmf (TRY do {
a ← map-spmf (of-int-mod-ring ∘ int) (sample-uniform (order Gp));
a' ← do {
(α, PK) ← Setup;
(I,σ) ← A1 PK;
let i = PickDistinct I;
plain-evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list max-deg
(order Gp));
let evals = g ^ a # map (λi. g ^ i) plain-evals;
let C = interpolate-on (zip (i#I) evals) α;
let witn-tupel = map (λ(x,y). (y,(C ÷Gp gGp ^Gp y) ^Gp (1/(α-x)))) (zip I
plain-evals);
φ' ← A2 PK σ C I witn-tupel;
return-spmf (poly φ' i)};
return-spmf (a = a')
} ELSE return-spmf False) True
  by (simp add: Let-def split-def o-def del: PickDistinct.simps)
  also have ... = DL-Gp.advantage (reduction A1 A2)
  unfolding DL-Gp.advantage-def DL-Gp.game-alt-def2 reduction.simps ..
  finally show ?thesis .
qed

```

Finally we assemble all proof steps for the weak hiding theorem

theorem *weak-hiding*:

```

assumes lossless-A1: ∧PK . lossless-spmf (A1 PK)
and lossless-A2: ∧PK σ C I W . lossless-spmf (A2 PK σ C I W)
shows weak-eval-hiding-advantage A1 A2
  ≤ DL-Gp.advantage (reduction A1 A2) + t-DL-Gp.advantage (reduction-tDL
A1)

```

proof –

```

have weak-eval-hiding-advantage A1 A2 = spmf (game1 A1 A2) True
  using hiding-game-to-game1
  unfolding weak-eval-hiding-advantage-def eval-hiding-advantage-def
  weak-eval-hiding-adversary1-def weak-eval-hiding-game-def .
also have ... ≤ spmf (game2 A1 A2) True + t-DL-Gp.advantage (reduction-tDL
A1)
  using assms fundamental-lemma-game1-game2 by blast
also have ... ≤ DL-Gp.advantage (reduction A1 A2) + t-DL-Gp.advantage

```

```

(reduction-tDL A1)
  by (simp add: game2-le-DL)
  finally show ?thesis .
qed

end

end

theory BatchKZG-def

imports KZG-correct

begin

```

17 Batch Opening Definition

```

locale BatchEvalKZG = KZG
begin

```

We define the batched version of the KZG according to the original KZG paper [8].

The batched version allows to verifiably open a commitment to a polynomial for up to `max_deg` points using only one witness. Note, that this batch version is different from the one mentioned in the Sonic [10] and PLONK [7] SNARKs, where multiple commitments can be batch opened for one point. The KZG [8] batched version is an extension of the KZG as defined in chapter 3 for the two functions `CreateWitnessBatch` and `VerifyEvalBatch`, which we define below as `eval_batch` and `verify_eval_batch`.

```

type-synonym 'e' batch-evaluation = 'e' mod-ring poly

```

17.1 Polynomial operations and prerequisites

calculate the remainder polynomial of $\varphi / \prod_{i \in B} (x-i)$ i.e. $r = \varphi \bmod \prod_{i \in B} (x-i)$

```

fun r :: 'e argument set  $\Rightarrow$  'e mod-ring poly  $\Rightarrow$  'e batch-evaluation where
  r B  $\varphi$  = do {
    let prod-B = prod ( $\lambda i. [-i, 1:]$ ) B;
     $\varphi \bmod \text{prod-B}$  }

```

lemma *deg-r*: $\text{degree } (r \ B \ \varphi) \leq \text{degree } \varphi$

```

by (smt (verit) add.right-neutral bot-nat-0.not-eq-extremum degree-0 degree-mod-less'
  div-poly-eq-0-iff less-or-eq-imp-le mod-div-mult-eq mult-eq-0-iff nat-le-linear order-trans-rules(21)
  r.simps)

```

calculate $(\varphi(x) - r(x)) / \prod_{i \in B} (x-i)$

```

fun  $\psi_B$  :: 'e argument set  $\Rightarrow$  'e mod-ring poly  $\Rightarrow$  'e mod-ring poly where

```

$\psi_B B \varphi = do \{$
 $\quad let \text{ prod-}B = prod (\lambda i. [-i, 1:]) \ B;$
 $\quad (\varphi - (r \ B \ \varphi)) \ div \ \text{prod-}B \}$

$\varphi(x) = (\varphi(x) / \prod_{i \in B} (x-i)) * \prod_{i \in B} (x-i) + \varphi \ mod \ \prod_{i \in B} (x-i)$

lemma $\varphi = \psi_B B \varphi * (prod (\lambda i. [-i, 1:]) \ B) + r \ B \ \varphi$
by *simp*

degree of $\prod_{i \in B} (x-i)$ is $|B|$

lemma *deg-Prod*: $degree (\prod_{i \in B} [-i, 1:]) = card (B::'e \ argument \ set)$
proof –
have *finite B* $\implies degree (\prod_{i \in B} [-i, 1:]) = card (B::'e \ argument \ set)$
proof (*induct B rule: finite-induct*)
case empty
then show *?case* **by** *simp*
next
case (insert a S)
have $degree ([-a, 1:] * (\prod_{i \in S} [-i, 1:])) = degree ([-a, 1:]) + degree$
 $(\prod_{i \in S} [-i, 1:])$
by (*rule degree-mult-eq*) *auto*
then show *?case*
by (*metis (no-types, lifting) One-nat-def card.insert degree-pCons-eq-if eq-numeral-extra(2)*
local.insert(1) local.insert(2) local.insert(3) one-pCons plus-1-eq-Suc prod.insert)
qed
then show *?thesis* **by** *fastforce*
qed

lemma *deg-r-B-le*: $degree (r \ B \ \varphi) \leq card \ B$
by (*metis (no-types, lifting) card-0-eq deg-Prod degree-0 degree-mod-less' less-or-eq-imp-le*
not-gr0 prod.empty prod.infinite r.simps verit-eq-simplify(24))

lemma *deg-r-B-less*: $B \neq \{\} \implies degree \ \varphi > card \ B \implies degree (r \ B \ \varphi) < card \ B$
by (*metis card-eq-0-iff card-gt-0-iff deg-Prod degree-0 degree-mod-less' finite r.simps*)

lemma *deg-div*: $degree ((x::'e \ mod\text{-}ring \ poly) \ div \ y) \leq degree \ x$
by (*metis (no-types, lifting) Polynomial.degree-div-less add-diff-cancel-left' bot-nat-0.extremum-strict*
degree-0 degree-mod-less' degree-mult-right-le diff-zero div-poly-eq-0-iff gr0I less-or-eq-imp-le
mod-div-mult-eq)

lemma *deg-ψ_B*: $degree (\psi_B B \ \varphi) \leq degree \ \varphi$
by (*simp add: poly-div-diff-left deg-div*)

$e \in B$ implies $\prod_{i \in B} (x-i)$ is 0 at e

lemma *i-in-B-prod-B-zero*[*simp*]:
assumes $(i::'e \ argument) \in B$
shows $poly (prod (\lambda i. [-i, 1:]) \ B) \ i = 0$
proof –
have *i-is-zero*: $(\lambda x. poly [-x, 1:] \ i) \ i = 0$ **by** *simp*
have $poly (prod (\lambda i. [-i, 1:]) \ B) \ i$

```

      = (prod (λx. poly [:-x,1:] i) B)
    using poly-prod by fast
  also have prod (λx. poly [:-x,1:] i) B = 0
  proof (rule prod-zero)
    show finite B
    by simp
    show ∃ a ∈ B. poly [:-a, 1:] i = 0
    using i-is-zero assms by fast
  qed
  finally show poly (prod (λi. [:-i,1:]) B) i = 0 .
  qed

```

$r(i) = \varphi(i)$ for all $i \in B$

lemma *r-eq-φ-on-B*:

```

  assumes (i::'e argument) ∈ B
  shows poly (r B φ) i = poly φ i
  proof -
    let ?prod-B = prod (λi. [:-i,1:]) B
    have poly φ i = poly (φ div ?prod-B * ?prod-B) i + poly (φ mod ?prod-B) i
      by (metis div-mult-mod-eq poly-hom.hom-add)
    moreover have poly (φ div ?prod-B * ?prod-B) i = 0
      using i-in-B-prod-B-zero[OF assms] by simp
    ultimately have poly φ i = poly (φ mod ?prod-B) i
      by fastforce
    then show poly (r B φ) i = poly φ i
      by simp
  qed

```

lemma $(\prod_{i \in B. [:-i, 1:]}) \text{ dvd } \varphi \implies \varphi \text{ mod } (\prod_{i \in B. [:-i, 1:]}) = 0$
 by fastforce

17.2 Function definitions

We define EvalBatch according to CreateWitnessBatch in [8] section 3.4 Batch Opening. The function reveals all points $(i, \varphi(i))$ for $i \in B$ using only one witness value. It returns $(B, r(x), w_i)$, where B is the provided set of positions, $r(x)$ is a polynomial holding all evaluations (i.e. $r(i) = \varphi(i)$ for all $i \in B$), and w_B is the witness for B and $r(x)$.

definition *eval-batch* :: $'a \text{ ck} \Rightarrow \text{trapdoor} \Rightarrow 'e \text{ mod-ring poly} \Rightarrow 'e \text{ argument set} \Rightarrow ('e \text{ batch-evaluation} \times 'a \text{ witness})$

where

