Polynomial Commitment Schemes

Tobias Rothmann

July 14, 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

theory *Pairing*

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: $e \in$ *carrier* $G_p \to$ *carrier* $G_p \to$ *carrier* G_T **and**
e-bilinearity[*simp*]: $\forall$ *a b*::*int* . $\forall$ *P Q*. *P* $\in$ *carrier* $G_p \wedge$ *Q* $\in$ *carrier* $G_p \longrightarrow$
e (*P* $[\hat{}]_{G_p}$ *a*) (*Q* $[\hat{}]_{G_p}$ *b*)
= (*e P Q*) $[\hat{}]_{G_T}$ (*a*∗*b*) **and**

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

The pairing function e is linear in the first component

lemma *e-linear-in-fst*:
assumes $P \in \mathit{carrier}\ G_p \wedge Q \in \mathit{carrier}\ G_p$
shows $e\ (P\ [\hat{}]_{G_p}\ (a{::}\mathit{int}))\ Q = (e\ P\ Q)\ [\hat{}]_{G_T}\ a$
proof −
have $e\ (P\ [\hat{}]_{G_p}\ a)\ Q = e\ (P\ [\hat{}]_{G_p}\ a)\ (Q\ [\hat{}]_{G_p}\ (1{::}\mathit{int}))$ **using** *assms* **by** *simp*
also have ... $= (e\ P\ Q)\ [\hat{}]_{G_T}\ (a{*}(1{::}\mathit{int}))$ **using** *assms e-bilinearity* **by** *fast*
also have ...=$(e\ P\ Q)\ [\hat{}]_{G_T}\ a$ **by** *simp*
finally show $e\ (P\ [\hat{}]_{G_p}\ a)\ Q = (e\ P\ Q)\ [\hat{}]_{G_T}\ a$ **.**
qed

The pairing function e is linear in the second component

lemma *e-linear-in-snd*:
assumes $P \in \mathit{carrier}\ G_p \wedge Q \in \mathit{carrier}\ G_p$
shows $e\ P\ (Q\ [\hat{}]_{G_p}\ (a{::}\mathit{int})) = (e\ P\ Q)\ [\hat{}]_{G_T}\ a$
proof −
have $e\ P\ (Q\ \ [\hat{}]_{G_p}\ a) = e\ (P\ [\hat{}]_{G_p}\ (1{::}\mathit{int}))\ (Q\ [\hat{}]_{G_p}\ a)$ **using** *assms* **by** *simp*
also have ... $= (e\ P\ Q)\ [\hat{}]_{G_T}\ ((1{::}\mathit{int}){*}a)$ **using** *assms e-bilinearity* **by** *fast*
also have ...=$(e\ P\ Q)\ [\hat{}]_{G_T}\ a$ **by** *simp*
finally show $e\ P\ (Q\ [\hat{}]_{G_p}\ a) = (e\ P\ Q)\ [\hat{}]_{G_T}\ a$ **.**
qed

lemma *addition-in-exponents-on-e*[*simp*]:
assumes $x \in \mathit{carrier}\ G_p \wedge y \in \mathit{carrier}\ G_p$
shows $(e\ x\ y)\ [\hat{}]_{G_T}\ (a{::}\mathit{int}) \otimes_{G_T} (e\ x\ y)\ [\hat{}]_{G_T}\ (b{::}\mathit{int}) = (e\ x\ y)\ [\hat{}]_{G_T}\ (a{+}b)$
by (*metis* G_T*.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 $\forall\, P\ Q.\ P \in \mathit{carrier}\ G_p \wedge Q \in \mathit{carrier}\ G_p \longrightarrow e\ P\ Q = \mathbf{1}_{G_T}$
proof(*intro allI*)
fix $P\ Q$
show $P \in \mathit{carrier}\ G_p \wedge Q \in \mathit{carrier}\ G_p \longrightarrow e\ P\ Q = \mathbf{1}_{G_T}$
proof
assume $P \in \mathit{carrier}\ G_p \wedge Q \in \mathit{carrier}\ G_p$
then obtain $p\ q{::}\mathit{int}$ **where** $\mathbf{g}_{G_p}\ [\hat{}]_{G_p}\ p = P \wedge \mathbf{g}_{G_p}\ [\hat{}]_{G_p}\ q = Q$
by (*metis* G_p*.generatorE int-pow-int*)
then have $e\ P\ Q = e\ (\mathbf{g}_{G_p}\ [\hat{}]_{G_p}\ p)\ (\mathbf{g}_{G_p}\ [\hat{}]_{G_p}\ q)$
by *blast*
also have ... $= e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}\ [\hat{}]_{G_T}\ (p{*}q)$
by *force*
also have ... $= \mathbf{1}_{G_T}\ [\hat{}]_{G_T}\ (p{*}q)$

using *asm* **by** *argo*
also have $\ldots = \mathbf{1}_{G_T}$
by *fastforce*
finally show $e\ P\ Q = \mathbf{1}_{G_T}$ **.**
qed
qed
then show *False* **using** *e-non-degeneracy* **by** *blast*
qed

lemma *e-g-g-in-carrier-GT*[*simp*]: $e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p} \in carrier\ G_T$
using *e-symmetric* **by** *fast*

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

lemma *pow-on-eq-card-GT*[*simp*]: $(\mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ (x::int) = \mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ (y::int)) = ([x= y]\ (mod\ p))$
by (*metis* (*no-types*, *lifting*) *CARD-G*$_T$ *G*$_T$*.finite-carrier G*$_T$*.gen-power-0 G*$_T$*.generator-closed G*$_T$*.int-pow-eq G*$_T$*.ord-ge-1 G*$_T$*.ord-le-group-order G*$_T$*.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*]:
$((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}))\ [\widehat{}]_{G_T}\ x = ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}))\ [\widehat{}]_{G_T}\ y \longleftrightarrow [x= y]\ (mod\ p)$
by (*metis CARD-G*$_T$ *G*$_T$*.finite-carrier G*$_T$*.int-pow-eq G*$_T$*.ord-dvd-group-order G*$_T$*.ord-eq-1 G*$_T$*.ord-id G*$_T$*.pow-ord-eq-ord-iff G*$_T$*.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* G_p G_T p e
for G_p :: ('*a*, '*b*) *cyclic-group-scheme* (**structure**)
and G_T:: ('*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: *max-deg* > 0 **and**
CARD-q: *int* $(CARD('q)) = p$ **and**
d-l-p: *max-deg* $< p$
begin

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

abbreviation *div-in-grp* (**infixr** $\div_1$ *70*)
where $x \div_1 y \equiv x \otimes_1 inv_1\ y$

2.0.1 mod_ring operations on pow of Gp

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

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

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

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

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

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

lemma *to-int-mod-ring-ge-0*: *to-int-mod-ring x* ≥ *0*
using *range-to-int-mod-ring* **by** *fastforce*

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

proof
assume *assm*: $\mathbf{g}\ \hat{}_{G_p}\ x = \mathbf{g}\ \hat{}_{G_p}\ y$
then have $\mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ *to-int-mod-ring* $x = \mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ *to-int-mod-ring* y
using *assm* **by** *blast*
then have $\mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ *nat* (*to-int-mod-ring* x) = $\mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ *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 [*nat* (*to-int-mod-ring* x) = *nat* (*to-int-mod-ring* y)] (*mod order* G_p)
using G_p.*pow-generator-eq-iff-cong* G_p.*finite-carrier* **by** *fast*
then have [*to-int-mod-ring* x = *to-int-mod-ring* y] (*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 [*to-int-mod-ring* x = *to-int-mod-ring* y] (*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*: $\mathbf{g}\ \hat{}_{G_p}$ *of-int-mod-ring* $x = \mathbf{g}\ [\hat{}]\ x$
proof −
have $\mathbf{g}\ \hat{}_{G_p}$ *of-int-mod-ring* $x = \mathbf{g}\ [\hat{}]$ (x *mod* p)
by (*simp add*: *CARD-q of-int-mod-ring.rep-eq to-int-mod-ring.rep-eq*)
also have $\ldots = \mathbf{g}\ [\hat{}]\ x$ **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*: $\mathbf{g}\ \hat{}_{G_p}$ *of-int-mod-ring* $x\ \hat{}\ (t::nat) = \mathbf{g}\ [\hat{}]\ x\ \hat{}\ t$
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*: $c \neq 0 \Longrightarrow$ *inj-on* ($\lambda x.\ x\ \hat{}_{G_p}\ c$) (*carrier* G_p)
proof
fix $x\ y$
assume c: $c \neq 0$
assume x: $x \in$ *carrier* G_p
assume y: $y \in$ *carrier* G_p
assume *asm*: $x\ \hat{}\ c = y\ \hat{}\ c$
obtain *x-pow* **where** *x-pow*: $\mathbf{g}\ \hat{}_{G_p}$ *x-pow* = x
using x
by (*metis* G_p.*generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int*)
obtain *y-pow* **where** *y-pow*: $\mathbf{g}\ \hat{}_{G_p}$ *y-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** $\hat{}_{G_p}$ *x-pow*) $\hat{}_{G_p}$ *c* = (**g** $\hat{}_{G_p}$ *y-pow*) $\hat{}_{G_p}$ *c*
using *asm x-pow y-pow* **by** *blast*
then have (**g** $\hat{}_{G_p}$ (*x-pow*∗*c*))= (**g** $\hat{}_{G_p}$ (*y-pow*∗*c*))
using *mod-ring-pow-pow-*G_p **by** *presburger*
then have *x-pow* ∗ *c* = *y-pow*∗*c*
using *pow-on-eq-card* **by** *force*
then have *x-pow* = *y-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* ∈ *carrier* G_T ⟹ *g* [^]$_{G_T}$ *x* = *g* [^]$_{G_T}$ (*x mod p*)
proof −
assume *asmpt*: *g* ∈ *carrier* G_T
have *p*=(*order* G_T) **by** (*simp add*: *CARD-*G_T)
also have *g*[^]$_{G_T}$ *x* = *g* [^]$_{G_T}$ (*x mod order* G_T)
proof −
have *g* [^]$_{G_T}$ *x* ∈ *carrier* G_T **using** *asmpt* **by** *simp*
let *?d* = *x div* (*order* G_T)
have *g* [^]$_{G_T}$ *x* = *g* [^]$_{G_T}$ (*?d* ∗ *order* G_T + *x mod order* G_T)
using *div-mult-mod-eq* **by** *presburger*
also have ...=*g* [^]$_{G_T}$ (*?d* ∗ *order* G_T) ⊗$_{G_T}$ *g* [^]$_{G_T}$ (*x mod order* G_T)
using G_T.*int-pow-mult asmpt* **by** *fast*
also have ...=**1**$_{G_T}$ ⊗$_{G_T}$ *g* [^]$_{G_T}$ (*x mod 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* [^]$_{G_T}$ *x* = *g* [^]$_{G_T}$ (*x mod order* G_T)
using *asmpt* **by** *fastforce*
qed
finally show *g* [^]$_{G_T}$ *x* = *g* [^]$_{G_T}$ (*x mod p*) **.**
qed

lemma *mod-ring-pow-mult-*G_T[*symmetric*]: *x* ∈ *carrier* G_T ⟹ (*x* [^]$_{G_T}$ (*to-int-mod-ring* (*a*::′*q mod-ring*))) ⊗$_{G_T}$ (*x* [^]$_{G_T}$ (*to-int-mod-ring b*))
= *x* [^]$_{G_T}$ (*to-int-mod-ring* (*a*+*b*))
proof −
assume *asmpt*: *x* ∈ *carrier* G_T
have *x* [^]$_{G_T}$ *to-int-mod-ring a* ⊗$_{G_T}$ *x* [^]$_{G_T}$ *to-int-mod-ring b* = *x* [^]$_{G_T}$ (*to-int-mod-ring a* + *to-int-mod-ring b*)
by (*simp add*: G_T.*int-pow-mult asmpt*)
also have ...= *x* [^]$_{G_T}$ ((*to-int-mod-ring a* + *to-int-mod-ring b*) *mod* (*CARD* (′*q*)))
using *pow-mod-order-*G_T *CARD-q asmpt* **by** *blast*
also have ...= *x* [^]$_{G_T}$ *to-int-mod-ring* (*a* + *b*)