```

  eval-batch PK td φ B = (
    let r = r B φ; — remainder of  $\varphi(x) / (\prod_{i \in B. (x-i)}$  i.e.  $\varphi(x) \text{ mod } (\prod_{i \in B. (x-i)})$ 
    ψ = ψB B φ; —  $(\varphi(x) - r(x)) / (\prod_{i \in B. (x-i)})$ 
    w-B = g-pow-PK-Prod PK ψ —  $g^{\psi}(\alpha)$ 
  in
    (r, w-B) — (B, r(x),  $g^{\psi}(\alpha)$ )

```

)

We define `VerifyEvalBatch` according to [8] section 3.4 Batch Opening. The function verifies the witness `w_B` for a set `B`, a polynomial `r(x)`, and a commitment `C` to a polynomial $\varphi(x)$.

definition *verify-eval-batch* :: 'a vk $\Rightarrow$ 'a commit $\Rightarrow$ 'e argument set $\Rightarrow$ ('e batch-evaluation $\times$ 'a witness) $\Rightarrow$ bool

where

verify-eval-batch PK C B val = (
 let (r-x, w_B) = val;
 g-pow-prod-B = g-pow-PK-Prod PK (prod ($\lambda i. [-i, 1:]$) B);
 g-pow-r = g-pow-PK-Prod PK r-x in
 (e g-pow-prod-B w_B $\otimes_{G_T}$ (e g g-pow-r) = e C g))
 — e($g^{\wedge}(\prod_{i \in B. (\alpha - i)), g^{\wedge} \psi(\alpha)}$) $\otimes$ e($g, g^{\wedge} r(\alpha)$) = e(C, g)

definition *valid-argument-batch* :: 'e argument set $\Rightarrow$ bool

where *valid-argument-batch* B = (card B $\leq$ max-deg)

definition *valid-eval-batch*::('e batch-evaluation $\times$ 'a witness) $\Rightarrow$ bool

where *valid-eval-batch* val = (let (r,w) = val in degree r < max-deg $\wedge$ w $\in$ carrier G_p)

the `BatchEvalKZG` is a polynomial commitment scheme

sublocale *bKZG*: *abstract-polynomial-commitment-scheme key-gen commit verify-poly eval-batch*

verify-eval-batch valid-poly valid-argument-batch valid-eval-batch .

end

end

theory *BatchKZG-correct*

imports *BatchKZG-def*

begin

18 Correctness of the batched KZG

locale *BatchEvalKZG-PCS-correct* = *BatchEvalKZG* + *KZG-PCS-correct*
begin

We show perfect correctness i.e. that the game played by an honest committer and honest verifier has guaranteed success; success probability 1.

We show the pairing check performed by `VerifyEvalVBatch` by values (e.g. with $g^{\phi(\alpha)}$ instead of the commitment C). This enables us to prove that the pairing check holds for e.g. a correctly computed commitment.

lemma *eq-on-e-Batch*: (e ($g^{\wedge}_{G_p}$ poly ($\prod_{i \in B. [-i, 1:]$) α) ($g^{\wedge}_{G_p}$ poly ($\psi_B B$ φ) α))

$\otimes_{G_T} (e \mathbf{g} (\mathbf{g} \hat{}_{G_p} \text{poly } (r \ B \ \varphi) \ \alpha))$
 $= e (\mathbf{g} \hat{}_{G_p} \text{poly } \varphi \ \alpha) \ \mathbf{g}$
proof –
have $(\text{poly } (\prod_{i \in B}. [- \ i, \ 1:]) \ \alpha) * (\text{poly } (\psi_B \ B \ \varphi) \ \alpha) + \text{poly } (r \ B \ \varphi) \ \alpha = \text{poly}$
 $\varphi \ \alpha$
by $(\text{metis } (\text{no-types, lifting}) \ \psi_B.\text{simps} \ \text{add.commute} \ \text{add-diff-cancel-right}' \ \text{div-poly-eq-0-iff}$
 $\text{minus-mod-eq-mult-div} \ \text{mod-div-mult-eq} \ \text{nonzero-mult-div-cancel-left} \ \text{poly-hom.hom-add}$
 $\text{poly-mult} \ r.\text{elims})$
then show *?thesis*
using *e-bilinear e-linear-in-fst e-linear-in-snd G_p .generator-closed addition-in-exponents-on-e*
by *presburger*
qed

theorem *KZG-correct: bKZG.correct-eval*
unfolding *bKZG.correct-eval-def valid-poly-def valid-argument-batch-def*
proof $(\text{intro allI, intro impI})$
fix $\varphi :: 'e \text{ mod-ring poly}$
fix $B :: 'e \text{ mod-ring set}$
assume $\text{deg-}\varphi$: $\text{degree } \varphi \leq \text{max-deg}$
assume cardB : $\text{card } B \leq \text{max-deg}$
show $\text{spmf } (\text{bKZG.correct-eval-game } \varphi \ B) \ \text{True} = 1$
proof –

show that $g^{\psi_B(\alpha)}$ is correctly computed using a correct public key PK

have $g\text{-pow-}\psi_B$: $\forall x. \ g\text{-pow-PK-Prod}$
 $(\text{map } (\lambda t. \ \mathbf{g} \hat{}_{G_p} \text{of-int-mod-ring } (\text{int } x) \ \wedge t) \ [0..<\text{max-deg} + 1])$
 $(\psi_B \ B \ \varphi) = \mathbf{g} \hat{}_{G_p} \text{poly } (\psi_B \ B \ \varphi) (\text{of-int-mod-ring } (\text{int } x))$
using $\text{deg-}\psi_B \ g\text{-pow-PK-Prod-correct} \ \text{le-trans } \text{deg-}\varphi$ **by** *blast*

show that $g^{r(\alpha)}$ is correctly computed using a correct public key PK

have $g\text{-pow-}r_B$: $\forall x. \ g\text{-pow-PK-Prod}$
 $(\text{map } (\lambda t. \ \mathbf{g} \hat{}_{G_p} \text{of-int-mod-ring } (\text{int } x) \ \wedge t) \ [0..<\text{max-deg} + 1])$
 $(r \ B \ \varphi) = \mathbf{g} \hat{}_{G_p} \text{poly } (r \ B \ \varphi) (\text{of-int-mod-ring } (\text{int } x))$
using $\text{deg-}r \ g\text{-pow-PK-Prod-correct} \ \text{le-trans } \text{deg-}\varphi$ **by** *blast*

show that $g^{\prod_{i \in B}(\alpha - i)}$ is correctly computed using a correct public key PK

have $g\text{-ow-Prod}$: $\forall x. \ g\text{-pow-PK-Prod}$
 $(\text{map } (\lambda t. \ \mathbf{g} \hat{}_{G_p} \text{of-int-mod-ring } (\text{int } x) \ \wedge t) \ [0..<\text{max-deg} + 1])$
 $(\prod_{i \in B}. [- \ i, \ 1:]) = \mathbf{g} \hat{}_{G_p} \text{poly } (\prod_{i \in B}. [- \ i, \ 1:]) (\text{of-int-mod-ring}$
 $(\text{int } x))$
using $\text{deg-Prod} \ g\text{-pow-PK-Prod-correct} \ \text{le-trans } \text{cardB} \ \text{less-imp-le-nat}$ **by** *presburger*

unfold Setup to gain access to the definition of the public key PK. This step is necessary to be able to use the conversions showed above

have $\text{spmf } (\text{bKZG.correct-eval-game } \varphi \ B) \ \text{True} =$
 $\text{spmf } (\text{do}\{$
 $x :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } G_p);$

```

let  $\alpha = \text{of-int-mod-ring } (\text{int } x)$ ;
let  $PK = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1]$ ;
( $C, d$ )  $\leftarrow \text{commit } PK \ \varphi$ ;
let  $W = \text{eval-batch } PK \ d \ \varphi \ B$ ;
return-spmf ( $\text{verify-eval-batch } PK \ C \ B \ W$ )
}) True
unfolding  $\text{bKZG.correct-eval-game-def key-gen-def Setup-def}$ 
   $\text{abstract-polynomial-commitment-scheme.correct-eval-game-def Let-def}$ 
by force

```

transform the computation from the functions into values.

```

also have ... = spmf (do{
   $x :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } G_p)$ ;
  return-spmf (
     $e (\mathbf{g} \hat{\mathbf{g}}_{G_p} \text{poly } (\prod_{i \in B. [-i, 1:]}) (\text{of-int-mod-ring } (\text{int } x))) (\mathbf{g} \hat{\mathbf{g}}_{G_p} \text{poly } (\psi_B$ 
 $B \ \varphi) (\text{of-int-mod-ring } (\text{int } x)))$ 
 $\otimes_{G_T} (e \ \mathbf{g} (\mathbf{g} \hat{\mathbf{g}}_{G_p} \text{poly } (r \ B \ \varphi) (\text{of-int-mod-ring } (\text{int } x))))$ 
 $= e (\mathbf{g} \hat{\mathbf{g}}_{G_p} \text{poly } \varphi (\text{of-int-mod-ring } (\text{int } x))) \ \mathbf{g})$ 
  }) True
unfolding  $\text{eval-batch-def verify-eval-batch-def commit-def Let-def split-def}$ 
   $\text{g-pow-PK-Prod-correct}[OF \ \text{deg-}\varphi]$ 
using  $\text{g-pow-}\psi_B \ \text{g-pow-}rB \ \text{g-ow-Prod}$ 
by simp

```

Use the pairing equality by value showed in 'eq_on_e_Batch' to conclude that the game simply returns True i.e. has a success probability of 1.

```

also have ... = spmf (do{
   $x :: \text{nat} \leftarrow \text{sample-uniform } (\text{order } G_p)$ ;
  return-spmf (True
})) True
using  $\text{eq-on-e-Batch deg-Prod by algebra}$ 
also have ... = spmf ( $\text{scale-spmf } (\text{weight-spmf } (\text{sample-uniform } (\text{Coset.order } G_p)))$ 
 $(\text{return-spmf True}))$  True
using  $\text{bind-spmf-const}[of \ \text{sample-uniform } (\text{Coset.order } G_p) \ \text{return-spmf True}]$ 
by presburger
also have ... = 1
  using  $\text{weight-sample-uniform-gt-0 CARD-}G_p \ \text{p-gr-two by simp}$ 
finally show ?thesis .
qed
qed
end
end
theory  $\text{BatchKZG-poly-bind}$ 

imports  $\text{BatchKZG-def KZG-poly-bind}$ 

begin

```

19 Polynomial Binding of the batched KZG

locale *BatchEvalKZG-PCS-poly-bind* = *BatchEvalKZG* + *KZG-CS-binding*
begin

We inherit polynomial binding for the batched KZG directly from the standard KZG.

theorem *bKZG.poly-bind-advantage* $\mathcal{A} \leq t\text{-DL-}G_p.\text{advantage}$ (*bind-reduction* $\mathcal{A}$)
using *polynomial-binding by blast*

end

end

theory *BatchKZG-eval-bind*

imports *BatchKZG-correct tBSDH-assumption CryptHOL-ext*
begin

20 Evaluation Binding of the batched KZG

locale *BatchEvalKZG-PCS-eval-bind* = *BatchEvalKZG-PCS-correct*
begin

We prove two evaluation binding properties: 1. the classical evaluation binding game from PCS 2. the binding game from the original KZG paper [8] The difference is that 1. shows that the batched evaluation function is binding following the classical definition (i.e. two batched evaluations for the same argument must be equal) and 2. shows that the batched evaluation function is binding with respect to single evaluation function.

20.1 t-BSDH game

We instantiate the t-BSDH game for the pairing e , with the group G_p as preimage group and G_T as target group.

sublocale *t-BSDH*: *t-BSDH* G_p G_T *max-deg of-int-mod-ring* $\circ$ *int pow-mod-ring*
 G_p *pow-mod-ring* G_T e
unfolding *t-BSDH-def*
using $G_T.\text{cyclic-group-axioms}$ $G_p.\text{cyclic-group-axioms}$ **by** *presburger*

20.2 Binding for the batched evaluation according to KZG10

type-synonym ($'a'$, $'e'$) *adversary* =
 $'a'$ *ck* $\Rightarrow$
 $('a'$ *commit* $\times$ $'e'$ *argument* $\times$ $'e'$ *evaluation* $\times$ $'a'$ *witness* $\times$ $'e'$ *argument set*
 $\times$ $'a'$ *witness* $\times$ $'e'$ *batch-evaluation*) *spmf*

This is the formalized evaluation binding game

definition *bind-game* :: ('a, 'e) adversary $\Rightarrow$ bool spmf
where *bind-game* $\mathcal{A} = \text{TRY do}$ {
 (*ck, vk*) $\leftarrow$ *key-gen*;
 (*C, i, φ -i, w-i, B, w-B, r-x*) $\leftarrow$ $\mathcal{A}$ *ck*;
 - :: unit $\leftarrow$ *assert-spmf* (*i* $\in$ *B* $\wedge$ φ -i $\neq$ poly *r-x i*
 $\wedge$ *valid-argument-batch* *B* $\wedge$ *valid-eval* (φ -i, w-i) $\wedge$ *valid-eval-batch* (*r-x, w-B*));

 let *b* = *verify-eval vk C i* (φ -i, w-i);
 let *b'* = *verify-eval-batch vk C B* (*r-x, w-B*);
 return-spmf (*b* $\wedge$ *b'*) } *ELSE return-spmf False*

The advantage of the adversary over the evaluation binding game is the probability that it wins.

definition *bind-advantage* :: ('a, 'e) adversary $\Rightarrow$ real
where *bind-advantage* $\mathcal{A} \equiv \text{spmf (bind-game } \mathcal{A}) \text{ True}$

20.2.1 Defining a reduction function to t-BSDH

The reduction function takes an adversary for the evaluation binding game and returns an adversary for the t-BSDH game. Specifically, the reduction uses the evaluation binding adversary to construct a winning strategy for the t-BSDH game (i.e. to win it every time). Essentially, it uses the fact that the values supplied by the adversary already break the t-BSDH assumption.

fun *reduction*
 :: ('a, 'e) adversary $\Rightarrow$ ('a, 'e, 'c) t-BSDH.adversary
where
reduction $\mathcal{A}$ *PK* = do {
 (*C, i, φ -i, w-i, B, w-B, r-x*) $\leftarrow$ $\mathcal{A}$ *PK*;
 let *p'* = *g-pow-PK-Prod PK* (*prod* ($\lambda i. [-i, 1:]$) *B* *div* $[-i, 1:]$);
 let *r'* = *g-pow-PK-Prod PK* ((*r-x* - $[: \text{poly } r-x i:]$) *div* $[-i, 1:]$);
 return-spmf ($-i :: 'e \text{ mod-ring}, (e \text{ } p' \text{ } w-B \otimes_{G_T} e \text{ } (r' \div_{G_p} w-i) \text{ } \mathbf{g}) \wedge_{G_T} (1/(\varphi-i - \text{poly } r-x i))$)) }

The reduction adversary extended for asserts that are present in the evaluation binding game. We use this definition to show equivalence to the evaluation binding game. Later on we can then easily over-estimate the probability from this extended version to the normal reduction.

fun *ext-reduction*
 :: ('a, 'e) adversary $\Rightarrow$ ('a, 'e, 'c) t-BSDH.adversary
where
ext-reduction $\mathcal{A}$ *PK* = do {
 (*C, i, φ -i, w-i, B, w-B, r-x*) $\leftarrow$ $\mathcal{A}$ *PK*;
 - :: unit $\leftarrow$ *assert-spmf* (*i* $\in$ *B* $\wedge$ φ -i $\neq$ poly *r-x i*
 $\wedge$ *valid-argument-batch* *B*
 $\wedge$ *valid-eval* (φ -i, w-i) $\wedge$ *valid-eval-batch* (*r-x, w-B*)
 $\wedge$ *verify-eval PK C i* (φ -i, w-i) $\wedge$ *verify-eval-batch PK C B*
 (*r-x, w-B*));

```

let p' = g-pow-PK-Prod PK (prod (λi. [-i,1:]) B div [-i,1:]);
let r' = g-pow-PK-Prod PK ((r-x - [:poly r-x i:]) div [-i,1:]);
return-spmf (-i::'e mod-ring, (e p' w-B ⊗GT e (r' ÷Gp w-i) g) ^GT (1/(φ-i -
poly r-x i))) }

```

20.2.2 Helping lemmas

An alternative but equivalent game for the evaluation binding game. This alternative game encapsulates the event that the Adversary wins in the `assert_spmf` statement. It's a closely adapted proof from `Sigma_Commit_Crypto.Commitment_Schemes bind_game_alt_def`

lemma *bind-game-alt-def*:

```

bind-game A = TRY do {
  (ck,vk) ← key-gen;
  (C, i, φ-i, w-i, B, w-B, r-x) ← A ck;
  - :: unit ← assert-spmf (i ∈ B ∧ φ-i ≠ poly r-x i
    ∧ valid-argument-batch B ∧ valid-eval (φ-i, w-i) ∧ valid-eval-batch (r-x, w-B));

```

```

  let b = verify-eval vk C i (φ-i, w-i);
  let b' = verify-eval-batch vk C B (r-x, w-B);
  - :: unit ← assert-spmf (b ∧ b');
  return-spmf True} ELSE return-spmf False
(is ?lhs = ?rhs)

```

proof –

```

have ?lhs = TRY do {
  (ck,vk) ← key-gen;
  TRY do {
    (C, i, φ-i, w-i, B, w-B, r-x) ← A ck;
    TRY do {
      - :: unit ← assert-spmf (i ∈ B ∧ φ-i ≠ poly r-x i
        ∧ valid-argument-batch B ∧ valid-eval (φ-i, w-i) ∧ valid-eval-batch (r-x, w-B));

```

```

      TRY do {
        let b = verify-eval vk C i (φ-i, w-i);
        let b' = verify-eval-batch vk C B (r-x, w-B);
        return-spmf (b ∧ b')
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False
} ELSE return-spmf False

```

```

  unfolding split-def bind-game-def
  by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp

```

```

also have ... = TRY do {
  (ck,vk) ← key-gen;
  TRY do {
    (C, i, φ-i, w-i, B, w-B, r-x) ← A ck;
    TRY do {
      - :: unit ← assert-spmf (i ∈ B ∧ φ-i ≠ poly r-x i

```

$\wedge \text{valid-argument-batch } B \wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B));$
TRY *do* {
 let $b = \text{verify-eval } vk \ C \ i \ (\varphi-i, w-i);$
 let $b' = \text{verify-eval-batch } vk \ C \ B \ (r-x, w-B);$
 $--unit \leftarrow \text{assert-spmf } (b \wedge b');$
 return-spmf *True*
} *ELSE* *return-spmf* *False*
} *ELSE* *return-spmf* *False*
} *ELSE* *return-spmf* *False*
} *ELSE* *return-spmf* *False*
by(*auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong*
bind-spmf-cong)
also have $\dots = ?rhs$
 unfolding *split-def Let-def*
 by(*fold try-bind-spmf-lossless2[OF lossless-return-spmf]*) *simp*
finally show *?thesis* .
qed

show that VerifyEvalBatch of B and r(x) and VerifEval on $i \in B$ and $\varphi-i$ such that $\varphi-i \neq r(x)$ implies that the t-BSDH is broken. This lemma captures that the adversaries messages already break the t-BSDH assumption.

lemma *verifys-impl-t-BSDH-break:*

assumes $i \in B \wedge \varphi-i \neq \text{poly } r-x \ i$
 $\wedge \text{valid-argument-batch } B$
 $\wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B)$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}^{\wedge \alpha} t) [0..<max-deg + 1]) \ C \ i \ (\varphi-i, w-i)$
 $\wedge \text{verify-eval-batch } (\text{map } (\lambda t. \mathbf{g}^{\wedge \alpha} t) [0..<max-deg + 1]) \ C \ B \ (r-x, w-B)$
shows $e \ \mathbf{g} \ \mathbf{g}^{\wedge_{G_T}} (1 / (\alpha + - i))$
 $=$
 $(e \ (g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}^{\wedge \alpha} t) [0..<max-deg + 1])$
 $\quad ((\prod_{i \in B. [- i, 1:]}) \text{ div } [- i, 1:])))$
 $\quad w-B$
 $\otimes_{G_T}$
 $e \ (g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}^{\wedge \alpha} t) [0..<max-deg + 1])$
 $\quad ((r-x - [: \text{poly } r-x \ i:]) \text{ div } [- i, 1:]) \otimes$
 $\quad \text{inv } w-i)$
 $\quad \mathbf{g})^{\wedge_{G_T}}$
 $\quad (1 / (\varphi-i - \text{poly } r-x \ i))$
(is ?goal = ?cmpt)

proof –

obtain b **where** $b: w-B = \mathbf{g}^{\wedge} b$
 apply (*insert assms*)
 apply (*simp only: valid-eval-batch-def Let-def split-def*)
 apply (*simp split: prod.split*)
 apply (*metis G_p.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int*)
 done
obtain y **where** $y: w-i = \mathbf{g}^{\wedge} y$
 apply (*insert assms*)

```

apply (simp only: valid-eval-def Let-def split-def)
apply (simp split: prod.split)
by (metis Gp.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int)
obtain  $r'-x$  where  $r'-x: r'-x = (r-x - [:poly\ r-x\ i:]) \mathit{div} [: - i, 1:]$  by blast
then have  $r'-x-r-x: r-x = r'-x * [: - i, 1:] + [:poly\ r-x\ i:]$ 
by (metis Groups.mult-ac(2) diff-eq-eq nonzero-mult-div-cancel-left one-neq-zero
pCons-eq-0-iff synthetic-div-correct')
obtain  $p'-x$  where  $p'-x:p'-x = (\prod_{i \in B}. [: - i, 1:]) \mathit{div} [: - i, 1:]$  by blast
then have  $p'-x-p-x: (\prod_{i \in B}. [: - i, 1:]) = p'-x * [: - i, 1:]$ 
using assms
by (metis (no-types, lifting) Groups.mult-ac(2) arith-extra-simps(6) i-in-B-prod-B-zero
nonzero-mult-div-cancel-left one-neq-zero pCons-eq-0-iff synthetic-div-correct')

```

the proof is essentially rearranging equations, an outline can be found in the batched versions evaluation binding proof section in the thesis paper.

```

from assms have  $e\ w-i\ (\mathit{map}\ (\lambda t. \mathbf{g} \wedge \alpha \wedge t)\ [0..<max-deg + 1])! 1 \otimes \mathit{inv}\ (\mathbf{g} \wedge i)) \otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \wedge_{G_T} \varphi-i$ 
=  $e\ (g\text{-pow-PK-Prod}\ (\mathit{map}\ (\lambda t. \mathbf{g} \wedge \alpha \wedge t)\ [0..<max-deg + 1])\ (\prod_{i \in B}. [: - i, 1:])))\ w-B \otimes_{G_T}$ 
 $e\ \mathbf{g}\ (g\text{-pow-PK-Prod}\ (\mathit{map}\ (\lambda t. \mathbf{g} \wedge \alpha \wedge t)\ [0..<max-deg + 1])\ r-x)\ (\mathbf{is}\ ?lhs = ?rhs)$ 
unfolding verify-eval-def verify-eval-batch-def Let-def split-def by auto
moreover have  $?lhs = e\ (\mathbf{g} \wedge y)\ (\mathbf{g} \wedge (\alpha-i)) \otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \wedge_{G_T} \varphi-i$ 
unfolding  $y$  using PK-i d-pos mod-ring-pow-mult-inv-Gp by auto
moreover have  $?rhs = e\ (\mathbf{g} \wedge poly\ (\prod_{i \in B}. [: - i, 1:])\ \alpha)\ (\mathbf{g} \wedge b) \otimes_{G_T} e\ \mathbf{g}\ (\mathbf{g} \wedge poly\ r-x\ \alpha)$ 
proof -
have  $g\text{-pow-PK-Prod}\ (\mathit{map}\ (\lambda t. \mathbf{g} \wedge \alpha \wedge t)\ [0..<max-deg + 1])\ (\prod_{i \in B}. [: - i, 1:] )$ 
=  $\mathbf{g} \wedge poly\ (\prod_{i \in B}. [: - i, 1:] )\ \alpha$ 
using g-pow-PK-Prod-correct assms deg-Prod le-simps(1)
unfolding valid-eval-batch-def valid-argument-batch-def by presburger
moreover have  $g\text{-pow-PK-Prod}\ (\mathit{map}\ (\lambda t. \mathbf{g} \wedge \alpha \wedge t)\ [0..<max-deg + 1])\ r-x$ 
=  $\mathbf{g} \wedge poly\ r-x\ \alpha$ 
using g-pow-PK-Prod-correct assms unfolding verify-eval-batch-def valid-argument-batch-def valid-eval-batch-def
by (simp split: prod.split add: split-def Let-def del: g-pow-PK-Prod.simps)
ultimately show ?thesis unfolding  $b$  by argo
qed
ultimately have  $e\ (\mathbf{g} \wedge y)\ (\mathbf{g} \wedge (\alpha-i)) \otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \wedge_{G_T} \varphi-i = e\ (\mathbf{g} \wedge poly\ (\prod_{i \in B}. [: - i, 1:] )\ \alpha)\ (\mathbf{g} \wedge b) \otimes_{G_T} e\ \mathbf{g}\ (\mathbf{g} \wedge poly\ r-x\ \alpha)$ 
by argo
then have  $y*(\alpha-i) + \varphi-i = (poly\ (\prod_{i \in B}. [: - i, 1:] )\ \alpha)*b + poly\ r-x\ \alpha$ 
using e-bilinear e-linear-in-snd
using addition-in-exponents-on-e pow-on-eq-card-GT-carrier-ext' by auto

```

mimicking steps from the batch opening binding proof in the original paper; see Appendix C.3 of [8].

```

then have  $(poly\ (\prod_{i \in B}. [: - i, 1:] )\ \alpha)*b - (\alpha-i)*y = \varphi-i - poly\ r-x\ \alpha$ 

```

```

    by (metis (no-types, lifting) add-diff-cancel-left' add-diff-cancel-right' add-diff-eq
mult.commute)
    then have (poly ( $\prod_{i \in B} [-i, 1:]$ )  $\alpha$ )* $b - (\alpha - i)*y = \varphi - i - (poly\ r'-x\ \alpha)*(\alpha - i)$ 
- poly  $[:poly\ r-x\ i:]$   $\alpha$ 
    using  $r'-x-r-x$  poly-mult poly-add
    by (smt (z3) ab-group-add-class.ab-diff-conv-add-uminus add-cancel-left-right
group-cancel.add2 more-arith-simps(6) more-arith-simps(9) mult-cancel-right1 poly-0
poly-pCons uminus-add-conv-diff)
    then have  $(\alpha - i)*(poly\ p'-x\ \alpha)*b - (\alpha - i)*y = \varphi - i - (poly\ r'-x\ \alpha)*(\alpha - i) -$ 
poly  $[:poly\ r-x\ i:]$   $\alpha$ 
    using  $p'-x-p-x$ 
    by (smt (verit) Groups.add-ac(2) add-uminus-conv-diff more-arith-simps(6)
mult.commute one-pCons poly-1 poly-mult poly-pCons)
    then have  $(\alpha - i)*((poly\ p'-x\ \alpha)*b - y + poly\ r'-x\ \alpha) = \varphi - i - poly\ [:poly\ r-x\ i:]$ 
 $\alpha$ 
    by (simp add: Groups.mult-ac(2) Groups.mult-ac(3) Rings.ring-distrib(1)
Rings.ring-distrib(4))
    then have  $(\alpha - i)*((poly\ p'-x\ \alpha)*b - y + poly\ r'-x\ \alpha) = \varphi - i - poly\ r-x\ i$ 
    by auto
    then have poly-eq-res:  $1/(\alpha - i) = ((poly\ p'-x\ \alpha)*b - y + poly\ r'-x\ \alpha)/(\varphi - i -$ 
poly  $r-x\ i)$ 
    by (metis (no-types, lifting) assms div-self divide-divide-eq-left mult.commute
mult-eq-0-iff right-minus-eq)
    moreover have ?cmt =  $(e\ (\mathbf{g} \wedge_{G_p} (poly\ p'-x\ \alpha))\ (\mathbf{g} \wedge_{G_p} b) \otimes_{G_T} e\ (\mathbf{g} \wedge_{G_p}$ 
 $(poly\ r'-x\ \alpha) \div_{G_p} (\mathbf{g} \wedge_{G_p} y))$ 
 $\mathbf{g} \wedge_{G_T} (1 / (\varphi - i - poly\ r-x\ i))$ 
    proof -
    have  $g\text{-pow-PK-Prod}\ (map\ (\lambda t. \mathbf{g} \wedge_{\alpha} t)\ [0..<max-deg + 1])\ ((\prod_{i \in B} [-i,$ 
 $1:])\ div\ [-i, 1:])$ 
    =  $\mathbf{g} \wedge_{G_p} (poly\ p'-x\ \alpha)$ 
    unfolding  $p'-x$ 
    by (rule  $g\text{-pow-PK-Prod-correct}$ )(metis (no-types, lifting) assms verify-eval-batch-def
valid-argument-batch-def deg-Prod deg-div le-trans nat-le-linear not-less)
    moreover have  $g\text{-pow-PK-Prod}\ (map\ (\lambda t. \mathbf{g} \wedge_{\alpha} t)\ [0..<max-deg + 1])\ ((r-x$ 
-  $[:poly\ r-x\ i:])\ div\ [-i, 1:])$ 
    =  $\mathbf{g} \wedge_{G_p} (poly\ r'-x\ \alpha)$ 
    unfolding  $r'-x$ 
    apply (rule  $g\text{-pow-PK-Prod-correct}$ )
    apply (insert assms)
    apply (simp add: verify-eval-batch-def valid-argument-batch-def valid-eval-batch-def

    del: One-nat-def  $g\text{-pow-PK-Prod.simps}$ )
    apply (metis (mono-tags, opaque-lifting) deg-div degree-diff-le degree-pCons-0

    le-trans less-or-eq-imp-le zero-le)
    done
    ultimately show ?thesis unfolding  $y\ b$  by argo
qed
moreover have  $\dots = e\ \mathbf{g}\ \mathbf{g} \wedge_{G_T} (((poly\ p'-x\ \alpha)*b - y + poly\ r'-x\ \alpha)/(\varphi - i -$ 

```