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

2.0.3 bilinearity operations for mod_ring elements

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

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

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

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

lemma *addition-in-exponents-on-e*[*simp*]:
assumes $x \in \textit{carrier}\ G_p \wedge y \in \textit{carrier}\ G_p$
shows $(e\ x\ y)\ \widehat{}_{G_T}\ a \otimes_{G_T} (e\ x\ y)\ \widehat{}_{G_T}\ b = (e\ x\ y)\ \widehat{}_{G_T}\ (a+b)$
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 $\forall P\ Q.\ P \in \textit{carrier}\ G_p \wedge Q \in \textit{carrier}\ G_p \longrightarrow e\ P\ Q = \mathbf{1}_{G_T}$
proof(*intro allI*)
fix $P\ Q$
show $P \in \textit{carrier}\ G_p \wedge Q \in \textit{carrier}\ G_p \longrightarrow e\ P\ Q = \mathbf{1}_{G_T}$
proof
assume $P \in \textit{carrier}\ G_p \wedge Q \in \textit{carrier}\ G_p$
then obtain $p\ q$::*int* **where** $\mathbf{g}_{G_p}\ [\widehat{}]_{G_p}\ p = P \wedge \mathbf{g}_{G_p}\ [\widehat{}]_{G_p}\ q = Q$
by (*metis* G_p*.generatorE int-pow-int*)
then have $e\ P\ Q = e\ (\mathbf{g}_{G_p}\ [\widehat{}]_{G_p}\ p)\ (\mathbf{g}_{G_p}\ [\widehat{}]_{G_p}\ q)$
by *blast*
also have $\ldots = e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}\ [\widehat{}]_{G_T}\ (p*q)$
by *force*
also have $\ldots = \mathbf{1}_{G_T}\ [\widehat{}]_{G_T}\ (p*q)$
using *asm* **by** *argo*
also have $\ldots = \mathbf{1}_{G_T}$
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*]: $(\mathbf{g}_{G_T}\ \widehat{}_{G_T}\ x = \mathbf{g}_{G_T}\ \widehat{}_{G_T}\ y) = (x{=}y)$
proof
assume *assm*: $\mathbf{g}_{G_T}\ \widehat{}_{G_T}\ x = \mathbf{g}_{G_T}\ \widehat{}_{G_T}\ y$
then have $\mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ \textit{to-int-mod-ring}\ x = \mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ \textit{to-int-mod-ring}\ y$
using *assm* **by** *blast*
then have $\mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ \textit{nat}\ (\textit{to-int-mod-ring}\ x) = \mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ \textit{nat}\ (\textit{to-int-mod-ring}\ y)$
using *to-int-mod-ring-ge-0*[*of x*] *to-int-mod-ring-ge-0*[*of y*] **by** *fastforce*
then have [$\textit{nat}\ (\textit{to-int-mod-ring}\ x) = \textit{nat}\ (\textit{to-int-mod-ring}\ y)$] ($\textit{mod order}\ G_T$)
using G_T*.pow-generator-eq-iff-cong* G_T*.finite-carrier* **by** *fast*
then have [$\textit{to-int-mod-ring}\ x = \textit{to-int-mod-ring}\ y$] ($\textit{mod order}\ G_T$)
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 [$\textit{to-int-mod-ring}\ x = \textit{to-int-mod-ring}\ y$] ($\textit{mod}\ p$)

```
    using CARD-G_T by fast
  then have to-int-mod-ring x = 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 g_G_T ^_G_T x = g_G_T ^_G_T y by fast
qed

lemma pow-on-eq-card-GT-carrier-ext'[simp]:
  ((e g_G_p g_G_p))^_G_T x = ((e g_G_p g_G_p))^_G_T y ⟷ x=y
proof
  assume g-pow-x-eq-g-pow-y: e g g ^_G_T x = e g g ^_G_T y
  obtain g-exp::nat where e g g = g_G_T [^]_G_T g-exp
    using G_T.generatorE e-g-g-in-carrier-GT by blast
  then have g-exp: e g g = g_G_T ^_G_T (of-int-mod-ring (int g-exp))
   by (metis CARD-G_T G_T.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 g g)^_G_T x =  g_G_T ^_G_T (of-int-mod-ring (int g-exp) * x)
    using g-exp
   by (metis CARD-G_T G_T.generator-closed G_T.int-pow-pow G_T.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 g g)^_G_T y = g_G_T ^_G_T (of-int-mod-ring (int g-exp) * y)
    using g-exp
   by (metis CARD-G_T G_T.generator-closed G_T.int-pow-pow G_T.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 g_G_p g_G_p) ^_G_T x = (e g_G_p g_G_p) ^_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* ⇒ ('*c*::*prime-card*) *mod-ring*
  and *exp* :: '*a* ⇒ '*c mod-ring* ⇒ '*a*
begin

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

The discrete Logarithm game

definition *game* :: ('*a*,'*c*) *adversary* ⇒ *bool spmf* **where**
  game $\mathcal{A}$ = *TRY do* {
    a ← *sample-uniform* (*Coset.order G*);
    a' ← $\mathcal{A}$ (*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* ⇒ *real*
  where *advantage* $\mathcal{A}$ = *spmf* (*game* $\mathcal{A}$) *True*

An alternative but equivalent game for the DL-game. This alternative game capsulates 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* {
    a ← *sample-uniform* (*Coset.order G*);
    a' ← $\mathcal{A}$ (*exp* **g** (*to-type a*));
    -::*unit* ← *assert-spmf* (*to-type a* = *a'*);
    return-spmf True
  } *ELSE return-spmf False*
  (**is** *?lhs* = *?rhs*)
proof −
   have *?lhs* = *TRY do* {
    a ← *sample-uniform* (*Coset.order G*);
    TRY do {
    a' ← $\mathcal{A}$ (*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* ⇒ (*'mr mod-ring*) *spmf*

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

definition *game* :: (*'a,'c*) *adversary* ⇒ *bool spmf* **where**
game $\mathcal{A}$ = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);
$\alpha' \leftarrow \mathcal{A}$ (*map* ($\lambda t'$. *exp* **g** ((*to-type* α)^t')) [*0..<t+1*]);
return-spmf (*to-type* $\alpha = \alpha'$)
} *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* ⇒ *real*
where *advantage* $\mathcal{A}$ = *spmf* (*game* $\mathcal{A}$) *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 $\mathcal{A}$ = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);
$\alpha' \leftarrow \mathcal{A}$ (*map* ($\lambda t'$. *exp* **g** ((*to-type* α)^t')) [*0..<t+1*]);
-::*unit* ← *assert-spmf* (*to-type* $\alpha = \alpha'$);
return-spmf True
} *ELSE return-spmf False*
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);
TRY do {
$\alpha' \leftarrow \mathcal{A}$ (*map* ($\lambda t'$. *exp* **g** ((*to-type* α)^t')) [*0..<t+1*]);
TRY return-spmf (*to-type* $\alpha = \alpha'$)
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* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);
TRY do {
$\alpha' \leftarrow \mathcal{A}$ (*map* ($\lambda t'$. *exp* **g** ((*to-type* α)^t')) [*0..<t+1*]);
TRY do {
- :: *unit* ← *assert-spmf* (*to-type* $\alpha = \alpha'$);
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 *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 *t-SDH* = *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* ⇒ ('*mr mod-ring* ∗'*grp*) *spmf*

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

definition *game* :: ('*a*,'*c*) *adversary* ⇒ *bool spmf* **where**
game $\mathcal{A}$ = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);
(*c*, *g*) ← $\mathcal{A}$ (*map* (λ*t'*. *exp* **g** ((*to-type* α)^*t'*)) [*0*..<*t*+*1*]);
return-spmf (*exp* **g** (*1*/((*to-type* α)+*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* ⇒ *real*
where *advantage* $\mathcal{A}$ = *spmf* (*game* $\mathcal{A}$) *True*

An alternative but equivalent game for the t-SDH-game. This alternative game capsulates 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 𝒜 = TRY do {
    α ← sample-uniform (Coset.order G);
    (c, g) ← 𝒜 (map (λt'. exp g ((to-type α) ^ t')) [0..<t+1]);
    -::unit ← assert-spmf (exp g (1/((to-type α)+c)) = g);
    return-spmf True
  } ELSE return-spmf False
  (is ?lhs = ?rhs)
proof −
  have ?lhs = TRY do {
    α ← sample-uniform (Coset.order G);
    TRY do {
      (c, g) ← 𝒜 (map (λt'. exp g ((to-type α) ^ t')) [0..<t+1]);
      TRY return-spmf (exp g (1/((to-type α)+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 {
    α ← sample-uniform (Coset.order G);
    TRY do {
      (c, g) ← 𝒜 (map (λt'. exp g ((to-type α) ^ t')) [0..<t+1]);
      TRY do {
        - :: unit ← assert-spmf (exp g (1/((to-type α)+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 tBSDH-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 a 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 + G_T : *cyclic-group* G_T
for G:: ('a, 'b) *cyclic-group-scheme* (**structure**) **and** G_T :: ('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-*G_T :: '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 $(g, g^{\alpha}, g^{\alpha^2}, \ldots, g^{\alpha^t})$ 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* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);
(*c*, *g*) ← $\mathcal{A}$ (*map* (λ*t'*. *exp* **g** ((*to-type* α)^*t'*)) [*0..<t+1*]);
return-spmf (*exp-*G_T (*e* **g g**) (*1*/((*to-type* α)+*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* {
$\alpha \leftarrow$ *sample-uniform* (*Coset.order G*);

```
    (c, g) ← 𝒜 (map (λt'. exp g ((to-type α)^t')) [0..<t+1]);
    -::unit ←assert-spmf (exp-G_T (e g g) (1/((to-type α)+c)) = g);
    return-spmf True
  } ELSE return-spmf False
  (is ?lhs = ?rhs)
proof −
  have ?lhs = TRY do {
    α ← sample-uniform (Coset.order G);
    TRY do {
      (c, g) ← 𝒜 (map (λt'. exp g ((to-type α)^t')) [0..<t+1]);
          TRY return-spmf (exp-G_T (e g g) (1/((to-type α)+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 {
    α ← sample-uniform (Coset.order G);
    TRY do {
      (c, g) ← 𝒜 (map (λt'. exp g ((to-type α)^t')) [0..<t+1]);
        TRY do {
          - :: unit ← assert-spmf (exp-G_T (e g g) (1/((to-type α)+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 commitment
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*: ⟦*lossless-spmf key-gen*;
⋀*ck p*. *valid-msg p* ⟹ *lossless-spmf* (*commit ck p*)⟧
⟹ *valid-msg p* ⟹ *lossless-spmf* (*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* ≡ (∀ *p i*. *valid-poly p* ⟶ *valid-argument i* ⟶ *spmf* (*correct-eval-game p i*) *True* = *1*)

We again reuse the previous work on commitment schemes

definition *poly-bind-game*
where *poly-bind-game* ≡ *cs.bind-game*

definition *poly-bind-advantage*
where *poly-bind-advantage* ≡ *cs.bind-advantage*

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

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* ⇒ *bool spmf*
where *eval-bind-game* $\mathcal{A}$ = *TRY do* {
(*ck*, *vk*) ← *key-gen*;
(*c*, *i*, *v*, *w*, *v'*, *w'*) ← $\mathcal{A}$ *ck*;
- :: *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*

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* ⇒ *real*
where *eval-bind-advantage* $\mathcal{A}$ ≡ *spmf* (*eval-bind-game* $\mathcal{A}$) *True*

type-synonym (*'vk'*, *'argument'*,*'state'*) *eval-hiding-adversary1* =
'vk' ⇒ (*'argument' list* × *'state'*) *spmf*

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

('vk' ⇒ 'state' ⇒ 'commit' ⇒ 'argument' list ⇒ ('evaluation' × 'witness') list ⇒
('r' poly) 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 *eval-hiding-game* :: 'r poly ⇒ ('vk, 'argument,'state) *eval-hiding-adversary1*
⇒
('r, 'vk, 'commit, 'argument, 'evaluation, 'witness, 'state) *eval-hiding-adversary2*
⇒ *bool spmf*
where *eval-hiding-game p* $\mathcal{A}1$ $\mathcal{A}2$ = *TRY do* {
(*ck*, *vk*) ← *key-gen*;
(*I*,σ) ← $\mathcal{A}1$ *vk*;
(*c*,*d*) ← *commit ck p*;
let W = *map* (λ*i*. *eval ck d p i*) *I*;
p' ← $\mathcal{A}2$ *vk* σ *c I W*;
return-spmf (*p* = *p'*)} *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 *eval-hiding-advantage* :: 'r poly ⇒ ('vk, 'argument,'state) *eval-hiding-adversary1*
⇒
('r, 'vk, 'commit, 'argument, 'evaluation, 'witness, 'state) *eval-hiding-adversary2*
⇒ *real*
where *eval-hiding-advantage p* $\mathcal{A}1$ $\mathcal{A}2$ ≡ *spmf* (*eval-hiding-game p* $\mathcal{A}1$ $\mathcal{A}2$)
True

type-synonym ('ck', 'commit', 'state') *knowledge-soundness-adversary1* = 'ck' ⇒
('commit' × 'state') *spmf*

type-synonym ('state', 'ck', 'argument', 'evaluation', 'witness') *knowledge-soundness-adversary2*

= 'ck' ⇒ 'state' ⇒ ('argument' × ('evaluation' × 'witness')) *spmf*

type-synonym ('r', 'commit', 'trapdoor') *extractor* = 'commit' ⇒ ('r' poly ×
'trapdoor') *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 *knowledge-soundness-game* :: ('ck, 'commit, 'state) *knowledge-soundness-adversary1*

⇒ ('state, 'ck, 'argument, 'evaluation, 'witness) *knowledge-soundness-adversary2*

⇒ ('r, 'commit, 'trapdoor) *extractor* ⇒ *bool spmf*

where *knowledge-soundness-game* $\mathcal{A}1$ $\mathcal{A}2$ *E* = *TRY do* {
(*ck*,*vk*) ← *key-gen*;
(*c*,σ) ← $\mathcal{A}1$ *ck*;
(*p*,*d*) ← *E c*;
(*i*, (*p-i*,π)) ← $\mathcal{A}2$ *ck* σ;
let w = (*p-i*, π);
let (*p-i*′,-) = *eval ck d p i*;
return-spmf (*verify-eval vk c i w* ∧ *p-i* ≠ *p-i*′ ∧ *valid-argument i* ∧ *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*) *knowledge-soundness-adversary1*

⇒ ('*state*, '*ck*, '*argument*, '*evaluation*, '*witness*) *knowledge-soundness-adversary2*

⇒ ('*r*, '*commit*, '*trapdoor*) *extractor* ⇒ *real*
where *knowledge-soundness-advantage* $\mathcal{A}1$ $\mathcal{A}2$ *E* ≡ *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_type*S, e.g. for int: intS

record ('a,'b) *Select* = *select*::'a ⇒'b *list* (‹¤1›)

We provide a few generalized type constructor implementations.

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

definition *listS*::
  ('a *list*, 'b) *Select*
  where *listS* ≡ (|*select* = (λ*xs*. *concat* (*map* (λ*x*. ¤*r x*) *xs*))|)

definition *treeS* ::
  ('a *tree*, 'b) *Select*
  where *treeS* ≡ (|*select* = (λ*x*. ¤*listS*(*inorder x*))|)

end

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

definition *multisetS* ::
  ('a *multiset*, 'b) *Select*
  where *multisetS* ≡ (|*select* = (λ*x*. ¤*listS r* (*sorted-list-of-multiset x*))|)

definition *fsetS* ::
  ('a *fset*, 'b) *Select*
  where *fsetS* ≡ (|*select* = (λ*x*. ¤*listS r* (*sorted-list-of-fset x*))|)

end

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

definition *prodS*::
  (('a × 'b), 'c) *Select* **where**
  prodS ≡ (|*select* = (λ(*a*,*b*). ¤*ra a* @ ¤*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* = (λ*fm*. ¤$_{listS}$ (*prodS ra rb*) (*sorted-list-of-fmap fm*))|)

end

definition *noSelect* :: (*'a*, *'b*) *Select*
  where *noSelect* ≡ (|*select* = (λ*x*. [])|)

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_type*C, 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* = (λ*ip op*. *list-all* (λ*x*. *ip* ⊳$_r$ *x*) *op*)|)

definition *treeC* ::
  (*'b*, *'c tree*) *Constrain*
  where *treeC* ≡ (|*constrain* = (λ*ip op*. *ip* ⊳$_{listC}$(*inorder op*))|)

end

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

definition *multisetC* ::

(*'b*, *'c multiset*) *Constrain*
where *multisetC* ≡ (|*constrain* = (λ*ip op*. *ip* ⊵$_{listC\ r}$ (*sorted-list-of-multiset op*))|)

definition *fsetC* ::
(*'b*, *'c fset*) *Constrain*
where *fsetC* ≡ (|*constrain* = (λ*ip op*. *ip* ⊵$_{listC\ r}$ (*sorted-list-of-fset op*))|)

end

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

definition *prodC*::
(*'b*, *'c* × *'d*) *Constrain* **where**
prodC ≡ (|*constrain* = (λ*ip* (*opa*,*opb*). *ip* ⊵$_{ra}$ *opa* ∧ *ip* ⊵$_{rb}$ *opb*)|)

end

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

definition *fmapC*::
(*'b*, (*'c*,*'d*) *fmap*) *Constrain* **where**
fmapC ≡ (|*constrain* = (λ*ip op*. *ip* ⊵$_{listC\ (prodC\ ra\ rb)}$ (*sorted-list-of-fmap op*))|)

end

definition *noConstrain* :: (*'b*,*'c*) *Constrain*
where *noConstrain* ≡ (|*constrain* = (λ*ip op*. *True*)|)

lemma *noConstrainTrue*[*simp*]: ⋀*ip op*. *ip* ⊵$_{noConstrain}$ *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* :: $('a \Rightarrow 'c\ spmf) \Rightarrow 'a \Rightarrow 'c\ spmf$ **where**
restrict $\mathcal{A}$ *arg* $\equiv$ *do* {
res $\leftarrow$ $\mathcal{A}$ *arg*;
-::*unit* $\leftarrow$ *assert-spmf* $((\natural_{sel}\ arg) \rhd_{con} res)$;
return-spmf res
}

end

8.4 Examples

locale *examples*
begin

examples for select

definition *intS* :: (*int*,*int*) *Select*
where *intS* $\equiv$ $(\!|select = (\lambda x.\ [x])|\!)$

lemma $\natural_{intS}\ (1::int) = [1::int]$
unfolding *intS-def*
by *simp*

lemma $\natural_{listS\ intS}\ [1] = [1]$
by(*simp add*: *listS-def intS-def*)

lemma $\natural_{listS\ (listS\ intS)}\ [[1]] = [1]$
unfolding *listS-def intS-def*
by *fastforce*

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

examples for constrain

definition *intC* :: (*int*, *int*) *Constrain*
where *intC* $\equiv$ $(\!|constrain = (\lambda ip\ op.\ sum\text{-}list\ ip = op)|\!)$

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

lemma $[0,1,2] \rhd_{listC\ intC} [3,3,3]$
unfolding *listC-def intC-def*
by *simp*

lemma $[0,1,2] \rhd_{prodC\ (listC\ intC)\ intC} ([3,3,3],3)$
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* ≡ (|*select* = (λ*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 vec=[c_0,...c_n] from the algorithm ensure that g=s_0 [^] c_0 ⊗ ... ⊗ s_n [^] c_n, where [s_0,...,s_n]=seen are the group values the algorithm was supplied with.

definition *constrain-grp* :: '*a list* ⇒ ('*a* × *int list*) ⇒ *bool*
where *constrain-grp seen-vec res* ≡
let (*g*,*c-vec*) = *res in*
(*length seen-vec* = *length c-vec*
∧ *g* = *fold* (λ *i acc*. *acc* ⊗$_G$ *seen-vec*!*i* [^]$_G$ (*c-vec*!*i*)) [*0*..<*length seen-vec*] **1**$_G$)

definition *groupC* :: (*'a*, (*'a* × *int list*)) *Constrain*
where *groupC* ≡ (|*constrain* = (λ*ip op*. *constrain-grp ip op*)|)

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

lemma *constrain-grp-is-in-carrier*:
assumes ∀ *g* ∈ *set seen-vec*. *g* ∈ *carrier G*
and *constrain-grp seen-vec* (*g*,*c-vec*)
shows *g* ∈ *carrier G*
proof −
have *g* = *fold* (λ *i acc*. *acc* ⊗ *seen-vec*!*i* [^] (*c-vec*!*i*)) [*0*..<*length seen-vec*] **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* (λ *i acc*. *acc* ⊗ *seen-vec*!*i* [^] (*c-vec*!*i*)) [*0*..<*length seen-vec*] **1** ∈ *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* (λ*i acc*. *acc* ⊗ *xs* ! *i* [^] *ys* ! *i*) [*0*..<*length xs*] **1** ∈ *carrier G*
by *fastforce*
moreover have *x* [^] *y* ∈ *carrier G*
by (*simp add*: *4*(*3*))
moreover have *fold* (λ*i acc*. *acc* ⊗ (*xs* @ [*x*]) ! *i* [^] (*ys* @ [*y*]) ! *i*) [*0*..<*length* (*xs* @ [*x*])] **1**
= (*fold* (λ*i acc*. *acc* ⊗ *xs* ! *i* [^] *ys* ! *i*) [*0*..<*length xs*] **1**) ⊗ *x* [^] *y*
proof −
let *?fyys* = λ*xs*::*'a list*. (λ*i acc*. *acc* ⊗ *xs* ! *i* [^] (*ys* @ [*y*]) ! *i*)
have *fold* (λ*i acc*. *acc* ⊗ (*xs* @ [*x*]) ! *i* [^] (*ys* @ [*y*]) ! *i*) [*0*..<*length* (*xs* @ [*x*])] **1**
= *fold* (*?fyys* (*xs*@[*x*])) [*0*..<*length* (*xs* @ [*x*])] **1** **by** *blast*
moreover have ... = (*fold* (*?fyys* (*xs*@[*x*])) [*length* (*xs* @ [*x*]) − *1*] ∘ *fold* (*?fyys* (*xs*@[*x*])) [*0*..<*length* (*xs*)]) **1**
by *auto*
moreover have ... = ((λ*acc*. *acc* ⊗ *x* [^] *y*) ∘ *fold* (*?fyys* (*xs*@[*x*])) [*0*..<*length* (*xs*)]) **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 ... = (*fold* (*?fyys* (*xs*@[*x*])) [*0*..<*length* (*xs*)] **1**) ⊗ *x* [^] *y*
by *force*
moreover have ... = (*fold* (λ*i acc*. *acc* ⊗ *xs* ! *i* [^] *ys* ! *i*) [*0*..<*length* (*xs*)] **1**) ⊗ *x* [^] *y*
proof −
have *fold* (*?fyys* (*xs*@[*x*])) [*0*..<*length* (*xs*)] **1**
= *fold* (λ*i acc*. *acc* ⊗ *xs* ! *i* [^] *ys* ! *i*) [*0*..<*length xs*] **1**
proof(*rule fold-cong*)

fix i
assume $i \in set\ [0..<length\ xs]$
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 $(\lambda acc.\ acc \otimes (xs\ @\ [x])\ !\ i\ [\frown]\ (ys\ @\ [y])\ !\ i) = (\lambda acc.\ acc \otimes xs\ !\ i\ [\frown]\ ys\ !\ i)$
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 $g \in carrier\ G$ **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* $\subseteq$ *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 $\mathcal{A}$-*id* :: $'a \Rightarrow ('a \times int\ list)\ spmf$
where $\mathcal{A}$-*id* = $(\lambda x.\ return\text{-}spmf\ (x,\ [1::int]))$

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

lemma

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

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

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

interpretation *list-id*: *Restrictive-Comp* (*listS groupS*) *listC groupC* **.**

lemma
assumes $g1 \in carrier\ G \wedge g2 \in carrier\ G \wedge g3 \in carrier\ G$
shows $list\text{-}id.restrict\ \mathcal{A}\text{-}list\text{-}fst\ [g1,g2,g3]$
$= \mathcal{A}\text{-}list\text{-}fst\ [g1,g2,g3]$
unfolding $\mathcal{A}\text{-}list\text{-}fst\text{-}def\ Restrictive\text{-}Comp.restrict\text{-}def$
unfolding $listS\text{-}def\ groupS\text{-}def\ listC\text{-}def\ groupC\text{-}def$
unfolding *constrain-grp-def*
by (*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 ‹
local
(∗*apply function f to the codaomain of a function*∗)
fun rcodomain-transf f (*Type* (*fun*, [*T*, *U*])) = (*Type* (*fun*, [*T*, *rcodomain-transf f U*]))
| *rcodomain-transf f T* = *f T*;

(∗*adjoin vec to t as a product type if the Type* = *t*∗)
fun adjoin t vec = *fn T* => *if T* = *t then Type* (*Product-Type.prod*, [*T*, *vec*]) *else T*

(∗*decurry a function*∗)
fun decurry (*Type* (*fun*, [*T1*, *Type* (*fun*, [*T2*, *T3*])])) =
decurry (*Type* (*fun*, [*Type* (*Product-Type.prod*, [*T1*, *T2*]), *T3*]))
| *decurry T* = *T*

fun extract-type-params t = *Term.add-tfree-namesT t* []

fun lift-to-algebraicT grpT vec = (*rcodomain-transf* (*Term.map-atyps* (*adjoin grpT vec*))) *o 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.fastype-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 $('a')adv = 'a\ list \Rightarrow 'a \Rightarrow nat \Rightarrow ('a * int * nat)\ spmf$

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

lift-to-algebraicT $('a)\ adv\ \ G$ => *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 $PK = (g,\ g\hat{}\alpha,\ \ldots\ ,\ g\hat{}(\alpha\hat{}t))$. Setup is a trusted Setup function, which generates the shared public key for both parties (prover and verifier).

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

This function computes $g\hat{}\varphi(\alpha)$, 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 ^ coeff φ i*, which computes $g\hat{}\varphi(a)$, given the public key: $\prod [0...degree\ \varphi].\ PK!i\hat{}coeff\ \varphi\ i = \prod [0...degree\ \varphi].\ g\hat{}(\alpha\hat{}i)\hat{}coeff\ \varphi\ i = \prod [0...degree\ \varphi].\ g\hat{}(coeff\ \varphi\ i * \alpha\hat{}i) = g\hat{}(\sum [0...degree\ \varphi].\ coeff\ \varphi\ i * \alpha\hat{}i) = g\hat{}\varphi(\alpha)$

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

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 mod-ring poly ⇒ 'e mod-ring ⇒ 'e mod-ring list ⇒ nat ⇒ 'e mod-ring list*

where *coeffs-n* φ *u* = (λ*q-coeffs*. λ*n*. *map* (λ(*i*::*nat*). (*nth-default 0 q-coeffs i* + *poly.coeff* φ *n* * *u* ^ (*n* − *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: $\varphi\ x - \varphi\ u = (\sum i{\leq}degree\ \varphi.\ coeff\ \varphi\ i * x\hat{}i) - (\sum i{\leq}degree\ \varphi.\ coeff\ \varphi\ i * x\hat{}i) = (x-u)(\sum i{\leq}degree\ \varphi.\ coeff\ \varphi\ i * (\sum j{<}i.\ u\hat{}(i - Suc\ j){*}x\hat{}j))$ (for the last step see the lemma power_diff_sumr2) Hence: $\psi\ x = (\sum i{\leq}degree\ \varphi.\ coeff\ \varphi\ i * (\sum j{<}i.\ u\hat{}(i - Suc\ j){*}x\hat{}j))$ 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 $(\sum i{\leq}degree\ \varphi.\ coeff\text{-}i{*}x\hat{}i)$ 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 degree φ = n, and adds (or could add) to each coefficient in every iteration: 0: 0 + 1: f1 + 2: f2*u + f2*x + 3: f3*u^2 + f3*u*x + f3*x^2 ... n: fn*u^(n-1) + ... fn*u^((n-1)-i)*x^i ... + fn*x^(n-1) we order it to compute the coefficients one by one (to compute vertical) 1: 0 + f1 + ... + fn*u^(n-1) + 2: 0 + f2 * x + ... + fn*u^((n-1)-1) * x + ... n: 0 + fn * x^(n-1)

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

definition ψ-*of* :: *'e mod-ring poly* ⇒ *'e mod-ring* ⇒ *'e mod-ring poly*
where ψ-*of* φ *u* = (*let*
ψ-*coeffs* = *foldl* (*coeffs-n* φ *u*) [] [*0*..<*Suc* (*degree* φ)] — coefficients of ψ
in Poly ψ-*coeffs*) — ψ

a wrapper around Setup that hands the srs to both parties

definition *key-gen* :: (*'a ck* × *'a vk*) *spmf*
where *key-gen* = *do* {
(α, *PK*) ← *Setup*;
return-spmf (*PK*,*PK*)
}

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 *commit* :: *'a ck* ⇒ *'e mod-ring poly* ⇒ (*'a commit* × *trapdoor*) *spmf*
where *commit PK* φ = *return-spmf* (*g-pow-PK-Prod PK* φ, ()) — $g\hat{}\varphi(\alpha)$

definition *verify-poly* :: *'a vk* ⇒ *'e mod-ring poly* ⇒ *'a commit* ⇒ *trapdoor* ⇒ *bool*
where *verify-poly PK* φ *C td* = (*C* = *g-pow-PK-Prod PK* φ) — $C = g\hat{}\varphi(\alpha)$

This is the createWitness function in the KZG paper

definition *Eval* :: *'a ck* ⇒ *trapdoor* ⇒ *'e mod-ring poly* ⇒ *'e mod-ring* ⇒ (*'e mod-ring* × *'a witness*)
where *Eval PK td φ i* = (*let ψ* = *ψ-of φ i* — *ψ* in *φ*(*x*) − *φ*(*i*) = (*x*−*i*) ∗ *ψ*(*x*)
in (*poly φ i*, *g-pow-PK-Prod PK ψ*) — (*φ*(*i*),*g*^*ψ*(*α*)))

definition *verify-eval* :: *'a vk* ⇒ *'a commit* ⇒ *'e mod-ring* ⇒ (*'e mod-ring* × *'a witness*) ⇒ *bool*
where *verify-eval PK C i val* = (*let* (*eval*,*w*) = *val*
in (*e w* (*PK*!*1* ÷$_{G_p}$ (**g**$_{G_p}$ ^$_{G_p}$ *i*)) ⊗$_{G_T}$ ((*e* **g**$_{G_p}$ **g**$_{G_p}$) ^$_{G_T}$ *eval*) = *e C* **g**$_{G_p}$))
— *e*(*g*^*ψ*(*α*), *g*^*α* / *g*^*i*) ⊗ *e*(*g*,*g*)^*φ*(*i*) = *e*(*C*, *g*)

definition *valid-poly*::*'e mod-ring poly* ⇒ *bool*
where *valid-poly φ* = (*degree φ* ≤ *max-deg*)

definition *valid-argument* :: *'e argument* ⇒ *bool*
where *valid-argument* - = *True*

definition *valid-eval*::(*'e mod-ring* × *'a witness*) ⇒ *bool*
where *valid-eval val* = (*let* (-,*w*) = *val in w* ∈ *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 *ψ-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*]: ∀ *i*<*n*. *coeffs-n φ u l n* ! *i* = *nth-default 0 l i* + *poly.coeff φ n* ∗ *u* ^ (*n* − *Suc i*)
unfolding *coeffs-n-def* **by** *auto*

lemma *ψ-coeffs-length*: *length* (*foldl* (*coeffs-n φ u*) [] [*0..<Suc n*]) = *n*
by *auto*