```

poly r-x i))
proof -
  have (e (g  $\hat{G}_p$  poly p'-x  $\alpha$ ) (g  $\hat{G}_p$  b)  $\otimes_{G_T}$ 
    e (g  $\hat{G}_p$  poly r'-x  $\alpha$   $\otimes$  inv (g  $\hat{G}_p$  y)) g)  $\hat{G}_T$  (1 / ( $\varphi$ -i - poly r-x i))
  = (e (g  $\hat{G}_p$  poly p'-x  $\alpha$ ) (g  $\hat{G}_p$  b)  $\otimes_{G_T}$ 
    e (g  $\hat{G}_p$  (poly r'-x  $\alpha$  - y)) g)  $\hat{G}_T$  (1 / ( $\varphi$ -i - poly r-x i))
  using mod-ring-pow-mult-inv- $G_p$  by presburger
  also have ... = (e g g  $\hat{G}_T$  ((poly p'-x  $\alpha$ )*b)  $\otimes_{G_T}$ 
    e g g  $\hat{G}_T$  (poly r'-x  $\alpha$  - y))  $\hat{G}_T$  (1 / ( $\varphi$ -i - poly r-x i))
  using e-linear-in-fst  $G_p$ .generator-closed e-bilinear by presburger
  also have ... = (e g g  $\hat{G}_T$  ((poly p'-x  $\alpha$ )*b + poly r'-x  $\alpha$  - y))  $\hat{G}_T$  (1 /
    ( $\varphi$ -i - poly r-x i))
  using add-diff-eq
  by (metis  $G_p$ .generator-closed addition-in-exponents-on-e)
  also have ... = e g g  $\hat{G}_T$  (((poly p'-x  $\alpha$ )*b + poly r'-x  $\alpha$  - y)/( $\varphi$ -i - poly
    r-x i))
  using  $G_T$ .int-pow-pow e-bilinear by force
  finally show ?thesis
  by (simp add: cross3-simps(14,8))
qed
ultimately
show ?thesis by fastforce
qed

```

20.3 Literal helping lemma

CryptHOL has some difficulties with simplifying, thus we need to use literal helping lemmas, that state the equalities we want to exchange literally.

lemma *literal-helping*:

```

(i  $\in$  (B::'e argument set)  $\wedge$ 
  ( $\varphi$ -i:: 'e evaluation)  $\neq$  (poly r-x i:: 'e evaluation)  $\wedge$ 
  valid-argument-batch B  $\wedge$ 
  valid-eval ( $\varphi$ -i,w-i)  $\wedge$  valid-eval-batch (r-x,w-B)  $\wedge$ 
  verify-eval (map ( $\lambda$ t. g  $\hat{G}_p$   $\alpha$   $\hat{G}_p$  t) [0.. $\text{max-deg} + 1$ ]) C i
  ( $\varphi$ -i,w-i)  $\wedge$ 
    verify-eval-batch (map ( $\lambda$ t. g  $\hat{G}_p$   $\alpha$   $\hat{G}_p$  t) ( $\alpha$ ::'e mod-ring)  $\hat{G}_p$  t)
    [0.. $\text{max-deg} + 1$ ]) C B (r-x,w-B)  $\wedge$ 
    e g g  $\hat{G}_T$  (1 / ( $\alpha$  + - i)) =
    (e (g-pow-PK-Prod (map ( $\lambda$ t. g  $\hat{G}_p$   $\alpha$   $\hat{G}_p$  t) [0.. $\text{max-deg} + 1$ ])
      (( $\prod$  i $\in$ B. [:- i, 1:]) div [:- i, 1:])))
      w-B  $\otimes_{G_T}$ 
    e (g-pow-PK-Prod (map ( $\lambda$ t. g  $\hat{G}_p$   $\alpha$   $\hat{G}_p$  t) [0.. $\text{max-deg} + 1$ ])
      ((r-x - [poly r-x i:]) div [:- i, 1:]  $\otimes$ 
        inv w-i))
      g)  $\hat{G}_T$ 
    ((1::'e mod-ring) / ( $\varphi$ -i - poly r-x i)))
 $\longleftrightarrow$ 
(i  $\in$  B  $\wedge$ 

```

```

 $\varphi\text{-}i \neq \text{poly } r\text{-}x \ i \wedge$ 
 $\text{valid-argument-batch } B \wedge$ 
 $\text{valid-eval } (\varphi\text{-}i, w\text{-}i) \wedge \text{valid-eval-batch } (r\text{-}x, w\text{-}B) \wedge$ 
 $\text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\alpha} t) [0..<\text{max-deg} + 1]) \ C \ i$ 
 $(\varphi\text{-}i, w\text{-}i) \wedge$ 
 $\text{verify-eval-batch } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\alpha} t) [0..<\text{max-deg} + 1]) \ C$ 
 $B \ (r\text{-}x, w\text{-}B))$ 
using verifys-impl-t-BSDH-break by fast

```

20.4 KZG eval bind game to reduction game - equivalence theorem

We show that the binding game is equivalent to the t-BSDH game with the extended reduction adversary.

lemma *bind-game-eq-t-BSDH-red*: $\text{bind-game } \mathcal{A} = \text{t-BSDH.game } (\text{ext-reduction } \mathcal{A})$

proof –

note $[simp] = \text{Let-def split-def}$

abbreviations for the `mod_ring` version of sample uniform nat and the public key

```

let  $?mr = \lambda \alpha. (\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring})$ 
let  $?PK = \lambda \alpha. (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{(\alpha)} t) [0..<\text{max-deg}+1])$ 

```

Firstly, unfold the t-BSDH game and the reduction adversary

```

have  $\text{t-BSDH.game } (\text{ext-reduction } \mathcal{A}) = \text{TRY do } \{$ 
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
 $(c, g) \leftarrow (\text{ext-reduction } \mathcal{A}) \ (?PK \ \alpha);$ 
 $--unit \leftarrow \text{assert-spmf } ((e \ \mathbf{g} \ \mathbf{g}) \hat{G_T} (1/((?mr \ \alpha)+c)) = g);$ 
 $\text{return-spmf True}$ 
 $\}$  ELSE  $\text{return-spmf False}$ 
unfolding t-BSDH.game-alt-def by (metis o-def)
also have  $\dots = \text{TRY do } \{$ 
 $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
 $(C, i, \varphi\text{-}i, w\text{-}i, B, w\text{-}B, r\text{-}x) \leftarrow \mathcal{A} \ (?PK \ \alpha);$ 
 $- :: \text{unit} \leftarrow \text{assert-spmf } (i \in B \wedge \varphi\text{-}i \neq \text{poly } r\text{-}x \ i$ 
 $\wedge \text{valid-argument-batch } B$ 
 $\wedge \text{valid-eval } (\varphi\text{-}i, w\text{-}i) \wedge \text{valid-eval-batch } (r\text{-}x, w\text{-}B)$ 
 $\wedge \text{verify-eval } (?PK \ \alpha) \ C \ i \ (\varphi\text{-}i, w\text{-}i) \wedge \text{verify-eval-batch}$ 
 $(?PK \ \alpha) \ C \ B \ (r\text{-}x, w\text{-}B));$ 
 $\text{let } p' = g\text{-pow-PK-Prod } (?PK \ \alpha) \ (\text{prod } (\lambda i. [-i, 1:]) \ B \ \text{div } [-i, 1:]);$ 
 $\text{let } r' = g\text{-pow-PK-Prod } (?PK \ \alpha) \ ((r\text{-}x - [\text{poly } r\text{-}x \ i:]) \ \text{div } [-i, 1:]);$ 
 $--unit \leftarrow \text{assert-spmf } ((e \ \mathbf{g} \ \mathbf{g}) \hat{G_T} (1/((?mr \ \alpha)+(-i))) = (e \ p' \ w\text{-}B \otimes_{G_T} e$ 
 $(r' \div_{G_p} w\text{-}i) \ \mathbf{g}) \hat{G_T} (1/(\varphi\text{-}i - \text{poly } r\text{-}x \ i)));$ 
 $\text{return-spmf True}$ 
 $\}$  ELSE  $\text{return-spmf False}$ 
by force

```

fold the two asserts together so we can reason about their joined content.

```

also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
  TRY do {
     $(C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A} (?PK \alpha);$ 
    TRY do {
      - :: unit  $\leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i$ 
         $\wedge \text{valid-argument-batch } B$ 
         $\wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B)$ 
         $\wedge \text{verify-eval } (?PK \alpha) \ C \ i \ (\varphi-i, w-i) \wedge \text{verify-eval-batch}$ 
 $(?PK \alpha) \ C \ B \ (r-x, w-B));$ 
      let  $p' = g\text{-pow-PK-Prod } (?PK \alpha) \ (\text{prod } (\lambda i. [-i, 1:]) \ B \ \text{div } [-i, 1:]);$ 
      let  $r' = g\text{-pow-PK-Prod } (?PK \alpha) \ ((r-x - [: \text{poly } r-x \ i:]) \ \text{div } [-i, 1:]);$ 
      -::unit  $\leftarrow \text{assert-spmf } ((e \ \mathbf{g} \ \mathbf{g}) \wedge_{G_T} (1/((?mr \ \alpha)+(-i))) = (e \ p' \ w-B \otimes_{G_T} e$ 
 $(r' \div_{G_p} w-i) \ \mathbf{g}) \wedge_{G_T} (1/(\varphi-i - \text{poly } r-x \ i)));$ 
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
} ELSE return-spmf False
unfolding split-def Let-def
by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
  TRY do {
     $(C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A} (?PK \alpha);$ 
    TRY do {
      let  $p' = g\text{-pow-PK-Prod } (?PK \alpha) \ (\text{prod } (\lambda i. [-i, 1:]) \ B \ \text{div } [-i, 1:]);$ 
      let  $r' = g\text{-pow-PK-Prod } (?PK \alpha) \ ((r-x - [: \text{poly } r-x \ i:]) \ \text{div } [-i, 1:]);$ 
      - :: unit  $\leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i$ 
         $\wedge \text{valid-argument-batch } B$ 
         $\wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B)$ 
         $\wedge \text{verify-eval } (?PK \alpha) \ C \ i \ (\varphi-i, w-i) \wedge \text{verify-eval-batch}$ 
 $(?PK \alpha) \ C \ B \ (r-x, w-B)$ 
         $\wedge (e \ \mathbf{g} \ \mathbf{g}) \wedge_{G_T} (1/((?mr \ \alpha)+(-i)))$ 
         $= (e \ p' \ w-B \otimes_{G_T} e \ (r' \div_{G_p} w-i) \ \mathbf{g}) \wedge_{G_T} (1/(\varphi-i - \text{poly}$ 
 $r-x \ i)));$ 
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
} ELSE return-spmf False
by (simp add: assert-collapse del: g-pow-PK-Prod.simps)
also have ... = TRY do {
   $\alpha \leftarrow \text{sample-uniform } (\text{order } G_p);$ 
   $(C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A} (?PK \alpha);$ 
  let  $p' = g\text{-pow-PK-Prod } (?PK \alpha) \ (\text{prod } (\lambda i. [-i, 1:]) \ B \ \text{div } [-i, 1:]);$ 
  let  $r' = g\text{-pow-PK-Prod } (?PK \alpha) \ ((r-x - [: \text{poly } r-x \ i:]) \ \text{div } [-i, 1:]);$ 
  - :: unit  $\leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i$ 
     $\wedge \text{valid-argument-batch } B$ 
     $\wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B)$ 

```

$$\begin{aligned}
& \wedge \text{verify-eval } (?PK \ \alpha) \ C \ i \ (\varphi-i, w-i) \wedge \text{verify-eval-batch} \\
& (?PK \ \alpha) \ C \ B \ (r-x, w-B) \\
& \wedge (e \ \mathbf{g} \ \mathbf{g}) \wedge_{G_T} (1/((?mr \ \alpha)+(-i))) \\
& = (e \ p' \ w-B \otimes_{G_T} e \ (r' \div_{G_p} w-i) \ \mathbf{g}) \wedge_{G_T} (1/(\varphi-i - \text{poly} \\
& r-x \ i)); \\
& \text{return-spmf True} \\
& \} \text{ ELSE return-spmf False} \\
& \text{unfolding split-def Let-def} \\
& \text{by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp}
\end{aligned}$$

erase the pairing check with the literal helping lemma to get closer to the eval bind game

$$\begin{aligned}
& \text{also have } \dots = \text{TRY do } \{ \\
& \quad \alpha \leftarrow \text{sample-uniform (order } G_p); \\
& \quad (C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A} (?PK \ \alpha); \\
& \quad - :: \text{unit} \leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i \\
& \quad \quad \wedge \text{valid-argument-batch } B \\
& \quad \quad \wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B) \\
& \quad \quad \wedge \text{verify-eval } (?PK \ \alpha) \ C \ i \ (\varphi-i, w-i) \wedge \text{verify-eval-batch} \\
& \quad (?PK \ \alpha) \ C \ B \ (r-x, w-B)); \\
& \quad \text{return-spmf True} \\
& \} \text{ ELSE return-spmf False} \\
& \text{unfolding Let-def using literal-helping by algebra}
\end{aligned}$$

Split the message checks off the Evaluations checks, as it is done in the eval bind game

$$\begin{aligned}
& \text{also have } \dots = \text{TRY do } \{ \\
& \quad \alpha \leftarrow \text{sample-uniform (order } G_p); \\
& \quad \text{TRY do } \{ \\
& \quad (C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A} (?PK \ \alpha); \\
& \quad \text{TRY do } \{ \\
& \quad - :: \text{unit} \leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i \\
& \quad \quad \wedge \text{valid-argument-batch } B \\
& \quad \quad \wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B) \\
& \quad \quad \wedge \text{verify-eval } (?PK \ \alpha) \ C \ i \ (\varphi-i, w-i) \wedge \text{verify-eval-batch} \\
& \quad (?PK \ \alpha) \ C \ B \ (r-x, w-B)); \\
& \quad \text{return-spmf True} \\
& \quad \} \text{ ELSE return-spmf False} \\
& \quad \} \text{ ELSE return-spmf False} \\
& \} \text{ ELSE return-spmf False} \\
& \text{unfolding split-def Let-def} \\
& \text{by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp} \\
& \text{also have } \dots = \text{TRY do } \{ \\
& \quad \alpha \leftarrow \text{sample-uniform (order } G_p); \\
& \quad \text{TRY do } \{ \\
& \quad (C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A} (?PK \ \alpha); \\
& \quad \text{TRY do } \{ \\
& \quad - :: \text{unit} \leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i \\
& \quad \quad \wedge \text{valid-argument-batch } B
\end{aligned}$$