lemma *sum-split*: *m≤n* ⟹ (∑ *i≤n*. *f i*) = (∑ *i≤m*. *f i*) + (∑ *i∈*{*m<..<Suc n*}. *f i*)
proof −
assume *m≤n*
then show (∑ *i≤n*. *f i*) = (∑ *i≤m*. *f i*) + (∑ *i∈*{*m<..<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*: *degree* (*ψ-of φ u*) ≤ *degree φ*
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 co computes the coefficients horizontal, in the sense that it computes the coefficients from 0 to degree φ = n, and adds (or could add) to each coefficient in every iteration: 0: 0 + 1: f1 + 2: f2*u + f2*x + 3: f3*u^2 + f3*u*x + f3*x^2 ... n: fn*u^(n-1) + ... fn*u^((n-1)-i)*x^i ... + fn*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 + f1 + f2*u + f3*u^2 + ... + fn*u^(n-1))*x^0 + 1: (0 + 0 + f2 + f3*u + ... + fn*u^(n-2))*x^1 ... n: (0 + 0 + 0 + 0 + ... + fn)*x^(n-1)

lemma *sum-horiz-to-vert*: *n≤degree* (*φ*::*'e mod-ring poly*) ⟹
(∑ *i≤n*. *poly.coeff φ i* ∗ (∑ *j<i*. *u* ^ (*i* − *Suc j*) ∗ *x* ^ *j*))
= (∑ *i≤n*. (∑ *j∈*{*i<..<Suc n*}. *poly.coeff φ j* ∗ *u* ^ (*j* − *Suc i*)) ∗ *x*^*i*)
proof (*induction n arbitrary*: *φ*)
case *0*
have (∑ *i≤0*. *poly.coeff φ i* ∗ (∑ *j<i*. *u* ^ (*i* − *Suc j*) ∗ *x* ^ *j*)) = *0* **by** *fastforce*
also have (∑ *i≤0*. (∑ *j∈*{*i<..<Suc 0*}. *poly.coeff φ j* ∗ *u* ^ (*j* − *Suc i*)) ∗ *x* ^ *i*) = *0*

by (*simp add*: *greaterThanLessThan-upt*)
ultimately show *?case* **by** *argo*
next
case (*Suc n*)
have ($\sum i{\leq}$*Suc n*. *poly.coeff* φ *i* $*$ ($\sum j{<}i$. *u* $\hat{}$ (*i* $-$ *Suc j*) $*$ *x* $\hat{}$ *j*))
$=$ ($\sum i{\leq}n$. *poly.coeff* φ *i* $*$ ($\sum j{<}i$. *u* $\hat{}$ (*i* $-$ *Suc j*) $*$ *x* $\hat{}$ *j*))
$+$ *poly.coeff* φ (*Suc n*) $*$ ($\sum j{<}$(*Suc n*). *u* $\hat{}$ (*Suc n* $-$ *Suc j*) $*$ *x* $\hat{}$ *j*) **by** *auto*
also have ... $=$ ($\sum i{\leq}n$. ($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $*$ *x* $\hat{}$ *i*)
$+$ *poly.coeff* φ (*Suc n*) $*$ ($\sum j{<}$(*Suc n*). *u* $\hat{}$ (*Suc n* $-$ *Suc j*) $*$ *x* $\hat{}$ *j*)
using *Suc.IH Suc.prems Suc-leD* **by** *presburger*
also have ...$=$($\sum i{\leq}n$. ($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $*$ *x* $\hat{}$ *i*)
$+$ ($\sum j{<}$(*Suc n*). *poly.coeff* φ (*Suc n*) $*$ *u* $\hat{}$ (*Suc n* $-$ *Suc j*) $*$ *x* $\hat{}$ *j*)
using *lessThan-Suc-atMost* **by** *presburger*
also have ...$=$($\sum i{<}$*Suc n*. ($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $*$ *x* $\hat{}$ *i*
$+$ *poly.coeff* φ (*Suc n*) $*$ *u* $\hat{}$ (*Suc n* $-$ *Suc i*) $*$ *x* $\hat{}$ *i*)
by (*simp add*: *sum.distrib*)
also have ...$=$($\sum i{<}$*Suc n*. (($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $+$ *poly.coeff* φ (*Suc n*) $*$ *u* $\hat{}$ (*Suc n* $-$ *Suc i*))$*$ *x* $\hat{}$ *i*)
by (*simp add*: *distrib-left mult.commute*)
also have ...$=$($\sum i{<}$*Suc n*. ($\sum j{\in}\{i{<}..{<}$*Suc* (*Suc n*)$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $*$ *x* $\hat{}$ *i*)
proof $-$
have $\forall$ (*i*::*nat*)$<$*Suc n*. (($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $+$ *poly.coeff* φ (*Suc n*) $*$ *u* $\hat{}$ (*Suc n* $-$ *Suc i*))
$=$ ($\sum j{\in}\{i{<}..{<}$*Suc* (*Suc n*)$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*))
proof
fix *i*
show *i* $<$ *Suc n* $\longrightarrow$
($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $+$ *poly.coeff* φ (*Suc n*) $*$ *u* $\hat{}$ (*Suc n* $-$ *Suc i*) $=$
($\sum j{\in}\{i{<}..{<}$*Suc* (*Suc n*)$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*))
proof
let *?f* $=$ (λ*j*. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*))
assume *asmp*: *i* $<$ *Suc n*
then show ($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *?f j*) $+$ *?f* (*Suc n*) $=$ ($\sum j{\in}\{i{<}..{<}$*Suc* (*Suc n*)$\}$. *?f j*)
by (*smt* (*verit*) *atLeastSucLessThan-greaterThanLessThan not-less-eq sum.op-ivl-Suc*)
qed
qed
then show ($\sum i{<}$*Suc n*. (($\sum j{\in}\{i{<}..{<}$*Suc n*$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $+$ *poly.coeff* φ (*Suc n*) $*$ *u* $\hat{}$ (*Suc n* $-$ *Suc i*)) $*$ *x* $\hat{}$ *i*) $=$
($\sum i{<}$*Suc n*. ($\sum j{\in}\{i{<}..{<}$*Suc* (*Suc n*)$\}$. *poly.coeff* φ *j* $*$ *u* $\hat{}$ (*j* $-$ *Suc i*)) $*$ *x* $\hat{}$

```
i)
      by fastforce
  qed
  also have ...=(∑ i≤Suc n. (∑ j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j −
Suc i)) * x ^ i)
  proof −
    have (∑ j∈{Suc n<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j − Suc (Suc n)))
* x ^ (Suc n) = 0
      by (simp add: greaterThanLessThan-upt)
    then have (∑ i<Suc n. (∑ j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j −
Suc i)) * x ^ i)
              = (∑ i<Suc n. (∑ j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j −
Suc i)) * x ^ i)
              + (∑ j∈{Suc n<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j − Suc (Suc
n))) * x ^ (Suc n)
      by force
    also have ...=(∑ i≤Suc n. (∑ j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j
− Suc i)) * x ^ i)
      by (simp add: lessThan-Suc-atMost)
    ultimately show (∑ i<Suc n. (∑ j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^
(j − Suc i)) * x ^ i)
          = (∑ i≤Suc n. (∑ j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j − Suc
i)) * x ^ i)
      by argo
  qed
  ultimately show ?case using Suc.prems 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 ⟹ foldl (coeffs-n φ u) [] [0..<Suc n]
! i
                                    = (∑ j∈{i<..<Suc n}. poly.coeff φ j * u ^ (j −
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 foldl (coeffs-n φ u) [] [0..<Suc (Suc n)]
      = map (λi. nth-default 0 (foldl (coeffs-n φ u) [] [0..<Suc n]) i + poly.coeff φ
(Suc n) * u ^ (Suc n − Suc i))
     [0..<Suc n]
      unfolding coeffs-n-def by force
    then have foldl (coeffs-n φ u) [] [0..<Suc (Suc n)] ! i
```

```
      = nth-default 0 (foldl (coeffs-n φ u) [] [0..<Suc n]) i + poly.coeff φ (Suc
n) * u ^ (Suc n - Suc i)
     by (metis (lifting) Suc.prems add-0 diff-zero nth-map-upt)
   also have ...=(∑j∈{i<..<Suc n}. poly.coeff φ j * u ^ (j - Suc i)) + poly.coeff
φ (Suc n) * u ^ (Suc n - Suc i)
     using Suc.IH[of i] by (simp add: True nth-default-def)
   also have ...=(∑j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j - Suc i))
   proof -
     have ∀ x y f. x<y ⟶ (∑j∈{x<..<y}. f j) + f y = (∑j∈{x<..<Suc y}. f j)
     by (metis Suc-le-eq atLeastSucLessThan-greaterThanLessThan sum.atLeastLessThan-Suc)
     then show (∑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)) using Suc.prems
by blast
   qed
   ultimately show ?thesis by argo
 next
   case False
   then have i-eq-n: i=n using Suc.prems by simp
   have foldl (coeffs-n φ u) [] [0..<Suc (Suc n)]
      = coeffs-n φ u (foldl (coeffs-n φ u) [] [0..<Suc n]) (Suc n) by simp
   also have ...=map (λ(i::nat). (nth-default 0 (foldl (coeffs-n φ u) [] [0..<Suc
n]) i
                    + poly.coeff φ (Suc n) * u ^ ((Suc n) - Suc i))) [0..<Suc n]
     unfolding coeffs-n-def by blast
   ultimately have foldl (coeffs-n φ u) [] [0..<Suc (Suc n)] ! i
                 = map (λ(i::nat). (nth-default 0 (foldl (coeffs-n φ u) [] [0..<Suc
n]) i
                    + poly.coeff φ (Suc n) * u ^ ((Suc n) - Suc i))) [0..<Suc n] ! i
     by argo
   also have ...= (nth-default 0 (foldl (coeffs-n φ u) [] [0..<Suc n]) i
                    + poly.coeff φ (Suc n) * u ^ ((Suc n) - Suc i))
     using Suc.prems calculation by auto
   also have ...=poly.coeff φ (Suc n) * u ^ ((Suc n) - Suc i)
     by (simp add: nth-default-eq-dflt-iff i-eq-n)
   also have (∑j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j - Suc i))
         = poly.coeff φ (Suc n) * u ^ ((Suc n) - Suc i)
   proof -
     have {i<..<Suc (Suc n)} = {Suc n}
     proof
       show {i<..<Suc (Suc n)} ⊆ {Suc n}
         by (simp add: greaterThanLessThan-upt i-eq-n)
       show {Suc n} ⊆ {i<..<Suc (Suc n)}
         by (simp add: i-eq-n)
     qed
     then show (∑j∈{i<..<Suc (Suc n)}. poly.coeff φ j * u ^ (j - Suc i))
         = poly.coeff φ (Suc n) * u ^ ((Suc n) - Suc i)
       by simp
   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 *poly* (ψ-*of-poly* φ *u*) *x* equal to the result sum of sum_horiz_to_vert.

lemma φ*x-m-*φ*u-eq-xmu-*ψ*x*: $\forall \varphi$::*'e mod-ring poly. poly* φ *x* − *poly* φ *u* = (*x*−*u*) ∗ *poly* (ψ-*of* φ *u*) *x*
proof
fix φ::*'e mod-ring poly*
fix *u x* :: *'e mod-ring*
show *poly* φ *x* − *poly* φ *u* = (*x*−*u*) ∗ *poly* (ψ-*of* φ *u*) *x*
proof −
let *?q-coeffs* = *foldl* (*coeffs-n* φ *u*) [] [*0..<Suc* (*degree* φ)]
let *?q-dirty* =(λ*x*. (∑ *i*≤*degree* φ. *poly.coeff* φ *i* ∗ (∑*j*<*i*. *u*^(*i* − *Suc j*) ∗ *x*^*j*)))
let *?q-vert* =(λ*x*. (∑ *i*≤*degree* φ. (∑ *j*∈{*i*<..<*Suc* (*degree* φ)}. *poly.coeff* φ *j* ∗ *u* ^ (*j* − *Suc i*)) ∗ *x*^*i*))
let *?q* = ψ-*of* φ *u*

have (∑ *i*≤*degree* φ. *poly.coeff* φ *i* ∗ *x* ^ *i*) − (∑ *i*≤*degree* φ. *poly.coeff* φ *i* ∗ *u* ^ *i*)
= (∑ *i*≤*degree* φ. *poly.coeff* φ *i* ∗ (*x* ^ *i* − *u* ^ *i*))
by (*simp add*: *sum-subtractf right-diff-distrib'*)
also have ... = (∑ *i*≤*degree* φ. (*x* − *u*) ∗ *poly.coeff* φ *i* ∗ (∑*j*<*i*. *u*^(*i* − *Suc j*) ∗ *x*^*j*))
by (*simp add*: *mult.assoc mult.left-commute power-diff-sumr2*)
also have ... = (*x* − *u*) ∗ (*?q-dirty x*)
by (*metis* (*mono-tags*, *lifting*) *mult.assoc mult-hom.hom-sum sum.cong*)
also have ...= (*x*−*u*) ∗ (*?q-vert x*) **using** *sum-horiz-to-vert* **by** *auto*
also have *?q-vert x* = *poly ?q x*
proof −

have (∑ *i*≤*degree* φ. *nth-default 0 ?q-coeffs i* ∗ *x*^*i*)
= (∑ *i*≤*degree ?q*. *nth-default 0 ?q-coeffs i* ∗ *x*^*i*)
proof −
have *degree ?q* ≤ *degree* φ **by**(*rule degree-q-le-*φ)
also have ∀ *n*. *n*≥*degree ?q* ∧ *n*≤*degree* φ ⟶ (∑ *i*≤*n*. *nth-default 0 ?q-coeffs i* ∗ *x*^*i*)
= (∑ *i*≤*degree ?q*. *nth-default 0 ?q-coeffs i* ∗ *x*^*i*)
proof
fix *n*

```
      show n≥degree ?q ∧ n≤degree φ ⟶ (∑ i≤n. nth-default 0 ?q-coeffs i *
x^i)
                                    = (∑ i≤degree ?q. nth-default 0 ?q-coeffs i *
x^i)
      proof
        let ?f = nth-default 0 ?q-coeffs
        assume asmp: n≥degree ?q ∧ n≤degree φ
        have ∀ i>degree ?q. ?f i = 0
          using coeff-eq-0 coeffs-n-def
          by (metis ψ-of-def coeff-Poly-eq)
        then have (∑ i∈{degree ?q <..<Suc n}. ?f i * x^i) = 0
          by fastforce
          also have (∑ i≤n. ?f i * x ^ i) = (∑ i≤degree ?q. ?f i * x ^ i) +
(∑ i∈{degree ?q <..<Suc n}. ?f i * x^i)
          using sum-split asmp by blast
        ultimately show (∑ i≤n. nth-default 0 ?q-coeffs i * x ^ i)
              = (∑ i≤degree ?q. nth-default 0 ?q-coeffs i * x ^ i)
          using asmp by auto
      qed
    qed
    ultimately show (∑ i≤degree φ . nth-default 0 ?q-coeffs i * x^i)
          = (∑ i≤degree ?q. nth-default 0 ?q-coeffs i * x^i)
      by blast
  qed
  also have ?q-vert x = (∑ i≤degree φ. nth-default 0 ?q-coeffs i * x^i)
  proof −
    have ∀ i≤degree φ. (∑ j∈{i<..<Suc (degree φ)}. poly.coeff φ j * u ^ (j −
Suc i))
                    = nth-default 0 ?q-coeffs i
    proof
      fix i
      show i ≤ degree φ ⟶
       (∑ j∈{i<..<Suc (degree φ)}. poly.coeff φ j * u ^ (j − Suc i)) =
       nth-default 0 ?q-coeffs i
      proof
        assume asmp: i ≤ degree φ
        then show (∑ j∈{i<..<Suc (degree φ)}. poly.coeff φ j * u ^ (j − Suc
i)) =
          nth-default 0 ?q-coeffs i
        proof (cases i<degree φ)
          case True
          have length ?q-coeffs = degree φ by simp
          then have nth-default 0 ?q-coeffs i
              = ?q-coeffs ! i
            by (simp add: True nth-default-nth)
              then show ?thesis using True ψ-of-ith-coeff-eq-sum-ith-coeff by
presburger
        next
          case False
```