```

       $\wedge$  valid-eval  $(\varphi-i, w-i) \wedge$  valid-eval-batch  $(r-x, w-B)$ );
- :: unit  $\leftarrow$  assert-spmf (verify-eval ( $?PK \alpha$ )  $C i (\varphi-i, w-i) \wedge$  verify-eval-batch
( $?PK \alpha$ )  $C B (r-x, w-B)$ );
  return-spmf True
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
  by (simp add: assert-collapse del: g-pow-PK-Prod.simps)
also have ... = TRY do {
   $\alpha \leftarrow$  sample-uniform (order  $G_p$ );
  ( $C, i, \varphi-i, w-i, B, w-B, r-x$ )  $\leftarrow \mathcal{A} (?PK \alpha)$ ;
  - :: unit  $\leftarrow$  assert-spmf ( $i \in B \wedge \varphi-i \neq \text{poly } r-x i$ 
       $\wedge$  valid-argument-batch  $B$ 
       $\wedge$  valid-eval  $(\varphi-i, w-i) \wedge$  valid-eval-batch  $(r-x, w-B)$ );
- :: unit  $\leftarrow$  assert-spmf (verify-eval ( $?PK \alpha$ )  $C i (\varphi-i, w-i) \wedge$  verify-eval-batch
( $?PK \alpha$ )  $C B (r-x, w-B)$ );
  return-spmf True
} ELSE return-spmf False
unfolding split-def Let-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp

```

the obtained game is already the evaluation binding game.

```

also have ... = bind-game  $\mathcal{A}$ 
using bind-game-alt-def unfolding key-gen-def Setup-def by simp
finally show ?thesis ..
qed

```

From the previous lemma we conclude that the adversary's advantage on winning the evaluation binding game is the same as winning the t-BSDH game with the extended reduction adversary.

lemma *evaluation-binding-ext-red*: $\text{bind-advantage } \mathcal{A} = \text{t-BSDH.advantage } (\text{ext-reduction } \mathcal{A})$

```

unfolding bind-advantage-def t-BSDH.advantage-def
using bind-game-eq-t-BSDH-red by presburger

```

Now we use overestimation to show that the advantage of winning the t-BSDH game with the extended reduction adversary is less than or equal to winning it with the normal reduction adversary.

lemma *overestimate-reductions*: $\text{spm} (t\text{-BSDH.game } (\text{ext-reduction } \mathcal{A})) \text{ True}$
 $\leq \text{spm} (t\text{-BSDH.game } (\text{reduction } \mathcal{A})) \text{ True}$
(is $\text{spm } ?lhgame \text{ True} \leq \text{spm } ?rhgame \text{ True}$)

```

proof -
  note [simp] = Let-def split-def

```

abbreviations for the mod_ring version of sample uniform nat and the public key

```

let ? $\alpha$  =  $\lambda \alpha. (\text{of-int-mod-ring } (\text{int } \alpha)::'e \text{ mod-ring})$ 
let ?PK =  $\lambda \alpha. (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\phantom{t}}_{G_p} ((? \alpha \alpha) \hat{t})) [0..<\text{max-deg}+1])$ 

```

We extend the t-BSDH game with the extended reduction adversary to a complete game.

```

have bind-red-ext: t-BSDH.game (ext-reduction  $\mathcal{A}$ ) = TRY do {
   $\alpha \leftarrow \text{sample-uniform}(\text{order } G_p)$ ;
   $(C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A}(\text{?PK } \alpha)$ ;
  - :: unit  $\leftarrow \text{assert-spmf } (i \in B \wedge \varphi-i \neq \text{poly } r-x \ i$ 
     $\wedge \text{valid-argument-batch } B$ 
     $\wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B)$ 
     $\wedge \text{verify-eval } (\text{?PK } \alpha) \ C \ i \ (\varphi-i, w-i) \wedge \text{verify-eval-batch}$ 
     $(\text{?PK } \alpha) \ C \ B \ (r-x, w-B))$ ;
  let  $p' = g\text{-pow-PK-Prod } (\text{?PK } \alpha) \ (\text{prod } (\lambda i. [-i, 1:]) \ B \ \text{div } [-i, 1:]);$ 
  let  $r' = g\text{-pow-PK-Prod } (\text{?PK } \alpha) \ ((r-x - [\text{poly } r-x \ i:]) \ \text{div } [-i, 1:]);$ 
  -::unit  $\leftarrow \text{assert-spmf } ((e \ \mathbf{g} \ \mathbf{g}) \wedge_{G_T} (1/((\text{?}\alpha \ \alpha) + (-i))) = (e \ p' \ w-B \otimes_{G_T} e$ 
     $(r' \div_{G_p} w-i) \ \mathbf{g}) \wedge_{G_T} (1/(\varphi-i - \text{poly } r-x \ i)))$ ;
  return-spmf True
} ELSE return-spmf False
by (force simp add: t-BSDH.game-alt-def[of (ext-reduction  $\mathcal{A}$ )])

```

We extend the t-BSDH game with reduction adversary to a complete game.

```

have eval-bind-red-ext: t-BSDH.game (reduction  $\mathcal{A}$ ) = TRY do {
   $\alpha \leftarrow \text{sample-uniform}(\text{order } G_p)$ ;
   $(C, i, \varphi-i, w-i, B, w-B, r-x) \leftarrow \mathcal{A}(\text{?PK } \alpha)$ ;
  let  $p' = g\text{-pow-PK-Prod } (\text{?PK } \alpha) \ (\text{prod } (\lambda i. [-i, 1:]) \ B \ \text{div } [-i, 1:]);$ 
  let  $r' = g\text{-pow-PK-Prod } (\text{?PK } \alpha) \ ((r-x - [\text{poly } r-x \ i:]) \ \text{div } [-i, 1:]);$ 
  -::unit  $\leftarrow \text{assert-spmf } ((e \ \mathbf{g} \ \mathbf{g}) \wedge_{G_T} (1/((\text{?}\alpha \ \alpha) + (-i))) = (e \ p' \ w-B \otimes_{G_T} e$ 
     $(r' \div_{G_p} w-i) \ \mathbf{g}) \wedge_{G_T} (1/(\varphi-i - \text{poly } r-x \ i)))$ ;
  return-spmf True
} ELSE return-spmf False
by (force simp add: t-BSDH.game-alt-def[of (reduction  $\mathcal{A}$ )])

```

We show the thesis in ennreal, which implies the plain thesis

```

have ennreal (spmf (t-BSDH.game (ext-reduction  $\mathcal{A}$ )) True)
   $\leq \text{ennreal } (\text{spmf } (\text{t-BSDH.game } (\text{reduction } \mathcal{A})) \ \text{True})$ 
unfolding bind-red-ext eval-bind-red-ext
apply (simp add: spmf-try-spmf ennreal-spmf-bind)
apply (rule nn-integral-mono)+
apply (simp add: assert-spmf-def)
apply (simp add: measure-spmf.emmeasure-eq-measure)
done

```

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

Finally we put everything together: we conclude that for every efficient adversary the advantage of winning the evaluation binding game for the batched KZG is less than or equal to breaking the t-BSDH assumption.

theorem *evaluation-binding*: *bind-advantage* $\mathcal{A} \leq \text{t-BSDH.} \text{advantage } (\text{reduction } \mathcal{A})$

```

using evaluation-binding-ext-red overestimate-reductions
unfolding t-BSDH.advantage-def
by algebra

end

```

```

end
theory BatchKZG-knowledge-sound

```

```

imports BatchKZG-eval-bind Algebraic-Group-Model
begin

```

21 Knowledge Soundness of the batched KZG

In this theory we prove knowledge soundness for the KZG, concretely the knowledge soundness as defined in the abstract polynomial commitment scheme. The proof is a reduction to the evaluation binding game which has been reduced to the t-Bilinear strong Diffie-Hellman problem in the `BatchKZG_eval_bind` theory.

```

hide-const restrict

```

```

locale BatchEvalKZG-PCS-knowledge-sound = BatchEvalKZG-PCS-eval-bind
begin

```

the AGM adversary types that are useful in defining reductions (i.e. the reduction to the evaluation binding game)

```

lift-to-algebraicT ('a ck, 'a commit, 'state) knowledge-soundness-adversary1  $G_p$ 

  => AGM-knowledge-soundness-adversary1
lift-to-algebraicT ('state, 'a ck, 'e mod-ring, 'e evaluation, 'a witness) knowledge-soundness-adversary2
   $G_p$  => AGM-knowledge-soundness-adversary2

```

We adapt the knowledge soundness adversary in round 2 to the batch version. Concretely, we ask the adversary to give a single point $i \in I$ at which the evaluation polynomial r is distinct from the extractor polynomial

```

type-synonym ('e', 'state', 'a') knowledge-soundness-adversary2-AGM
  = ('a' ck  $\times$  'state')  $\Rightarrow$  ('e' mod-ring  $\times$  'e' argument set  $\times$  ('e' batch-evaluation
   $\times$  ('a' witness  $\times$  int list))) spmf

```

The extractor is an algorithm that plays against the adversary. It is granted access to the adversaries messages and state (which we neglect in this case as we do not need it because the calculation vector is enough to create sensible values) and has to come up with a polynomial such that the adversary cannot create valid opening points that are not part of the polynomial.

```

type-synonym ('a', 'e') extractor =

```

$(\text{'a' commit} \times \text{int list}) \Rightarrow$
 $(\text{'e' mod-ring poly} \times \text{unit}) \text{ spmf}$

restrict for AGM adversaries 1 and 2

interpretation *AGM1: Algebraic-Algorithm* $G_p \text{ listS } G_p.\text{groupS} \text{ prodC } G_p.\text{groupC}$
 noConstrain
by (unfold-locals)

$\text{'ck'} \Rightarrow \text{'state'} \Rightarrow (\text{'argument'} \times (\text{'evaluation'} \times \text{'witness'})) \text{ spmf}$

interpretation *AGM2: Algebraic-Algorithm* $G_p \text{ prodS } (\text{listS } G_p.\text{groupS}) \text{ noSelect}$

$\text{prodC noConstrain } (\text{prodC noConstrain } (\text{prodC noConstrain } G_p.\text{groupC}))$
by (unfold-locals)

definition *knowledge-soundness-game-AGM* :: $(\text{'state'}, \text{'a'}) \text{ AGM-knowledge-soundness-adversary1}$

$\Rightarrow (\text{'e'}, \text{'state'}, \text{'a'}) \text{ knowledge-soundness-adversary2-AGM} \Rightarrow (\text{'a'}, \text{'e'}) \text{ extractor} \Rightarrow$
 bool spmf

where *knowledge-soundness-game-AGM* $\mathcal{A1} \mathcal{A2} \mathcal{E} = \text{TRY do } \{$
 $(\text{ck}, \text{vk}) \leftarrow \text{key-gen};$
 $((\text{c}, \text{cvec}), \sigma) \leftarrow \text{AGM1.restrict } \mathcal{A1} \text{ ck};$
 $(\text{p}, \text{td}) \leftarrow \mathcal{E} (\text{c}, \text{cvec});$
 $(\text{i}, \text{I}, (\text{r}, (\text{w}, \text{wvec}))) \leftarrow \text{AGM2.restrict } \mathcal{A2} (\text{ck}, \sigma);$
 $\text{let } (\text{p-i}, \text{w'}) = \text{Eval ck td p i};$
 $\text{return-spmf } (\text{verify-eval-batch vk c I (r, w)} \wedge \text{poly r i} \neq \text{p-i}$
 $\wedge \text{valid-argument-batch I} \wedge \text{valid-eval-batch (r, w)} \wedge \text{i} \in \text{I})$
 $\} \text{ ELSE return-spmf False}$

definition *ks-to-eval-bind-reduction* :: $(\text{'state'}, \text{'a'}) \text{ AGM-knowledge-soundness-adversary1}$

$\Rightarrow (\text{'e'}, \text{'state'}, \text{'a'}) \text{ knowledge-soundness-adversary2-AGM} \Rightarrow (\text{'a'}, \text{'e'}) \text{ extractor}$
 $\Rightarrow (\text{'a'}, \text{'e'}) \text{ adversary where}$
ks-to-eval-bind-reduction $\mathcal{A1} \mathcal{A2} \mathcal{E} \text{ ck} = \text{do } \{$
 $((\text{c}, \text{cvec}), \sigma) \leftarrow \text{AGM1.restrict } \mathcal{A1} \text{ ck};$
 $(\text{p}, \text{td}) \leftarrow \mathcal{E} (\text{c}, \text{cvec});$
 $(\text{i}, \text{I}, (\text{r-x}, (\text{w}, \text{wvec}))) \leftarrow \text{AGM2.restrict } \mathcal{A2} (\text{ck}, \sigma);$
 $\text{let } (\text{p-i}, \text{w'}) = \text{Eval ck td p i};$
 $\text{return-spmf } (\text{c}, \text{i}, \text{p-i}, \text{w'}, \text{I}, \text{w}, \text{r-x})$
 $\}$

Extractor definition

fun $E :: (\text{'a'}, \text{'e'}) \text{ extractor where}$
 $E (\text{c}, \text{cvec}) = \text{return-spmf } (\text{Poly } (\text{map } (\text{of-int-mod-ring} :: \text{int} \Rightarrow \text{'e mod-ring'}) \text{ cvec}), ())$

lemma *reduction-map-imp:*

$\text{let ck} = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1];$
 $\text{vk} = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1];$
 $(\text{p-i}, \text{w-i}) = \text{Eval ck td } (\text{Poly } (\text{map } (\text{of-int-mod-ring} :: \text{int} \Rightarrow \text{'e mod-ring'})$
 $\text{cvec})) \text{ i}$

```

in
  length ck = length cvec
  ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
  ∧ verify-eval-batch vk c I (r-x,w) ∧ poly r-x i ≠ p-i
  ∧ valid-argument-batch I ∧ valid-eval-batch (r-x,w) ∧ i ∈ I
→
  length ck = length cvec
  ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
  ∧ i ∈ I ∧ p-i ≠ poly r-x i
  ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w)
  ∧ verify-eval vk c i (p-i, w-i)
  ∧ verify-eval-batch vk c I (r-x,w)
proof -
  define ck where ck-def: ck = map (λt. gGp ⌈Gp (α⌈t)) [0..<max-deg+1]
  define vk where vk-def: vk = map (λt. gGp ⌈Gp (α⌈t)) [0..<max-deg+1]
  define p-i where p-i-def: p-i = fst (Eval ck td (Poly (map (of-int-mod-ring::int
⇒'e mod-ring) cvec)) i )
  define w-i where w-i-def: w-i = snd (Eval ck td (Poly (map (of-int-mod-ring::int
⇒'e mod-ring) cvec)) i )

  have length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ verify-eval-batch vk c I (r-x,w) ∧ poly r-x i ≠ p-i
    ∧ valid-argument-batch I ∧ valid-eval-batch (r-x,w) ∧ i ∈ I
    →
      length ck = length cvec
      ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
      ∧ i ∈ I ∧ p-i ≠ poly r-x i
      ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w)
      ∧ verify-eval vk c i (p-i, w-i)
      ∧ verify-eval-batch vk c I (r-x,w)
  (is ?lhs → ?rhs)
proof
  assume asm: ?lhs
  show ?rhs
  proof(intro conjI)
    from asm show length ck = length cvec by blast
    from asm show c = fold (λ(i::nat) acc::'a. acc ⊗ ck ! i [⌈ cvec ! i) [0..<length
ck] 1 by blast
    from asm show i ∈ I by blast
    from asm show p-i ≠ poly r-x i by blast
    from asm show valid-argument-batch I by blast
    from asm show valid-eval-batch (r-x, w) by blast
    from asm show verify-eval-batch vk c I (r-x, w) by blast
    show valid-eval (p-i, w-i)
  proof -
    have g-pow-PK-Prod ck (ψ-of (Poly (map of-int-mod-ring cvec)) i)
    = gGp ⌈Gp (poly (ψ-of (Poly (map of-int-mod-ring cvec)) i) α)
    unfolding ck-def

```

```

proof (rule g-pow-PK-Prod-correct)
  show degree ( $\psi$ -of (Poly (map of-int-mod-ring cvec)) i)  $\leq$  max-deg
  proof (rule le-trans[OF degree-q-le- $\varphi$ ])
    have length (map of-int-mod-ring cvec) = max-deg + 1
    using asm unfolding ck-def by force
    moreover have length (coeffs (Poly (map of-int-mod-ring cvec)))  $\leq$ 
length (map of-int-mod-ring cvec)
    by (metis coeffs-Poly length-map length-strip-while-le)
    ultimately show degree (Poly (map of-int-mod-ring cvec))  $\leq$  max-deg
    using degree-eq-length-coeffs[of Poly (map of-int-mod-ring cvec)]
    by (metis le-diff-conv)
  qed
qed
then show ?thesis
  unfolding valid-eval-def
  by (simp add: Eval-def p-i-def w-i-def)
qed
show verify-eval vk c i (p-i, w-i)
proof -
  let ?cvec = (map of-int-mod-ring cvec::'e mod-ring list)

  have length-cvec: length ?cvec = max-deg + 1
  using asm unfolding ck-def by force
  moreover have length (coeffs (Poly ?cvec))  $\leq$  length ?cvec
  by (metis coeffs-Poly length-strip-while-le)
  ultimately have deg-poly-calc-vec-le-max-deg: degree (Poly ?cvec)  $\leq$  max-deg
  using degree-eq-length-coeffs[of Poly ?cvec]
  by (metis le-diff-conv)

  have 1: (g-pow-PK-Prod (map ( $\lambda t$ .  $\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha \hat{\phantom{g}} t$ ) [0.. $\text{max-deg} + 1$ ])
    ( $\psi$ -of (Poly ?cvec) i))
    = ( $\mathbf{g} \hat{\phantom{g}}_{G_p} \text{poly } (\psi\text{-of } (\text{Poly } ?\text{cvec}) i) \alpha$ )
  proof(rule g-pow-PK-Prod-correct)
    show degree ( $\psi$ -of (Poly ?cvec) i)  $\leq$  max-deg
    by (rule le-trans[OF degree-q-le- $\varphi$ ])(fact deg-poly-calc-vec-le-max-deg)
  qed

  have 2: map ( $\lambda t$ .  $\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha \hat{\phantom{g}} t$ ) [0.. $\text{max-deg} + 1$ ] ! 1 =  $\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha$ 
    by (metis (no-types, lifting) One-nat-def add.commute d-pos diff-zero
le-add-same-cancel1 le-zero-eq length-upt nth-map nth-upt plus-1-eq-Suc power-one-right
zero-compare-simps(1))

  have 3: ( $\mathbf{g} \hat{\phantom{g}}_{G_p} \text{poly } (\text{Poly } ?\text{cvec}) \alpha$ ) = c
  proof -
    have ( $\mathbf{g} \hat{\phantom{g}}_{G_p} \text{poly } (\text{Poly } ?\text{cvec}) \alpha$ )
      = g-pow-PK-Prod (map ( $\lambda t$ .  $\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha \hat{\phantom{g}} t$ ) [0.. $\text{max-deg} + 1$ ]) (Poly
?cvec)
    by (rule g-pow-PK-Prod-correct[symmetric])(fact deg-poly-calc-vec-le-max-deg)
    also have g-pow-to-fold: ... = fold ( $\lambda i \text{ acc. acc} \otimes_{G_p} (\mathbf{g} \hat{\phantom{g}}_{G_p} \alpha \hat{\phantom{g}} i)$ )

```

```

 $\hat{\phantom{x}}_{G_p}$  (poly.coeff (Poly ?cvec) i))
  [0.. $\text{Suc}$  (degree (Poly ?cvec))]  $\mathbf{1}_{G_p}$ 
  by (rule g-pow-PK-Prod-to-fold)(fact deg-poly-calc-vec-le-max-deg)
  also have ...
  = fold ( $\lambda$  i acc. acc  $\otimes_{G_p}$  ( $\mathbf{g}_{G_p} \hat{\phantom{x}}_{G_p} (\alpha \hat{\phantom{x}} i)$ )  $\hat{\phantom{x}}_{G_p}$  (?cvec!i)) [0.. $\text{max-deg}+1$ ]
 $\mathbf{1}_{G_p}$ 
  proof -
    have fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! i) [0.. $\text{max-deg} +$ 
    1]  $\mathbf{1}$ 
      = fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! i)
      ([0.. $\text{Suc}$  (degree (Poly ?cvec))] @ [ $\text{Suc}$  (degree (Poly ?cvec)).. $\text{max-deg}$ 
      + 1])
       $\mathbf{1}$ 
    proof -
      have  $\text{Suc}$  (degree (Poly ?cvec))  $\leq$   $\text{max-deg} + 1$ 
      by (simp add: deg-poly-calc-vec-le-max-deg)
      then show ?thesis
      by (metis (lifting) nat-le-iff-add upt-add-eq-append zero-order(1))
    qed
    also have ... = fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! i)
      [ $\text{Suc}$  (degree (Poly ?cvec)).. $\text{max-deg} + 1$ ]
      (fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! i)
      [0.. $\text{Suc}$  (degree (Poly ?cvec))]  $\mathbf{1}$ )
      by fastforce
    also have ... = fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  poly.coeff (Poly
    ?cvec) i)
      [0.. $\text{Suc}$  (degree (Poly ?cvec))]
       $\mathbf{1}$ 
    proof -
      have fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! i) [0.. $\text{Suc}$  (degree
      (Poly ?cvec))]  $\mathbf{1}$ 
      = fold ( $\lambda$  i acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} i$ )  $\hat{\phantom{x}}_{G_p}$  poly.coeff (Poly ?cvec) i)
      [0.. $\text{Suc}$  (degree (Poly ?cvec))]  $\mathbf{1}$ 
      proof (rule List.fold-cong)
        show  $\bigwedge x. x \in \text{set } [0.. $\text{Suc}$  (degree (Poly ?cvec))] \implies$ 
          ( $\lambda$  acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} x$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! x) =
          ( $\lambda$  acc. acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} x$ )  $\hat{\phantom{x}}_{G_p}$  poly.coeff (Poly ?cvec) x)
        proof
          fix x::nat
          fix acc::'a
          assume asm:  $x \in \text{set } [0.. $\text{Suc}$  (degree (Poly ?cvec))]$ 
          then have ?cvec ! x = poly.coeff (Poly ?cvec) x
          by (metis  $\langle \text{length } ?\text{cvec} = \text{max-deg} + 1 \rangle$  atLeastLessThan-iff co-
          eff-Poly deg-poly-calc-vec-le-max-deg dual-order.trans less-Suc-eq-le nth-default-nth
          semiring-norm(174) set-upt)
          then show acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha \hat{\phantom{x}} x$ )  $\hat{\phantom{x}}_{G_p}$  ?cvec ! x = acc  $\otimes$  ( $\mathbf{g} \hat{\phantom{x}}_{G_p} \alpha$ 
           $\hat{\phantom{x}} x$ )  $\hat{\phantom{x}}_{G_p}$  poly.coeff (Poly ?cvec) x
          by presburger
        qed
      qed
    qed
  
```