```
            then have i=degree φ using asmp by fastforce
            have length ?q-coeffs = degree φ by simp
            then have nth-default 0 ?q-coeffs i = 0
              by (simp add: ‹i = degree φ› nth-default-beyond)
            also have (∑ j∈{i<..<Suc (degree φ)}. poly.coeff φ j * u ^ (j − Suc
i))
                    = 0  using False greaterThanLessThan-upt by auto
            ultimately show ?thesis by argo
          qed
        qed
      qed
      then show ?q-vert x = (∑ i≤degree φ. nth-default 0 ?q-coeffs i * x^i)
        by force
    qed
    ultimately show ?q-vert x = poly ?q x
      by (metis (no-types, lifting) ψ-of-def coeff-Poly-eq poly-altdef sum.cong)
  qed
  ultimately show poly φ x − poly φ u = (x−u) * poly (ψ-of φ 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 billinear function e.

```
lemma eq-on-e: (e (g_{G_p} ^_{G_p} (poly (ψ-of φ i) α))  (g_{G_p} ^_{G_p} α ÷_{G_p} g_{G_p} ^_{G_p}
i))
    ⊗_{G_T} (e g_{G_p} g_{G_p})^_{G_T} (poly φ i)
    = e (g_{G_p} ^_{G_p} (poly φ α)) g_{G_p}
proof −
  have e-in-carrier: (e g_{G_p} g_{G_p} ) ∈ carrier G_T using e-symmetric by blast
  have e (g_{G_p} ^_{G_p} poly (ψ-of φ i) α) (g_{G_p} ^_{G_p} α ÷_{G_p} g_{G_p} ^_{G_p} i) ⊗_{G_T} e g_{G_p}
g_{G_p} ^_{G_T} poly φ i
    = e (g_{G_p} ^_{G_p} poly (ψ-of φ i) α) (g_{G_p} ^_{G_p} (α − i)) ⊗_{G_T} e g_{G_p} g_{G_p} ^_{G_T}
poly φ i
    using mod-ring-pow-mult-inv-G_p by force
  also have ...= (e g_{G_p} g_{G_p} ) ^_{G_T} ((poly (ψ-of φ i) α) * (α−i))  ⊗_{G_T} e g_{G_p}
g_{G_p} ^_{G_T} poly φ i
    using G_p.generator-closed e-bilinear by presburger
  also have ...= (e g_{G_p} g_{G_p} ) ^_{G_T} ((poly (ψ-of φ i) α) * (α−i) + poly φ i)
    using mod-ring-pow-mult-G_T e-in-carrier by presburger
  also have ...= (e g_{G_p} g_{G_p} ) ^_{G_T} (poly φ α)
    by (metis Groups.mult-ac(2) φx-m-φu-eq-xmu-ψx diff-add-cancel)
  also have ...= e (g_{G_p} ^_{G_p} (poly φ α)) g_{G_p}
    by (simp add: e-linear-in-fst)
  finally show e (g_{G_p} ^_{G_p} poly (ψ-of φ i) α) (g_{G_p} ^_{G_p} α ÷_{G_p} g_{G_p} ^_{G_p} i) ⊗_{G_T}
e g_{G_p} g_{G_p} ^_{G_T} poly φ i =
    e (g_{G_p} ^_{G_p} poly φ α) 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 ith public parameter is actually the ith public parameter ($g\hat{}(\alpha\hat{}i)$)

lemma *PK-i*: $i \leq t \Longrightarrow$ *map* ($\lambda t.$ **g** $\hat{}_{G_p}$ α $\hat{}$ t) [*0..<t + 1*] ! $i =$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ($\alpha\hat{}i$)
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 ($\prod$PK. ϕ) = g * g$\hat{}$($\alpha\hat{}$ 1st coefficient of ϕ) * g$\hat{}$(($\alpha\hat{}$2) $\hat{}$ 2nd coefficient of ϕ) * ... * g$\hat{}$(($\alpha\hat{}$t) $\hat{}$ t-th coefficient of ϕ) Which is the first prestep to showing ($\prod$PK. ϕ) = g$\hat{}\phi(\alpha)$.

lemma *g-pow-PK-Prod-to-fold*[*simp*]: *degree* $\varphi \leq t \Longrightarrow$ *g-pow-PK-Prod* (*map* ($\lambda t.$ **g** $\hat{}_{G_p}$ α $\hat{}$ t) [*0..<t + 1*]) φ
= *fold* ($\lambda pk\ g.\ g \otimes_{G_p}$ ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ($\alpha\hat{}pk$)) $\hat{}_{G_p}$ (*poly.coeff* φ *pk*)) [*0..<Suc* (*degree* φ)] $\mathbf{1}_{G_p}$
proof −
let *?PK* = *map* ($\lambda t.$ **g** $\hat{}_{G_p}$ α $\hat{}$ t) [*0..<t + 1*]
let *?g-pow-PK* = *g-pow-PK-Prod ?PK* φ
assume *asmpt*: *degree* $\varphi \leq t$
have *?g-pow-PK* = *fold* ($\lambda pk\ g.\ g \otimes_{G_p}$ *?PK*!*pk* $\hat{}_{G_p}$ (*poly.coeff* φ *pk*)) [*0..<Suc* (*degree* φ)] $\mathbf{1}_{G_p}$
by *auto*
also have *fold* ($\lambda pk\ g.\ g \otimes_{G_p}$ (*?PK*)!*pk* $\hat{}_{G_p}$ (*poly.coeff* φ *pk*)) [*0..<Suc* (*degree* φ)] $\mathbf{1}_{G_p}$
= *fold* ($\lambda pk\ g.\ g \otimes_{G_p}$ ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ($\alpha\hat{}pk$)) $\hat{}_{G_p}$ (*poly.coeff* φ *pk*)) [*0..<Suc* (*degree* φ)] $\mathbf{1}_{G_p}$
proof(*rule List.fold-cong*)

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

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

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

$n]$ $\mathbf{1}_{G_p}$
$\otimes_{G_p}$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ (*poly.coeff* φ (*Suc* n) $*$ α $\hat{}$ (*Suc* n))
using *Suc* **by** *auto*
also have ...=*fold* (λn *g*. g $\otimes_{G_p}$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ (*poly.coeff* φ n $*$ α $\hat{}$ 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 g^$\phi(\alpha)$ from the public parameters, ($\prod$PK. ϕ) = g^$\phi(\alpha)$. With lemma g_pow_PK_Prod_to_fold, we form the g_pow_PK_Prod part, which represents ($\prod$PK. ϕ), into g * g^(α ^ 1st coefficient of ϕ) * g^((α^2) ^ 2nd coefficient of ϕ) * ... * g^((α^t) ^ t-th coefficient of ϕ). Which we further form into $g\hat{}(\sum i{\leq}n.$ *coeff* φ i $*$ $\alpha\hat{}i)$, which is nothing else then g^$\phi(\alpha)$ (poly_altdef), with the poly_altdef_to_fold lemma

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

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

theorem *KZG-correct*: *correct-eval*
proof −

```
have ⋀φ i. valid-poly φ ⟹ spmf (correct-eval-game φ i) True = 1
proof −
  fix φ i
  assume assms: valid-poly φ
  show spmf (correct-eval-game φ i) True = 1
  proof −
    let ?α = λx. of-int-mod-ring (int x)
    let ?PK = λx. (map (λt. g ^_{G_p} ?α x ^ t) [0..<max-deg+1])

    have correct-eval-game φ i = do {
    (ck, vk) ← key-gen;
    (c,d) ← commit ck φ;
    let w  = Eval ck d φ i;
    return-spmf (verify-eval vk c i w)
    }
      unfolding correct-eval-game-def ..
    also have ... = do {
       x::nat ← sample-uniform (order G_p);
       return-spmf
            (e (g-pow-PK-Prod (?PK x) (ψ-of φ i))((?PK x)!1 ÷_{G_p} (g_{G_p} ^_{G_p}
i))
             ⊗_{G_T} e g_{G_p} g_{G_p} ^_{G_T} poly φ i
                = e (g-pow-PK-Prod (?PK x) φ) 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 ← sample-uniform (order G_p);
       return-spmf
            (e (g_{G_p} ^_{G_p} (poly (ψ-of φ i) (?α x))) (( g_{G_p} ^_{G_p} (?α x))  ÷_{G_p}
(g_{G_p} ^_{G_p} i)) ⊗_{G_T} ((e g_{G_p} g_{G_p}) ^_{G_T} (poly φ i ))
                = e (g_{G_p} ^_{G_p} (poly φ (?α x))) g_{G_p})}
    proof −
      let ?g-pow-φ = λx. g-pow-PK-Prod (?PK x) φ
      let ?g-pow-ψ = λx. g-pow-PK-Prod (?PK x) (ψ-of φ i)
      let ?g-pow-α = λx. (?PK x)!1
      have g-pow-φ: ?g-pow-φ = (λx. g_{G_p} ^_{G_p} (poly φ (?α x)))
      using g-pow-PK-Prod-correct assms unfolding valid-poly-def by presburger
      have degree-ψ: degree (ψ-of φ i) ≤ max-deg
        using assms degree-q-le-φ le-trans unfolding valid-poly-def by fast
      have g-pow-ψ: ?g-pow-ψ = (λx. g_{G_p} ^_{G_p} (poly (ψ-of φ i) (?α x)))
        using g-pow-PK-Prod-correct[OF degree-ψ] by presburger
      have g-pow-α: ?g-pow-α = (λx. g_{G_p} ^_{G_p} (?α 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 ← sample-uniform (order G_p);
       return-spmf True}
      using eq-on-e by presburger
```

also have ... = *scale-spmf* (*weight-spmf* (*sample-uniform* (*order* G_p))) (*return-spmf True*)
using *bind-spmf-const* **by** *metis*
finally show *?thesis* **by** (*simp add*: G_p.*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*: *ennreal* (*spmf game1 True*) ≤ *ennreal* (*spmf game2 True*) ⟹
spmf game1 True ≤ *spmf game2 True*
by *simp*

lemma *unpack-bind-spmf*: $X = X' \Longrightarrow$ *bind-spmf X Y* = *bind-spmf X' Y*
by *simp*

lemma *unpack-bind-spmf'*: $Y = Y' \Longrightarrow$ *bind-spmf X Y* = *bind-spmf X Y'*
by *simp*

lemma *unpack-bind-spmf-fun*: $X = X' \Longrightarrow$ *bind-spmf X* (λ*y*. *f y*) = *bind-spmf X'* (λ*y*. *f y*)
by (*fact CryptHOL-ext.unpack-bind-spmf*)

lemma *unpack-try-spmf*: $X = X' \Longrightarrow$ *TRY X ELSE Y* = *TRY X' ELSE Y*
by *simp*

lemma *unpack-try-spmf'*: $Y = Y' \Longrightarrow$ *TRY X ELSE Y* = *TRY X ELSE Y'*
by *simp*

lemma *spmf-eqI'*: $X = X' \Longrightarrow$ *spmf X Y* = *spmf X' Y*
by *simp*

12.1 SPMF True

lemma *return-spmf-assert*: *TRY return-spmf X ELSE return-spmf False* =
TRY bind-spmf (*assert-spmf X*) (λ-. *return-spmf True*) *ELSE return-spmf False*
by (*simp add*: *try-bind-assert-spmf*)

lemma *bind-spmf-independent-return-spmf*:
lossless-spmf x $\Longrightarrow$ *bind-spmf x* (λ*x*. *return-spmf y*) = *return-spmf y*
by (*simp add*: *bind-eq-return-spmf*)

lemma *bind-spmf-le*:
(⋀*x*. *spmf* (*f x*) *True* ≤ *spmf* (*f' x*) *True*) $\Longrightarrow$ *spmf* (*bind-spmf p f*) *True* ≤ *spmf* (*bind-spmf p f'*) *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 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 nn-integral-cong pmf-nonneg real-norm-def*)
apply *simp*
done

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

lemma *try-spmf-le*:
assumes *spmf x True* ≤ *spmf x' True*
shows *spmf* (*TRY x ELSE return-spmf False*) *True* ≤ *spmf* (*TRY x' ELSE return-spmf False*) *True*
apply (*rule ennreal-spmf*)
apply (*simp add*: *spmf-try-spmf*)
apply (*rule assms*)
done

lemma *try-spmf-true-else-false-le*: *spmf* (*TRY X ELSE return-spmf False*) *True* ≤ *spmf X True*
by (*simp add*: *spmf-try-spmf*)

lemma *del-assert*: *spmf* (*bind-spmf* (*assert-spmf X*) (λ- . *Y*)) *True* ≤ *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* ⟶ *X'*) ⟹
spmf (*bind-spmf* (*assert-spmf X*) (λ- . *Y*)) *True*
≤ *spmf* (*bind-spmf* (*assert-spmf X'*) (λ- . *Y*)) *True*
using *del-assert* **by** *fastforce*

lemma *assert-ret-unit*: *bind-spmf* (*assert-spmf x*) (λ*x* . *y*) = *bind-spmf* (*assert-spmf x*) (λ- . *y*)
by *presburger*

lemma *assert-based-eq*:
assumes *x* ⟹ *y* = *y'*
shows *bind-spmf* (*assert-spmf x*) (λ- . *y*)
= *bind-spmf* (*assert-spmf x*) (λ- . *y'*)
by (*auto simp add*: *assms assert-spmf-def*)

lemma *assert-commute*: *bind-spmf X* (λ*x*. *bind-spmf* (*assert-spmf Y*) (λ-. *Z x*))
= *bind-spmf* (*assert-spmf Y*) (λ-. *bind-spmf X* (λ*x*. *Z x*))
by (*rule bind-commute-spmf*)

lemma *assert-collapse*: *bind-spmf* (*assert-spmf X*) (λ-. *bind-spmf* (*assert-spmf Y*) (λ-. *Z*)) =
bind-spmf (*assert-spmf* (*X* ∧ *Y*)) (λ-. *Z*)
by (*smt* (*verit*) *assert-spmf-simps*(*1*,*2*) *return-None-bind-spmf return-bind-spmf*)

lemma *assert-cong*: *X* = *Y* ⟹ *rel-spmf* (=) (*assert-spmf X*) (*assert-spmf Y*)
by (*simp add*: *spmf-rel-eq*)

lemma *rel-spmf-bind-assert-reflI*: (*Z* ⟹ *rel-spmf P X Y*) ⟹
rel-spmf P (*bind-spmf* (*assert-spmf Z*) (λ-. *X*)) (*bind-spmf* (*assert-spmf Z*) (λ-. *Y*))
using *rel-spmf-bind-reflI* **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* :: *nat* ⇒ *nat* ⇒ *nat set spmf*
where *sample-uniform-set k n* = *spmf-of-set* {*x*. *x* ⊆ {..<*n*} ∧ *card x* = *k*}

lemma *spmf-sample-uniform-set*: *spmf* (*sample-uniform-set k n*) *x*
= *indicator* {*x*. *x* ⊆ {..<*n*} ∧ *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*≤*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*≤*n* ⟹ *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*) ⟷ *k* ≤ *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* ⊆ {..<*n*} ∧ *card x* = *k*}
by(*simp add*: *sample-uniform-set-def*)

12.3 Sample uniform list

definition *sample-uniform-list* :: *nat* ⇒ *nat* ⇒ *nat list spmf*
where *sample-uniform-list k n* = *spmf-of-set* {*x*. *set x* ⊆ {..<*n*} ∧ *length x* = *k*}

lemma *spmf-sample-uniform-list*: *spmf* (*sample-uniform-list k n*) *x*
= *indicator* {*x*. *set x* ⊆ {..<*n*} ∧ *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* ∨ *0*<*n*)
proof −
have *0* < *n* ⟶ (∃ *x*. *set x* ⊆ {..<*n*} ∧ *length x* = *k*)
proof
assume *zero-l-n*: *0* < *n*
show ∃ *x*. *set x* ⊆ {..<*n*} ∧ *length x* = *k*
proof