```

qed
qed simp+
moreover have  $\forall \text{init} \in \text{carrier } G_p.$ 
  fold  $(\lambda i \text{ acc. acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} i) \hat{\wedge}_{G_p} ?cvec ! i)$ 
     $[Suc (\text{degree } (Poly ?cvec))..<max-deg + 1]$ 
     $\text{init}$ 
  =  $\text{init}$ 
proof
  fix  $\text{init} :: 'a$ 
  assume  $\text{init-in-carrier: init} \in \text{carrier } G_p$ 
  have fold  $(\lambda i \text{ acc. acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} i) \hat{\wedge}_{G_p} ?cvec ! i)$ 
     $[Suc (\text{degree } (Poly ?cvec))..<max-deg + 1]$ 
     $\text{init} = \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1})$ 
     $[Suc (\text{degree } (Poly ?cvec))..<max-deg + 1]$ 
     $\text{init}$ 
  proof (rule List.fold-cong)
    show  $\bigwedge x. x \in \text{set } [Suc (\text{degree } (Poly ?cvec))..<max-deg + 1] \implies$ 
       $(\lambda \text{acc. acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} x) \hat{\wedge}_{G_p} ?cvec ! x) = (\lambda \text{acc. acc} \otimes \mathbf{1})$ 
    proof
      fix  $x::nat$ 
      fix  $\text{acc} :: 'a$ 
      assume  $\text{asm: } x \in \text{set } [Suc (\text{degree } (Poly ?cvec))..<max-deg + 1]$ 
      show  $\text{acc} \otimes (\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} x) \hat{\wedge}_{G_p} ?cvec ! x = \text{acc} \otimes \mathbf{1}$ 
      proof -
        have  $?cvec ! x = 0$  using  $\text{asm length-cvec}$ 
        by (smt (verit) add.commute coeff-Poly-eq in-set-conv-nth
le-degree length-upt less-diff-conv not-less-eq-eq nth-default-eq-dflt-iff nth-upt or-
der.refl trans-le-add2)
        then have  $(\mathbf{g} \hat{\wedge}_{G_p} \alpha \hat{\wedge} x) \hat{\wedge}_{G_p} ?cvec ! x = \mathbf{1}$  by simp
        then show  $?thesis$  by argo
      qed
    qed
  qed simp+
  also have  $\dots = \text{init}$ 
  proof (induction  $\text{max-deg}$ )
    case 0
    then show  $?case$  by fastforce
  next
    case  $(Suc \text{max-deg})$ 
    have fold  $(\lambda i \text{ acc. acc} \otimes \mathbf{1}) [Suc (\text{degree } (Poly ?cvec))..<Suc \text{max-deg}$ 
+ 1]  $\text{init}$ 
      = fold  $(\lambda i \text{ acc. acc} \otimes \mathbf{1}) ([Suc (\text{degree } (Poly ?cvec))..<max-deg +$ 
1] @  $[Suc \text{max-deg}])$   $\text{init}$ 
      by (simp add:  $\text{init-in-carrier}$ )
    also have  $\dots = \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1}) [Suc \text{max-deg}] (\text{fold } (\lambda i \text{ acc.}$ 
 $\text{acc} \otimes \mathbf{1}) [Suc (\text{degree } (Poly ?cvec))..<max-deg + 1] \text{init})$ 
      by force
    also have  $\dots = \text{fold } (\lambda i \text{ acc. acc} \otimes \mathbf{1}) [Suc \text{max-deg}] \text{init}$  using
Suc.IH by argo

```

```

    also have ... = init  $\otimes$  1 by force
    also have ... = init by (simp add: init-in-carrier)
    finally show ?case .
qed
finally show fold ( $\lambda i$  acc. acc  $\otimes$  ( $\mathbf{g} \hat{G}_p \alpha \hat{t}$ )  $\hat{G}_p ?cvec ! i$ )
  [Suc (degree (Poly ?cvec)).. $\max\text{-deg} + 1$ ]
  init
  = init .
qed
ultimately show ?thesis
  by (metis (no-types, lifting)  $G_p$ .generator-closed  $G_p$ .int-pow-closed  $\langle \mathbf{g} \hat{G}_p \text{poly} (\text{Poly} ?cvec) \alpha = g\text{-pow-PK-Prod} (\text{map} (\lambda t. \mathbf{g} \hat{G}_p \alpha \hat{t}) [0..\max\text{-deg} + 1]) (\text{Poly} ?cvec) \rangle g\text{-pow-to-fold}$ )
qed
finally show ?thesis by presburger
qed
also have ...
  = fold ( $\lambda i$  acc. acc  $\otimes_{G_p} (\text{map} (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..\max\text{-deg} + 1]) ! i$ 
 $\hat{G}_p (?cvec ! i)) [0..\max\text{-deg} + 1] \mathbf{1}_{G_p}$ 
  proof(rule List.fold-cong)
    show 1 = 1 by simp
    show  $[0..\max\text{-deg} + 1] = [0..\max\text{-deg} + 1]$  by simp
    show  $\bigwedge x. x \in \text{set } [0..\max\text{-deg} + 1] \implies$ 
      ( $\lambda \text{acc. acc} \otimes (\mathbf{g} \hat{G}_p \alpha \hat{x}) \hat{G}_p ?cvec ! x$ ) =
      ( $\lambda \text{acc. acc} \otimes \text{map} (\lambda t. \mathbf{g} \hat{G}_p \alpha \hat{t}) [0..\max\text{-deg} + 1] ! x \hat{G}_p ?cvec !$ 
 $x$ )
    proof
      fix  $x :: \text{nat}$ 
      fix  $\text{acc} :: 'a$ 
      assume asm:  $x \in \text{set } [0..\max\text{-deg} + 1]$ 
      show  $\text{acc} \otimes (\mathbf{g} \hat{G}_p \alpha \hat{x}) \hat{G}_p ?cvec ! x$ 
        =  $\text{acc} \otimes \text{map} (\lambda t. \mathbf{g} \hat{G}_p \alpha \hat{t}) [0..\max\text{-deg} + 1] ! x \hat{G}_p ?cvec ! x$ 
        using PK-i[symmetric] asm
        by (metis Suc-eq-plus1 atLeastLessThan-iff less-Suc-eq-le set-upt)
    qed
  qed
also have ...
  = fold ( $\lambda i$  acc. acc  $\otimes_{G_p} (\text{map} (\lambda t. \mathbf{g}_{G_p} \hat{G}_p (\alpha \hat{t})) [0..\max\text{-deg} + 1]) ! i$ 
 $\hat{G}_p (\text{of-int-mod-ring } (cvec ! i))) [0..\max\text{-deg} + 1] \mathbf{1}_{G_p}$ 
  proof(rule List.fold-cong)
    fix  $x$ 
    assume  $x \in \text{set } [0..\max\text{-deg} + 1]$ 
    then have  $x < \text{length } cvec$ 
      using asm unfolding ck-def
      by fastforce
    then show ( $\lambda \text{acc. acc} \otimes \text{map} (\lambda t. \mathbf{g} \hat{G}_p \alpha \hat{t}) [0..\max\text{-deg} + 1] ! x \hat{G}_p$ 
 $\text{map of-int-mod-ring } cvec ! x$ ) =
      ( $\lambda \text{acc. acc} \otimes \text{map} (\lambda t. \mathbf{g} \hat{G}_p \alpha \hat{t}) [0..\max\text{-deg} + 1] ! x \hat{G}_p \text{of-int-mod-ring}$ 

```

```

(cvec ! x))
  by force
qed simp+
also have ... = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)] [0..<length ck] 1
proof -
  have length-eq-max-deg: length (map (λt. g ^ α ^ t) [0..<max-deg + 1])
= max-deg + 1
  by force
  have mod-ring-trnsf-eq-plain: ⋀g x. g ∈ carrier Gp ⇒ g [⌈Gp
(to-int-mod-ring (of-int-mod-ring x::'e mod-ring)) = g [⌈Gp x
proof -
  fix g x
  assume g-in-carrier: g ∈ carrier Gp
  have mod-red: to-int-mod-ring (of-int-mod-ring x::'e mod-ring) = x mod
p
  unfolding of-int-mod-ring-def to-int-mod-ring-def
  by (metis CARD-q of-int-mod-ring.rep-eq of-int-mod-ring-def
to-int-mod-ring.rep-eq to-int-mod-ring-def)
  then show g [⌈Gp (to-int-mod-ring (of-int-mod-ring x::'e mod-ring))
= g [⌈Gp x
  using carrier-pow-mod-order-Gp g-in-carrier mod-red by metis
qed
show ?thesis
proof(rule List.fold-cong)
  fix x
  assume x ∈ set [0..<max-deg + 1]
  then show (λacc. acc ⊗ map (λt. g ^ α ^ t) [0..<max-deg + 1] ! x ^
of-int-mod-ring (cvec ! x)) = (λacc. acc ⊗ ck ! x [⌈ cvec ! x)
  unfolding ck-def length-eq-max-deg using mod-ring-trnsf-eq-plain
  by (metis (no-types, lifting) Gp.generator-closed Gp.int-pow-closed
atLeastLessThan-iff length-upt nth-map set-upt verit-minus-simplify(2))
  qed (simp add: ck-def)+
qed
also have ... = c
  using asm unfolding ck-def by fast
  finally show ?thesis .
qed
show ?thesis
  unfolding verify-eval-def Eval-def Let-def split-def g-pow-PK-Prod-correct
  using eq-on-e[of (Poly ?cvec) i α]
  by (metis 1 2 3 Eval-def ck-def vk-def p-i-def w-i-def eq-on-e fst-conv
snd-conv)
qed
qed
qed
then show ?thesis
  unfolding ck-def vk-def p-i-def w-i-def Let-def split-def
  by blast
qed

```

theorem *ks-game-to-eval-bind-game*: *spmf* (knowledge-soundness-game-AGM $\mathcal{A}1$ $\mathcal{A}2$ E) *True*
 $\leq$ *spmf* (bind-game (ks-to-eval-bind-reduction $\mathcal{A}1$ $\mathcal{A}2$ E)) *True*
(is ?lhs $\leq$?rhs)

proof –
note [*simp*] = *Let-def split-def*

have *spmf* (knowledge-soundness-game-AGM $\mathcal{A}1$ $\mathcal{A}2$ E) *True* = *spmf* (TRY do {
{
 $(ck, vk) \leftarrow \text{key-gen};$
 $((c, cvec), \sigma) \leftarrow \mathcal{A}1\ ck;$
- :: *unit* $\leftarrow \text{assert-spmf}$ ($\text{length } ck = \text{length } cvec$
 $\wedge c = \text{fold } (\lambda\ i\ acc.\ acc \otimes ck!i\ [\bigwedge] (cvec!i))\ [0..\text{length } ck]\ \mathbf{1}$);
 $(p, td) \leftarrow E\ (c, cvec);$
 $(i, I, (r, (w, wvec))) \leftarrow \text{AGM2.restrict } \mathcal{A}2\ (ck, \sigma);$
 $\text{let } (p-i, w') = \text{Eval } ck\ td\ p\ i;$
 $\text{return-spmf } (\text{verify-eval-batch } vk\ c\ I\ (r, w) \wedge \text{poly } r\ i \neq p-i$
 $\wedge \text{valid-argument-batch } I \wedge \text{valid-eval-batch } (r, w) \wedge i \in I)$
} ELSE *return-spmf False*) *True*
unfolding *knowledge-soundness-game-AGM-def AGM1.restrict-def listS-def*
 $G_p.\text{groupS-def noSelect-def}$
 $\text{Restrictive-Comp.restrict-def prodC-def } G_p.\text{groupC-def } G_p.\text{constrain-grp-def}$
 noConstrain-def
by force
also have ... = *spmf* (TRY do {
 $(ck, vk) \leftarrow \text{key-gen};$
TRY do {
 $((c, cvec), \sigma) \leftarrow \mathcal{A}1\ ck;$
TRY do {
- :: *unit* $\leftarrow \text{assert-spmf}$ ($\text{length } ck = \text{length } cvec$
 $\wedge c = \text{fold } (\lambda\ i\ acc.\ acc \otimes ck!i\ [\bigwedge] (cvec!i))\ [0..\text{length } ck]\ \mathbf{1}$);
TRY do {
 $(p, td) \leftarrow E\ (c, cvec);$
TRY do {
 $(i, I, (r, (w, wvec))) \leftarrow \text{AGM2.restrict } \mathcal{A}2\ (ck, \sigma);$
 $\text{let } (p-i, w') = \text{Eval } ck\ td\ p\ i;$
TRY do {
 $\text{return-spmf } (\text{verify-eval-batch } vk\ c\ I\ (r, w) \wedge \text{poly } r\ i \neq p-i$
 $\wedge \text{valid-argument-batch } I \wedge \text{valid-eval-batch } (r, w) \wedge i \in I)$
} ELSE *return-spmf False*
} ELSE *return-spmf False*
} ELSE *return-spmf False*
} ELSE *return-spmf False*
} ELSE *return-spmf False*) *True*
unfolding *Let-def split-def*
by ($\text{fold try-bind-spmf-lossless2}[OF\ \text{lossless-return-spmf}])\ \text{simp}$

```

also have ... = spmf (TRY do {
  (ck,vk) ← key-gen;
  TRY do {
    ((c,cvec),σ) ← A1 ck;
    TRY do {
      - :: unit ← assert-spmf (length ck = length cvec
        ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..length ck] 1);
      TRY do {
        (p,td) ← E (c,cvec);
        TRY do {
          (i, I, (r, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
          let (p-i,w') = Eval ck td p i;
          TRY do {
            - :: unit ← assert-spmf (verify-eval-batch vk c I (r,w) ∧ poly r i ≠ p-i
              ∧ valid-argument-batch I ∧ valid-eval-batch (r,w) ∧ i ∈ I);
            return-spmf True
          } ELSE return-spmf False
        } ELSE return-spmf False
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False) True
  by (rule spmf-eqI')(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1
intro!: try-spmf-cong bind-spmf-cong)
  also have ... = spmf (TRY do {
    (ck,vk) ← key-gen;
    ((c,cvec),σ) ← A1 ck;
    - :: unit ← assert-spmf (length ck = length cvec
      ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..length ck] 1);
    (p,td) ← E (c,cvec);
    (i, I, (r, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
    let (p-i,w') = Eval ck td p i;
    - :: unit ← assert-spmf (verify-eval-batch vk c I (r,w) ∧ poly r i ≠ p-i
      ∧ valid-argument-batch I ∧ valid-eval-batch (r,w) ∧ i ∈ I);
    return-spmf True
  } ELSE return-spmf False) True
  unfolding Let-def split-def
  by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])simp
also have ... = spmf(TRY do {
  (ck,vk) ← key-gen;
  ((c,cvec),σ) ← A1 ck;
  (p,td) ← E (c,cvec);
  (i, I, (r, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w') = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..length ck] 1);
  - :: unit ← assert-spmf (verify-eval-batch vk c I (r,w) ∧ poly r i ≠ p-i
    ∧ valid-argument-batch I ∧ valid-eval-batch (r,w) ∧ i ∈ I);
  return-spmf True

```

```

} ELSE return-spmf False) True
apply (rule spmf-eqI')
apply (rule try-spmf-cong)
apply (rule unpack-bind-spmf'; simp)+
apply (subst assert-commute)
apply simp+
done
also have ... = spmf (TRY do {
  x :: nat ← sample-uniform (order Gp);
  let (α::'e mod-ring) = of-int-mod-ring (int x);
  let ck = map (λt. gGp ^Gp (α ^t)) [0..max-deg+1];
  let vk = map (λt. gGp ^Gp (α ^t)) [0..max-deg+1];
  ((c,cvec),σ) ← A1 ck;
  let (p,td) = (Poly (map (of-int-mod-ring::int ⇒'e mod-ring) cvec),());
  (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w') = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈] (cvec!i)) [0..length ck] 1
    ∧ verify-eval-batch vk c I (r-x,w) ∧ poly r-x i ≠ p-i
    ∧ valid-argument-batch I ∧ valid-eval-batch (r-x,w) ∧ i ∈ I);
  return-spmf True
} ELSE return-spmf False) True
by (simp add: key-gen-def Setup-def assert-collapse)
also have ... ≤ spmf (TRY do {
  x :: nat ← sample-uniform (order Gp);
  let (α::'e mod-ring) = of-int-mod-ring (int x);
  let ck = map (λt. gGp ^Gp (α ^t)) [0..max-deg+1];
  let vk = map (λt. gGp ^Gp (α ^t)) [0..max-deg+1];
  ((c,cvec),σ) ← A1 ck;
  let (p,td) = (Poly (map (of-int-mod-ring::int ⇒'e mod-ring) cvec),());
  (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w-i) = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈] (cvec!i)) [0..length ck] 1
    ∧ i ∈ I ∧ p-i ≠ poly r-x i
    ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w)
    ∧ verify-eval vk c i (p-i, w-i)
    ∧ verify-eval-batch vk c I (r-x,w));
  return-spmf True
} ELSE return-spmf False) True
unfolding E.simps key-gen-def Setup-def
apply (rule try-spmf-le)
apply (unfold Let-def split-def)
apply (rule bind-spmf-le)+
apply (rule assert-imp)
apply (insert reduction-map-imp)
apply (unfold Let-def split-def)
apply (intro impI)
apply simp

```

```

apply force
done
also have ... = spmf (TRY do {
  (ck,vk) ← key-gen;
  ((c,cvec),σ) ← A1 ck;
  (p,td) ← E (c,cvec);
  (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w-i) = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ i ∈ I ∧ p-i ≠ poly r-x i
    ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w)
    ∧ verify-eval vk c i (p-i, w-i)
    ∧ verify-eval-batch vk c I (r-x,w));
  return-spmf True
} ELSE return-spmf False) True
by (simp add: key-gen-def Setup-def assert-collapse)
also have ... = spmf (TRY do {
  (ck,vk) ← key-gen;
  ((c,cvec),σ) ← A1 ck;
  (p,td) ← E (c,cvec);
  (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w-i) = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ i ∈ I ∧ p-i ≠ poly r-x i
    ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w));
  -::unit ← assert-spmf (verify-eval vk c i (p-i, w-i)
    ∧ verify-eval-batch vk c I (r-x,w));
  return-spmf True
} ELSE return-spmf False) True
by (simp add: assert-collapse)
also have ... = spmf (TRY do {
  (ck,vk) ← key-gen;
  TRY do {
    ((c,cvec),σ) ← A1 ck;
    TRY do {
      (p,td) ← E (c,cvec);
      TRY do{
        (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
        TRY do{
          let (p-i,w-i) = Eval ck td p i;
          - :: unit ← assert-spmf (length ck = length cvec
            ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
            ∧ i ∈ I ∧ p-i ≠ poly r-x i
            ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch
(r-x, w));
          TRY do {
            -::unit ← assert-spmf (verify-eval vk c i (p-i, w-i)

```

```

       $\wedge$  verify-eval-batch vk c I (r-x,w));
    return-spmf True
  } ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False
} ELSE return-spmf False) True
unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])simp
also have ... = spmf (TRY do {
  (ck,vk)  $\leftarrow$  key-gen;
  TRY do {
    ((c,cvec), $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  ck;
    TRY do {
      (p,td)  $\leftarrow$  E (c,cvec);
      TRY do{
        (i, I, (r-x, (w, wvec)))  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
        TRY do{
          let (p-i,w-i) = Eval ck td p i;
          - :: unit  $\leftarrow$  assert-spmf (length ck = length cvec
             $\wedge$  c = fold ( $\lambda$  i acc. acc  $\otimes$  ck!i [ $\top$ ] (cvec!i)) [0..1
             $\wedge$  i  $\in$  I  $\wedge$  p-i  $\neq$  poly r-x i
             $\wedge$  valid-argument-batch I  $\wedge$  valid-eval (p-i, w-i)  $\wedge$  valid-eval-batch
(r-x, w));
          TRY do {
            return-spmf (verify-eval vk c i (p-i, w-i)
               $\wedge$  verify-eval-batch vk c I (r-x,w))
          } ELSE return-spmf False
        } ELSE return-spmf False
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False) True
by (rule spmf-eqI')(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1
intro!: try-spmf-cong bind-spmf-cong)
also have ... = spmf (TRY do {
  (ck,vk)  $\leftarrow$  key-gen;
  ((c,cvec), $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  ck;
  (p,td)  $\leftarrow$  E (c,cvec);
  (i, I, (r-x, (w, wvec)))  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
  let (p-i,w-i) = Eval ck td p i;
  - :: unit  $\leftarrow$  assert-spmf (length ck = length cvec
     $\wedge$  c = fold ( $\lambda$  i acc. acc  $\otimes$  ck!i [ $\top$ ] (cvec!i)) [0..1
     $\wedge$  i  $\in$  I  $\wedge$  p-i  $\neq$  poly r-x i
     $\wedge$  valid-argument-batch I  $\wedge$  valid-eval (p-i, w-i)  $\wedge$  valid-eval-batch (r-x, w));
  return-spmf (verify-eval vk c i (p-i, w-i)
     $\wedge$  verify-eval-batch vk c I (r-x,w))
  } ELSE return-spmf False) True

```

```

unfolding Let-def split-def
by (fold try-bind-spmf-lossless2[OF lossless-return-spmf])simp
also have ... = spmf (TRY do {
  (ck,vk) ← key-gen;
  ((c,cvec),σ) ← A1 ck;
  (p,td) ← E (c,cvec);
  (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w-i) = Eval ck td p i;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1 ]);
  - :: unit ← assert-spmf (i ∈ I ∧ p-i ≠ poly r-x i
    ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w));
  return-spmf (verify-eval vk c i (p-i, w-i)
    ∧ verify-eval-batch vk c I (r-x,w))
  } ELSE return-spmf False) True
by (simp add: assert-collapse)
also have ... = spmf (TRY do {
  (ck,vk) ← key-gen;
  ((c,cvec),σ) ← A1 ck;
  - :: unit ← assert-spmf (length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1 ]);
  (p,td) ← E (c,cvec);
  (i, I, (r-x, (w, wvec))) ← AGM2.restrict A2 (ck,σ);
  let (p-i,w-i) = Eval ck td p i;
  - :: unit ← assert-spmf (i ∈ I ∧ p-i ≠ poly r-x i
    ∧ valid-argument-batch I ∧ valid-eval (p-i, w-i) ∧ valid-eval-batch (r-x, w));
  return-spmf (verify-eval vk c i (p-i, w-i)
    ∧ verify-eval-batch vk c I (r-x,w))
  } ELSE return-spmf False) True
apply (rule spmf-eqI')
apply (unfold Let-def split-def)
apply (simp split: prod.split)
apply (rule unpack-try-spmf)
apply (rule unpack-bind-spmf'; rule ext)+
apply (subst assert-commute)
apply (subst assert-commute)
apply blast
done
also have ... ≤ spmf (bind-game (ks-to-eval-bind-reduction A1 A2 E)) True
unfolding bind-game-def ks-to-eval-bind-reduction-def
unfolding knowledge-soundness-game-AGM-def AGM1.restrict-def listS-def
Gp.groupS-def noSelect-def
Restrictive-Comp.restrict-def prodC-def Gp.groupC-def Gp.constrain-grp-def
noConstrain-def split-def Let-def
by force
finally show ?thesis .
qed

```

Finally we put everything together: we conclude that for every efficient ad-

versary in the AGM the advantage of winning the knowledge soundness game is less equal to breaking the t-BSDH assumption.

theorem *knowledge-soundness*:

```

  spmf (knowledge-soundness-game-AGM  $\mathcal{A}1$   $\mathcal{A}2$   $E$ ) True
    ≤  $t$ -BSDH.advantage (reduction (ks-to-eval-bind-reduction  $\mathcal{A}1$   $\mathcal{A}2$   $E$ ))
  using ks-game-to-eval-bind-game
  using evaluation-binding[of ks-to-eval-bind-reduction  $\mathcal{A}1$   $\mathcal{A}2$   $E$ ]
  unfolding bind-advantage-def
  using landau-o.R-trans by blast

```

end

end

References

- [1] G. Arnon, A. Chiesa, G. Fenzi, and E. Yogev. WHIR: Reed–solomon proximity testing with super-fast verification. Cryptology ePrint Archive, Paper 2024/1586, 2024. <https://eprint.iacr.org/2024/1586>.
- [2] D. Boneh and V. Shoup. A graduate course in applied cryptography. Online textbook, version 0.6, <https://toc.cryptobook.us/book.pdf>, 2023.
- [3] D. Butler, A. Lochbihler, D. Aspinall, and A. Gascón. Formalising Σ -protocols and commitment schemes using CryptHOL. *Journal of Automated Reasoning*, 65(4):521–567, 2021. Cryptology ePrint Archive, Paper 2019/1185. <https://eprint.iacr.org/2019/1185>.
- [4] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Paper 2019/1047, 2019. Published at EUROCRYPT 2020. <https://eprint.iacr.org/2019/1047>.
- [5] B. E. Diamond and J. Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023. <https://eprint.iacr.org/2023/1784>.
- [6] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 33–62. Springer, 2018.
- [7] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. <https://eprint.iacr.org/2019/953>.

- [8] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010. Extended version: Technical Report CACR 2010-10, Centre for Applied Cryptographic Research, University of Waterloo.
- [9] J. Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. Cryptology ePrint Archive, Paper 2020/1274, 2020. Published at TCC 2021. <https://eprint.iacr.org/2020/1274>.
- [10] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. Cryptology ePrint Archive, Paper 2019/099, 2019. Published at ACM CCS 2019. <https://eprint.iacr.org/2019/099>.
- [11] H. Zeilberger, B. Chen, and B. Fisch. BaseFold: Efficient field-agnostic polynomial commitment schemes from foldable codes. Cryptology ePrint Archive, Paper 2023/1705, 2023. Published at CRYPTO 2024. <https://eprint.iacr.org/2023/1705>.