```
    let ?x = replicate k 0
    show set ?x ⊆ {..<n} ∧ 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 ⟹ 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)
⟷ (k=0 ∨ 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 ⊆ {..<n} ∧ 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) ⊆ {..<p} ∧ length xs = k+1 ∧ hd xs = x}
proof −
 fix x
 have set-spmf (map-spmf ((#) x) (sample-uniform-list k p))
   = ((#) x) ‘ {xs. set xs ⊆ {..<p} ∧ length xs = k}
   unfolding sample-uniform-list-def
   by (simp add: finite-lists-length-eq)
 also have ... = {xs. set (tl xs) ⊆ {..<p} ∧ length xs = k+1 ∧ hd xs = x}
 proof (rule equalityI; rule subsetI)
   fix ys assume ys ∈ (#) x ‘ {xs. set xs ⊆ {..<p} ∧ length xs = k}
   then obtain zs where zs ∈ {xs. set xs ⊆ {..<p} ∧ length xs = k} and ys =
x # zs
     by blast
   then show ys ∈ {xs. set (tl xs) ⊆ {..<p} ∧ length xs = k+1 ∧ hd xs = x}
     by simp
 next
   fix ys assume ys: ys ∈ {xs. set (tl xs) ⊆ {..<p} ∧ length xs = k+1 ∧ hd xs =
x}
```

```
    then have facts: set (tl ys) ⊆ {..<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 ∈ {xs. set xs ⊆ {..<p} ∧ length xs = k}
      using azs facts len-zs by simp
    ultimately show ys ∈ (#) x ` {xs. set xs ⊆ {..<p} ∧ length xs = k}
      by (metis image-eqI)
  qed
  finally show set-spmf (map-spmf ((#) x) (sample-uniform-list k p))
    = {xs. set (tl xs) ⊆ {..<p} ∧ length xs = k + 1 ∧ hd xs = x}
    .
qed

lemma set-spmf-lhs-imp-length-gr-0: xs ∈ set-spmf (map-spmf ((#) x) (sample-uniform-list
k p))
  ⟹ 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 ← sample-uniform p;
        map-spmf ((#) x) (sample-uniform-list k p)} = sample-uniform-list (k+1)
p
  (is ?lhs = ?rhs)
proof −
  have ∀ s. spmf ?lhs s = spmf ?rhs s
  proof
    fix s
    have spmf-rhs: spmf ?rhs s = indicator {x. set x ⊆ {..<p} ∧ length x = k+1}
s / real p^(k+1)
      unfolding spmf-sample-uniform-list ..
    have spmf-lhs: spmf ?lhs s
      = (∑x<p. ennreal (spmf (map-spmf ((#) x) (sample-uniform-list k p)) s))

      / of-nat (card {..<p})
      unfolding ennreal-spmf-bind
      unfolding sample-uniform-def
      unfolding nn-integral-spmf-of-set
      ..
    show spmf ?lhs s = spmf ?rhs s
    proof (cases s)
      case Nil
      have spmf ?lhs s = 0
      proof −
```

```
      have ⋀x. spmf (map-spmf ((#) x) (sample-uniform-list k p)) s = 0
        using set-spmf-lhs set-spmf-lhs-imp-length-gr-0 spmf-eq-0-set-spmf
        unfolding Nil
        by fast
      then show ?thesis
       using spmf-lhs by force
    qed
    moreover have spmf ?rhs s = 0
      using spmf-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 spmf ?lhs s
        = (indicator {x. set x ⊆ {..<p} ∧ length x = k} s-tl / (real p^(k+1)))
      proof −
        have spmf-is-sum: spmf ?lhs s =
            sum (λx. ennreal (spmf (map-spmf ((#) x) (sample-uniform-list k p))
s)) {..<p}
            / real (card {..<p})
            unfolding spmf-lhs
            using ennreal-of-nat-eq-real-of-nat by presburger
       also have sum (λx. ennreal (spmf (map-spmf ((#) x) (sample-uniform-list
k p)) s)) {..<p}
          = sum (λx. ennreal (spmf (map-spmf ((#) x) (sample-uniform-list k p))
s)) (({..<p}−{s-hd}) ∪ {s-hd})
          by (simp add: True insert-absorb)
       also have ... = sum (λx. ennreal (spmf (map-spmf ((#) x) (sample-uniform-list
k p)) s)) ({..<p}−{s-hd})
          + ennreal (spmf (map-spmf ((#) s-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 (spmf (map-spmf ((#) s-hd) (sample-uniform-list
k p)) s)
        proof −
          have ⋀x. x ∈ {..<p}−{s-hd} ⟶ ennreal (spmf (map-spmf ((#) x)
(sample-uniform-list k p)) s) = 0
          proof
            fix x
            assume x ∈ {..<p} − {s-hd}
            then have x ≠ s-hd by blast
            then show ennreal (spmf (map-spmf ((#) x) (sample-uniform-list k
p)) s) = 0
              using set-spmf-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 ⊆ {..<p} ∧ length x = k} s-tl / (real
p^k)
          unfolding spmf-sample-uniform-list ..
      finally have ennreal (spmf ?lhs s) = ennreal ((indicator {x. set x ⊆ {..<p}
∧ length x = k} s-tl / (real p^k))) /  ennreal (real (card {..<p}))
          by argo
        then have spmf ?lhs s = (indicator {x. set x ⊆ {..<p} ∧ length x = k}
s-tl / (real p^k)) / (real (card {..<p}))
          using assms(1) divide-ennreal by auto
        also have ... = (indicator {x. set x ⊆ {..<p} ∧ length x = k} s-tl / (real
p^(k+1)))
          by auto
        finally show ?thesis .
      qed

      moreover have spmf ?rhs s
        = (indicator {x. set x ⊆ {..<p} ∧ length x = k} s-tl / (real p^(k+1)))
      proof −
        have (s-hd # s-tl) ∈ {x. set x ⊆ {..<p} ∧ length x = k + 1}
        ⟷ s-tl ∈ {x. set x ⊆ {..<p} ∧ length x = k}
          by (simp add: True)
        then show ?thesis
          unfolding spmf-sample-uniform-list Cons
          by (metis (no-types, lifting) indicator-simps(1) indicator-simps(2))
      qed
      ultimately show ?thesis by presburger
    next
      case False
      have spmf ?lhs s = 0
    proof −
      have ⋀x. x<p ⟶ spmf (map-spmf ((#) x) (sample-uniform-list k p)) s =
0
        unfolding Cons
        using set-spmf-lhs spmf-eq-0-set-spmf False
        by fastforce
      then show ?thesis
       using spmf-lhs by force
    qed
    moreover have spmf ?rhs s = 0
      using spmf-rhs
      unfolding Cons
```

```
      using False
      by fastforce
    ultimately show ?thesis by presburger
    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: sample_uniform_list (k+1) = Cons (sample_uniform) (sample_uniform_list k)

```
corollary pretty-Cons-random-list-split:
  assumes p>1
  shows sample-uniform-list (k+1) p =
    do {x ← sample-uniform p;
        xs ← (sample-uniform-list k p);
        return-spmf (x#xs)}
  (is ?lhs = ?rhs)
proof −
  have ?rhs = do {x ← sample-uniform p;
          map-spmf ((#) x) (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 sample-uniform-poly :: nat ⇒ 'a::zero poly spmf
  where sample-uniform-poly t = spmf-of-set {p. degree p ≤ t}

lemma of-int-mod-inj-on-ff: inj-on (of-int-mod-ring ∘ int:: nat ⇒ 'e::prime-card
mod-ring) {..<CARD ('e)}
proof
  fix x
  fix y
  assume x: x ∈ {..<CARD('e)}
  assume y: y ∈ {..<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 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 (*sample-uniform-poly t::'e mod-ring poly spmf*) = *do* {
evals::(*'e*::*prime-card*) *mod-ring list* ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*:: *nat* ⇒ *'e mod-ring*)) (*sample-uniform-list* (*t+1*) (*CARD* (*'e*)));
return-spmf (*lagrange-interpolation-poly* (*zip I evals*))}
(**is** *?lhs = ?rhs*)
proof −
have *?rhs = do* {
evals ← *spmf-of-set* {*x*::*'e mod-ring list*. *length x = t + 1*};
return-spmf (*lagrange-interpolation-poly* (*zip I evals*))}
proof −
have *uni-list-set-is-card-set*: (⋃ (*set* ‘ {*x*::*nat list*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *length x = t + (1*::*nat)*}))
= {..<*CARD*(*'e*)}
proof
show {..<*CARD*(*'e*)} ⊆ ⋃ (*set* ‘ {*x*::*nat list*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *length x = t + (1*::*nat)*})
proof
fix *x*
assume *x* ∈ {..<*CARD*(*'e*)}
then have *replicate* (*t+1*) *x* ∈ {*x*::*nat list*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *length x = t + (1*::*nat)*}
by *fastforce*
then show *x* ∈ ⋃ (*set* ‘ {*x*::*nat list*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *length x = t + (1*::*nat)*})
by *fastforce*
qed
qed *auto*
have *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*:: *nat* ⇒ *'e mod-ring*)) (*sample-uniform-list* (*t + 1*) *CARD*(*'e*))
= *spmf-of-set* ((*map* (*of-int-mod-ring* ∘ *int*:: *nat* ⇒ *'e mod-ring*)) ‘ {*x*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *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 (*map* (*of-int-mod-ring* ∘ *int*:: *nat* ⇒ *'e mod-ring*)) ‘ {*x*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *length x = t+1*}
= {*x*::*'e mod-ring list*. *length x = t + 1*}
proof
show {*x*::*'e mod-ring list*. *length x = t + (1*::*nat)*}
⊆ *map* (*of-int-mod-ring* ∘ *int*:: *nat* ⇒ *'e mod-ring*) ‘ {*x*::*nat list*. *set x* ⊆ {..<*CARD*(*'e*)} ∧ *length x = t + (1*::*nat)*}
proof

```
      fix x
      assume asm: x ∈ {x::'e 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 = map of-int-mod-ring x-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 (λi. if i≥0 then i else 0)  x-int
          unfolding x-nat using int-nat-eq by simp
        moreover have ∀ x ∈ set x-int. x≥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 ∈ {x::nat list. set x ⊆ {..<CARD('e)} ∧ 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 = map (of-int-mod-ring ∘ int:: nat ⇒ 'e mod-ring) x-nat
        by (simp add: x-eq-map-x-int x-int-x-nat)
       ultimately show x ∈ map (of-int-mod-ring ∘ int:: nat ⇒ 'e mod-ring)
        ` {x::nat list. set x ⊆ {..<CARD('e)} ∧ length x = t + (1::nat)}
        by blast
    qed
   qed auto
   finally show ?thesis by metis
 qed
 also have ... =  do {
     evals ← spmf-of-set {x::'e mod-ring list. length x = t + 1};
     let φ = lagrange-interpolation-poly (zip I evals);
     return-spmf φ} unfolding Let-def ..
 also have ... = map-spmf (λevals. lagrange-interpolation-poly (zip I evals))
                    (spmf-of-set {x::'e mod-ring list. length x = t + 1})
   by (simp add: map-spmf-conv-bind-spmf)
 also have ... = spmf-of-set ((λevals. lagrange-interpolation-poly (zip I evals)) `
{x::'e mod-ring list. length x = t + 1})
 proof (rule map-spmf-of-set-inj-on)
   show inj-on (λevals::'e mod-ring list. lagrange-interpolation-poly (zip I evals))
    {x::'e mod-ring list. length x = t + (1::nat)}
   proof
     fix x y
```

assume *x-in*:$x \in \{x::'e$ *mod-ring list. length* $x = t + (1::nat)\}$
assume *y-in*:$y \in \{x::'e$ *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*<*length I* $\Longrightarrow$ *poly* (*lagrange-interpolation-poly* (*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*<*length I* $\Longrightarrow$ *poly* (*lagrange-interpolation-poly* (*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*<*length I* $\Longrightarrow$ *poly* (*lagrange-interpolation-poly* (*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* $p \leq t\}$
(**is** *?map* = *?set*)
proof −
have $\{p.$ *degree* $p \leq t\}$ = (λ*evals*::$'e$ *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* $p \leq t\}$
$\subseteq$ (λ*evals*::$'e$ *mod-ring list. lagrange-interpolation-poly* (*zip I evals*)) ‘
$\{x::'e$ *mod-ring list. length* $x = t + (1::nat)\}$
proof
fix *x*
assume *asm*: $x \in \{p::'e$ *mod-ring poly. degree* $p \leq t\}$
obtain *x-evals* **where** *x-evals*: *x-evals* = *map* (λi. *poly x i*) *I* **by** *simp*
then have *length x-evals* = *t*+*1*
by (*simp add*: *assms*(*2*))
moreover have *x* = *lagrange-interpolation-poly* (*zip I x-evals*)
apply (*rule uniqueness-of-interpolation-point-list*[*of zip I x-evals x lagrange-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 $x \in (\lambda evals::'e\ mod\text{-}ring\ list.\ lagrange\text{-}interpolation\text{-}poly\ (zip\ I\ evals))\ `\ \{x::'e\ mod\text{-}ring\ list.\ length\ x = t + (1::nat)\}$
by *blast*
qed
show $(\lambda evals::'e\ mod\text{-}ring\ list.\ lagrange\text{-}interpolation\text{-}poly\ (zip\ I\ evals))\ `\ \{x::'e\ mod\text{-}ring\ list.\ length\ x = t + (1::nat)\} \subseteq \{p::'e\ mod\text{-}ring\ poly.\ degree\ p \leq t\}$
proof
fix *x*
assume *asm*: $x \in (\lambda evals::'e\ mod\text{-}ring\ list.\ lagrange\text{-}interpolation\text{-}poly\ (zip\ I\ evals))\ `\ \{x::'e\ mod\text{-}ring\ list.\ length\ x = t + (1::nat)\}$
then show $x \in \{p::'e\ mod\text{-}ring\ poly.\ degree\ p \leq t\}$
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 <= 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-*G_p: *t-DL* G_p *max-deg of-int-mod-ring* ∘ *int pow-mod-ring* G_p
unfolding *t-DL-def*
by (*rule* G_p*.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-α-square-free* :: *'a* ⇒ *'e mod-ring poly* ⇒ *'e mod-ring* **where**
find-α-square-free g-pow-α φ = (*let* (*c*, *polys*) = *finite-field-factorization φ*;
deg1-polys = *filter* (λ*f*. *degree f* = *1*) *polys*;
root-list = *map* (λ*p*. *poly.coeff p 0*) *deg1-polys*;
α-roots = *filter* (λ*r*. *g-pow-α* = $\mathbf{g}_{G_p}$ ^$_{G_p}$ −*r*) *root-list*
in −*α-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-α* :: *'a* ⇒ *'e mod-ring poly* ⇒ *'e mod-ring* **where**
find-α g-pow-α φ = *find-α-square-free g-pow-α* (*radical φ*)

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* ⇒ (*'a*,*'e*) *t-DL.adversary*

where
bind-reduction $\mathcal{A}$ *PK* = *do* {
(*C*, *φ*, -, *φ'*, -) ← $\mathcal{A}$ *PK*;
let α = *find-α* (*PK*!*1*) (*φ* − *φ'*);
return-spmf α}

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-ring poly*, *'a commit*, *unit*) *bind-adversary* ⇒ (*'a*,*'e*) *t-DL.adversary*

where
stronger-bind-reduction $\mathcal{A}$ *PK* = *do* {
(*C*, φ, -, φ', -) ← $\mathcal{A}$ *PK*;
- :: *unit* ← *assert-spmf*($\varphi \neq \varphi'$
∧ *valid-poly* φ
∧ *valid-poly* φ'
∧ (*C* = *g-pow-PK-Prod PK* φ)
∧ (*C* = *g-pow-PK-Prod PK* φ'));
let α = *find-α* (*PK*!*1*) (φ − φ');
return-spmf α}

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-deg degree* $\varphi' \leq$ *max-deg*
shows *g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0..<max-deg+1*]) φ
= *g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0..<max-deg+1*]) φ'
⟷ *poly* (φ − φ') α = *0*
proof
assume *commit-eq*:
g-pow-PK-Prod (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ α ^ *t*) [*0..<max-deg* + *1*]) φ
= *g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ α ^ *t*) [*0..<max-deg* + *1*]) φ'
have *acc-φ*: *g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ α ^ *t*) [*0..<max-deg* + *1*]) φ
= $\mathbf{g}_{G_p}$ ^$_{G_p}$ (*poly* φ α)
by (*metis g-pow-PK-Prod-correct assms*(*1*))
moreover have *acc-φ'*:
g-pow-PK-Prod (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ α ^ *t*) [*0..<max-deg* + *1*]) φ'
= $\mathbf{g}_{G_p}$ ^$_{G_p}$ (*poly* φ' α)
by (*metis g-pow-PK-Prod-correct assms*(*2*))
ultimately show (*poly* (φ − φ') α = *0*)
using *pow-on-eq-card commit-eq* **by** *fastforce*
next
assume *poly-eq-0*: *poly* (φ − φ') α = *0*
have *acc-φ*: *g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ α ^ *t*) [*0..<max-deg* + *1*]) φ
= $\mathbf{g}_{G_p}$ ^$_{G_p}$ (*poly* φ α)
by (*metis g-pow-PK-Prod-correct assms*(*1*))
moreover have *acc-φ'*:
g-pow-PK-Prod (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ α ^ *t*) [*0..<max-deg* + *1*]) φ'

```
    = g_{G_p} ^_{G_p} (poly φ' α )
    by (metis g-pow-PK-Prod-correct assms(2))
  ultimately show g-pow-PK-Prod (map (λt. g ^_{G_p} α ^ t) [0..<max-deg + 1])
φ
                = g-pow-PK-Prod (map (λt. g ^_{G_p} α ^ t) [0..<max-deg + 1]) φ'
    using poly-eq-0 by fastforce
qed

lemma finite-field-factorization-roots:
fixes φ::'e mod-ring poly
  assumes sf-f: square-free φ
    and us: finite-field-factorization φ = (c,us)
  shows poly φ α = 0 ⟷ c=0 ∨ (∃ u∈set us. poly u α = 0)
proof
  assume asm: poly φ α = 0
  have smult: φ = Polynomial.smult c (prod-list us)
    using finite-field-factorization-explicit[OF assms] ..
  then show c = 0 ∨ (∃ u∈set us. poly u α = 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 ∨ (∃ u∈set us. poly u α = 0)
  have smult: φ = Polynomial.smult c (prod-list us)
    using finite-field-factorization-explicit[OF assms] ..
  show poly φ α = 0
  proof (cases c=0)
    case True
    then show ?thesis using smult by force
  next
    case False
    then have (∃ u∈set us. poly u α = 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 mod-ring poly) ∧ irreducible u
  shows (∃ x. poly u x = 0) ⟷ degree u = 1
proof
  assume asm: ∃ x. poly u x = 0
  show 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*:] ∗ *synthetic-div u c* **by** *simp*
from *c deg-ne-1* **have** *deg-u-pos*: *degree u* ≥ *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*) ≥ *1* **using** *degree-synthetic-div*[*of u c*] **by** *linarith*
then have ¬(*synthetic-div u c*) *dvd 1* **by** *auto*
moreover have ¬[:−*c*, *1*:] *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*: ∀ *x*. *poly u x* = *poly.coeff u 0* + *x*
proof
fix *x*
have *poly u x* = (∑ *i*≤*degree u*. *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 ∃ *x*. *poly u x* = *0*
proof
show *poly u* (−*poly.coeff u 0*) = *0*
using *poly-deg-1* **by** *fastforce*
qed
qed

lemma *poly-eq0-is-find-α-sf-eq-α*:
assumes *φ* ≠ *0* ∧ *square-free φ*
shows *poly φ α* = *0* ⟹ *find-α-square-free* (**g**$_{G_p}$ ^$_{G_p}$ *α*) *φ* = *α*
proof −
assume *asm*: *poly φ α* = *0*
obtain *c polys* **where** *c-polys*: (*c*, *polys*) = *finite-field-factorization φ*
by (*metis prod.exhaust*)
then have *c*≠*0* **using** *assms*
by (*metis finite-field-factorization-explicit smult-0-left*)

then have $\exists\, u \in$ *set polys. poly u* $\alpha = 0$ **using** *c-polys asm*
by (*metis assms finite-field-factorization-roots*)
then obtain *u* **where** *u*: $u \in$ *set polys* $\wedge$ *poly u* $\alpha = 0$ **by** *blast*
then have *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*: *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-α-square-free* (**g** $\hat{}_{G_p}$ α) $\varphi = \alpha$
proof −
have *find-α-square-free* (**g** $\hat{}_{G_p}$ α) φ
$=$ (− (*filter* (λr. **g** $\hat{}_{G_p}$ α = **g** $\hat{}_{G_p}$ $-r$) (*map* (λp. *poly.coeff p 0*) (*filter* (λf. *degree f* $= 1$) (*snd* (*finite-field-factorization* φ))))) ! *0*)
unfolding *find-α-square-free.simps* **by** (*simp add*: *split-def*)
also have ... $=$ (− (*filter* (λr. **g** $\hat{}_{G_p}$ α = **g** $\hat{}_{G_p}$ $-r$) (*map* (λp. *poly.coeff p 0*) (*filter* (λf. *degree f* $= 1$) (*polys*)))) ! *0*)
using *c-polys* **by** (*smt* (*verit*, *best*) *snd-conv*)
also have ... $= \alpha$
proof −
have $u \in$ *set* (*filter* (λf. *degree f* $= 1$) *polys*)
by (*simp add*: ‹*degree u* $= 1$› *u*)
then have $-\alpha \in$ *set* (*map* (λp. *poly.coeff p 0*) (*filter* (λf. *degree f* $= 1$) *polys*))
using *u-coeff0* ‹*degree u* $= 1$› **by** *force*
then have $-\alpha \in$ *set* (*filter* (λr. **g** $\hat{}_{G_p}$ α = **g** $\hat{}_{G_p}$ $- r$) (*map* (λp. *poly.coeff p 0*) (*filter* (λf. *degree f* $= 1$) *polys*)))
by *fastforce*
moreover have $\forall\, xs.\ -\alpha \in$ *set xs* $\longrightarrow$ *set* (*filter* (λr. **g** $\hat{}_{G_p}$ α = **g** $\hat{}_{G_p}$ $- r$) *xs*) $= \{-\alpha\}$
by (*auto simp*: *pow-on-eq-card*)
ultimately have *set* (*filter* (λr. **g** $\hat{}_{G_p}$ α = **g** $\hat{}_{G_p}$ $- r$) (*map* (λp. *poly.coeff p 0*) (*filter* (λf. *degree f* $= 1$) *polys*))) $= \{-\alpha\}$
by *fastforce*
then have *filter* (λr. **g** $\hat{}_{G_p}$ α = **g** $\hat{}_{G_p}$ $- r$) (*map* (λp. *poly.coeff p 0*) (*filter* (λf. *degree f* $= 1$) *polys*)) ! *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*: $\varphi \neq 0 \Longrightarrow$ *poly* φ $\alpha = 0 \Longrightarrow$ *find-α* (**g**$_{G_p}$ $\hat{}_{G_p}$

$\alpha)$ $\varphi = \alpha$
proof −
assume *φ-ne-0*: $\varphi \neq 0$
assume *α-root*: *poly* φ $\alpha = 0$
have (*radical* φ) $\neq 0$
by (*simp add*: *φ-ne-0*)
moreover have *poly* (*radical* φ) $\alpha = 0$
using *α-root same-zeros-radical* **by** *blast*
moreover have *square-free* (*radical* φ)
using ‹(*radical* φ) $\neq 0$› *φ-ne-0 squarefree-radical squarefree-square-free* **by** *blast*
ultimately show *find-α* ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ α) $\varphi = \alpha$
unfolding *find-α.simps* **using** *poly-eq0-is-find-α-sf-eq-α* **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'*: ($\varphi \neq \varphi' \wedge$ *valid-poly* $\varphi \wedge$ *valid-poly* φ'
$\wedge$ (C = *g-pow-PK-Prod* (*map* (λt. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::'*e mod-ring*)^*t*)) [*0..<max-deg+1*]) φ)
$\wedge$ (C = *g-pow-PK-Prod* (*map* (λt. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::'*e mod-ring*)^*t*)) [*0..<max-deg+1*]) φ'))
= ($\varphi \neq \varphi' \wedge$ *valid-poly* $\varphi \wedge$ *valid-poly* φ'
$\wedge$ (C = *g-pow-PK-Prod* (*map* (λt. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::'*e mod-ring*)^*t*)) [*0..<max-deg+1*]) φ)
$\wedge$ (C = *g-pow-PK-Prod* (*map* (λt. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::'*e mod-ring*)^*t*)) [*0..<max-deg+1*]) φ')
$\wedge$ (*poly* (φ − (φ')) (*of-int-mod-ring* (*int* α)::'*e mod-ring*) = *0*))
unfolding *valid-poly-def*
using *commit-eq-is-poly-diff-α-eq-0* **by** *fast*

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

$\wedge$ (C = *g-pow-PK-Prod* (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*]) φ)
$\wedge$ (C = *g-pow-PK-Prod* (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*]) φ')
$\wedge$ (*find-α* ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*))) (φ − φ') = (*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*))
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs* **using** *poly-eq0-imp-find-α-eq-α*
by (*meson right-minus-eq*)
next
assume *?rhs*
then show *?lhs*
unfolding *valid-poly-def*
using *commit-eq-is-poly-diff-α-eq-0* **by** *fast*
qed

lemma *helping-3-α-is-found*: $\varphi \neq \varphi' \wedge$ *valid-poly* $\varphi \wedge$ *valid-poly* φ'
$\wedge$ (C = *g-pow-PK-Prod* (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*]) φ) $\wedge$ (C = *g-pow-PK-Prod* (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*]) φ')
$\wedge$ (*find-α* ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*))) (φ − φ') = (*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*))
$\longleftrightarrow \varphi \neq \varphi' \wedge$ *valid-poly* $\varphi \wedge$ *valid-poly* φ'
$\wedge$ (C = *g-pow-PK-Prod* (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*]) φ) $\wedge$ (C = *g-pow-PK-Prod* (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*]) φ')
$\wedge$ (*find-α* ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*))) (φ − φ') = (*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*))
$\wedge$ *of-int-mod-ring* (*int* α) = *find-α* ((*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*of-int-mod-ring* (*int* α)::$'e$ *mod-ring*)$\hat{}t$)) [*0..<max-deg+1*])!*1*) (φ − φ')
(**is** *?lhs* = *?rhs*)
proof −
have *map* (λ*t*::*nat*. $\mathbf{g}$ $\hat{}_{G_p}$ *of-int-mod-ring* (*int* α) $\hat{}$ *t*) [*0*::*nat..<max-deg* + (*1*::*nat*)] ! (*1*::*nat*) = $\mathbf{g}$ $\hat{}_{G_p}$ *of-int-mod-ring* (*int* α)
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-DL-*G_p*.game* (*stronger-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 *?*α = $\lambda\alpha$. (*of-int-mod-ring* (*int* α)::*'e mod-ring*)
let *?PK* = $\lambda\alpha$. (*map* (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ ((*?*α α)^t)) [*0..<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* $\mathcal{A}$ = *TRY do* {
(*ck*,*vk*) ← *key-gen*;
(*c*, *m*, *d*, *m'*, *d'*) ← $\mathcal{A}$ *ck*;
- :: *unit* ← *assert-spmf*(*m* ≠ *m'* ∧ *valid-poly m* ∧ *valid-poly m'*);
let b = *verify-poly vk m c d*;
let b' = *verify-poly vk m' c d'*;
- :: *unit* ← *assert-spmf* (*b* ∧ *b'*);
return-spmf True} *ELSE return-spmf False*
by (*simp add*: *abstract-commitment.bind-game-alt-def*)
also have ...= *TRY do* {
α ← *sample-uniform* (*order* G_p);
let PK = *map* (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ ((*of-int-mod-ring* (*int* α)::*'e mod-ring*)^t)) [*0..<max-deg+1*];
(*C*, φ, *d*, φ', *d'*) ← $\mathcal{A}$ *PK*;
- :: *unit* ← *assert-spmf*($\varphi \neq \varphi'$ ∧ *valid-poly* φ ∧ *valid-poly* φ');
- :: *unit* ← *assert-spmf* ((*C* = *g-pow-PK-Prod PK* φ) ∧ (*C* = *g-pow-PK-Prod PK* φ'));
return-spmf True} *ELSE return-spmf False*
unfolding *key-gen-def verify-poly-def Setup-def* **by** *simp*
also have ... = *TRY do* {
α ← *sample-uniform* (*order* G_p);
TRY do {
(*C*, φ, *d*, φ', *d'*) ← $\mathcal{A}$ (*?PK* α);
TRY do {
- :: *unit* ← *assert-spmf*($\varphi \neq \varphi'$ ∧ *valid-poly* φ ∧ *valid-poly* φ');
- :: *unit* ← *assert-spmf* ((*C* = *g-pow-PK-Prod* (*?PK* α) φ) ∧ (*C* = *g-pow-PK-Prod* (*?PK* α) φ'));
return-spmf True
} *ELSE return-spmf False*
} *ELSE return-spmf False*
} *ELSE return-spmf False*
unfolding *split-def*
by(*fold try-bind-spmf-lossless2*[*OF lossless-return-spmf*])*simp*
also have ...= *TRY do* {
α ← *sample-uniform* (*order* G_p);
TRY do {
(*C*, φ, *d*, φ', *d'*) ← $\mathcal{A}$ (*?PK* α);
TRY do {

```
      - :: unit ← assert-spmf (φ ≠ φ' ∧ valid-poly φ ∧ valid-poly φ'
      ∧ (C = g-pow-PK-Prod (?PK α) φ) ∧ (C = g-pow-PK-Prod (?PK α) φ'));
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
  by (presburger add: assert-collapse)
 also have ...= TRY do {
  α ← sample-uniform (order G_p);
  TRY do {
  (C, φ, d, φ', d') ← A (?PK α);
    TRY do {
      - :: unit ← assert-spmf (φ ≠ φ' ∧ valid-poly φ ∧ valid-poly φ'
      ∧ (C = g-pow-PK-Prod (?PK α) φ) ∧ (C = g-pow-PK-Prod (?PK α) φ')
      ∧ (poly (φ − φ') (?α α) = 0));
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
  } ELSE return-spmf False
   using helping-1-add-poly-φ-m-φ' by presburger
also have ...= TRY do {
  α ← sample-uniform (order G_p);
  TRY do {
  (C, φ, d, φ', d') ← A (?PK α);
    TRY do {
      - :: unit ← assert-spmf (φ ≠ φ' ∧ valid-poly φ ∧ valid-poly φ'
      ∧ (C = g-pow-PK-Prod (?PK α) φ) ∧ (C = g-pow-PK-Prod (?PK α) φ')
      ∧ (find-α (g_{G_p} ^_{G_p} ((?α α))) (φ − φ') = (?α α)));
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
  } ELSE return-spmf False
   using helping-2-factorize-α by presburger
also have ...= TRY do {
  α ← sample-uniform (order G_p);
  TRY do {
  (C, φ, d, φ', d') ← A (?PK α);
    TRY do {
      - :: unit ← assert-spmf (φ ≠ φ' ∧ valid-poly φ ∧ valid-poly φ'
      ∧ (C = g-pow-PK-Prod (?PK α) φ) ∧ (C = g-pow-PK-Prod (?PK α) φ')
      ∧ (find-α (g_{G_p} ^_{G_p} ((?α α))) (φ − φ') = (?α α))
      ∧ ((?α α) = find-α ((?PK α)!1) (φ − φ')));
      return-spmf True
    } ELSE return-spmf False
  } ELSE return-spmf False
  } ELSE return-spmf False
   using helping-3-α-is-found
   by presburger
 also have ...= TRY do {
```

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

by *fastforce*
also have ... = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
$\alpha' \leftarrow$ (*stronger-bind-reduction* $\mathcal{A}$) (*?PK* α);
-::*unit* $\leftarrow$ *assert-spmf* ((*?*α α) = α');
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 Gp

sublocale *t-SDH-*G_p: *t-SDH* G_p *max-deg of-int-mod-ring* ∘ *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-ring*, *'e mod-ring*, *'a witness*) *eval-bind-adversary* ⇒ (*'a*,*'e*) *t-SDH.adversary*
where
  eval-bind-reduction $\mathcal{A}$ *PK* = *do* {
  (*C*, *i*, *φ-of-i*, *w-i*, *φ'-of-i*, *w'-i*) ← $\mathcal{A}$ *PK*;
  return-spmf (−*i*::*'e mod-ring*, (*w-i* ÷$_{G_p}$ *w'-i*) ^$_{G_p}$ (*1* / (*φ'-of-i* − *φ-of-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-ring*, *'e mod-ring*, *'a witness*) *eval-bind-adversary* ⇒ (*'a*,*'e*) *t-SDH.adversary*
where
  ext-eval-bind-reduction $\mathcal{A}$ *PK* = *do* {
  (*C*, *i*, *v*, *w*, *v'*, *w'*) ← $\mathcal{A}$ *PK*;
  - :: *unit* ← *assert-spmf* (*v* ≠ *v'*
                ∧ *valid-argument i*
                ∧ *valid-eval* (*v*,*w*)
                ∧ *valid-eval* (*v'*,*w'*)
                ∧ *verify-eval PK C i* (*v*,*w*)
                ∧ *verify-eval PK C i* (*v'*,*w'*));

return-spmf $(-i::'e$ *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* $\neq$ φ'-*of-i* $\wedge$ *w-i* $\neq$ *w'-i*
$\wedge$ *w-i* $\in$ *carrier* G_p $\wedge$ *w'-i* $\in$ *carrier* G_p
$\wedge$ *verify-eval* $((map\ (\lambda t.\ \mathbf{g}_{G_p} \hat{}_{G_p} ((\alpha)\hat{}t))\ [0..<max\text{-}deg{+}1]))\ C\ i\ (\varphi\text{-}of\text{-}i, w\text{-}i)$
$\wedge$ *verify-eval* $((map\ (\lambda t.\ \mathbf{g}_{G_p} \hat{}_{G_p} ((\alpha)\hat{}t))\ [0..<max\text{-}deg{+}1]))\ C\ i\ (\varphi'\text{-}of\text{-}i, w'\text{-}i)$
shows $\mathbf{g}_{G_p} \hat{}_{G_p} (1/(\alpha + (-i))) = (w\text{-}i \div_{G_p} w'\text{-}i) \hat{}_{G_p} (1 \,/\, (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
proof −
let *?PK* $= \lambda\alpha.\ (map\ (\lambda t.\ \mathbf{g}_{G_p} \hat{}_{G_p} ((\alpha)\hat{}t))\ [0..<max\text{-}deg{+}1])$
obtain ψ_i **where** ψ_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** ψ_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* $((\mathbf{g}_{G_p} \hat{}_{G_p} \alpha) \div_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} i)) \otimes_{G_T} ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi\text{-}of\text{-}i)$
$=$ *e w'-i* $((\mathbf{g}_{G_p} \hat{}_{G_p} \alpha) \div_{G_p} (\mathbf{g}_{G_p} \hat{}_{G_p} i)) \otimes_{G_T} ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi'\text{-}of\text{-}i)$
proof −
have *map* $(\lambda t.\ \mathbf{g} \hat{}\ \alpha \hat{}\ t)\ [0..<max\text{-}deg\ +\ 1]\ !\ 1 =\ \mathbf{g} \hat{}\ \alpha$
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* $(\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi\text{-}of\text{-}i)$
$=$ *e w'-i* $(\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi'\text{-}of\text{-}i)$
using *mod-ring-pow-mult-inv-*G_p **by** *presburger*
then have *e* $(\mathbf{g}_{G_p} \hat{}_{G_p} \psi_i)\ (\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi\text{-}of\text{-}i)$
$=$ *e* $(\mathbf{g}_{G_p} \hat{}_{G_p} \psi_i')\ (\mathbf{g}_{G_p} \hat{}_{G_p} (\alpha - i)) \otimes_{G_T} ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} \varphi'\text{-}of\text{-}i)$
using $\psi_i\ \psi_i'$ **by** *fast*
then have $(e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} (\psi_i * (\alpha - i) + \varphi\text{-}of\text{-}i)$
$= (e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}) \hat{}_{G_T} (\psi_i' * (\alpha - i) + \varphi'\text{-}of\text{-}i)$
using *e-bilinear*
by (*metis* G_p*.generator-closed addition-in-exponents-on-e*)
then have $\psi_i * (\alpha - i) + \varphi\text{-}of\text{-}i = \psi_i' * (\alpha - i) + \varphi'\text{-}of\text{-}i$
using *pow-on-eq-card-GT-carrier-ext'*
by *blast*
then have $(\psi_i - \psi_i') * (\alpha - i) = \varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i$
by (*simp add*: *algebra-simps*)
then have $(\psi_i - \psi_i')/(\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i) = 1/(\alpha - i)$
by (*metis* $\psi_i\ \psi_i'$ *assms divide-divide-eq-left divide-self-if eq-iff-diff-eq-0*)

moreover have $(w\text{-}i \div_{G_p} w'\text{-}i) \mathbin{\hat{}}_{G_p} (1 \,/\, (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i)) = \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} ((\psi_i - \psi_i')/(\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
proof −
have $(w\text{-}i \div_{G_p} w'\text{-}i) \mathbin{\hat{}}_{G_p} (1 \,/\, (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
$= ((\mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} \psi_i) \div_{G_p} (\mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} \psi_i')) \mathbin{\hat{}}_{G_p} (1 \,/\, (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
using $\psi_i\ \psi_i'$ **by** *fast*
also have $\ldots = (\mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\psi_i - \psi_i')) \mathbin{\hat{}}_{G_p} (1 \,/\, (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
using *mod-ring-pow-mult-inv-*G_p **by** *presburger*
also have $\ldots = \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} ((\psi_i - \psi_i')/(\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
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*: $v \neq v'$
$\wedge$ *valid-argument i*
$\wedge$ *valid-eval* (v,w)
$\wedge$ *valid-eval* (v',w')
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])$ $C\ i\ (v,w)$
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])$ $C\ i\ (v',w')$
$\longleftrightarrow$
$v \neq v' \wedge w \neq w'$
$\wedge$ *valid-argument i*
$\wedge$ *valid-eval* (v,w)
$\wedge$ *valid-eval* (v',w')
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])$ $C\ i\ (v,w)$
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])$ $C\ i\ (v',w')$
proof
assume *asm*: $v \neq v'$
$\wedge$ *valid-argument i*
$\wedge$ *valid-eval* (v,w)
$\wedge$ *valid-eval* (v',w')
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])\ C\ i\ (v,w)$
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])\ C\ i\ (v',w')$
have $w \neq w'$
proof −
obtain *w-pow* **where** *w-i-pow*: $w = \mathbf{g} \mathbin{\hat{}}_{G_p} w\text{-}pow$
proof −

have $w \in$ *carrier* G_p
using *asm* **by** (*simp add*: *valid-eval-def*)
then show $(\bigwedge$*w-pow*. $w = \mathbf{g} \mathbin{\hat{}} w\text{-}pow \Longrightarrow thesis) \Longrightarrow thesis$
by (*metis* G_p*.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int*)
qed
obtain *w'-pow* **where** *w'-i-pow*: $w' = \mathbf{g} \mathbin{\hat{}}_{G_p} w'\text{-}pow$
proof −
have $w' \in$ *carrier* G_p
using *asm* **by** (*simp add*: *valid-eval-def*)
then show $(\bigwedge$*w-pow*. $w' = \mathbf{g} \mathbin{\hat{}} w\text{-}pow \Longrightarrow thesis) \Longrightarrow thesis$
by (*metis* G_p*.generatorE g-pow-to-int-mod-ring-of-int-mod-ring int-pow-int*)
qed

from *asm*
have $e\ w\ ((map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])\ !\ 1 \otimes inv\ (\mathbf{g} \mathbin{\hat{}}_{G_p} i))$
$\otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} v$
$= e\ w'\ ((map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])\ !\ 1 \otimes inv\ (\mathbf{g} \mathbin{\hat{}}_{G_p} i))$
$\otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} v'$ **unfolding** *verify-eval-def* **by** *force*
then have $e\ (\mathbf{g} \mathbin{\hat{}}_{G_p} w\text{-}pow)\ ((\mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} \alpha) \otimes inv\ (\mathbf{g} \mathbin{\hat{}}_{G_p} i))$
$\otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} v$
$= e\ (\mathbf{g} \mathbin{\hat{}}_{G_p} w'\text{-}pow)\ ((\mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} \alpha) \otimes inv\ (\mathbf{g} \mathbin{\hat{}}_{G_p} i))$
$\otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} v'$
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 $e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} (w\text{-}pow * (\alpha - i))$
$\otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} v$
$= e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} (w'\text{-}pow * (\alpha - i))$
$\otimes_{G_T} e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} v'$
by (*metis* G_p*.generator-closed e-bilinear mod-ring-pow-mult-inv-*G_p)
then have $e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} (w\text{-}pow * (\alpha - i) + v)$
$= e\ \mathbf{g}\ \mathbf{g} \mathbin{\hat{}}_{G_T} (w'\text{-}pow * (\alpha - i) + v')$
using *mod-ring-pow-mult-*G_T **by** *auto*
then have $w\text{-}pow * (\alpha - i) + v = w'\text{-}pow * (\alpha - i) + v'$
using *pow-on-eq-card-GT-carrier-ext'* **by** *blast*
then have $w\text{-}pow \neq w'\text{-}pow$ **using** *asm* **by** *force*

then show *?thesis*
using *w-i-pow w'-i-pow pow-on-eq-card* **by** *presburger*
qed
then show $v \neq v' \wedge w \neq w'$
$\wedge$ *valid-argument i*
$\wedge$ *valid-eval* (v,w)
$\wedge$ *valid-eval* (v',w')
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])$
$C\ i\ (v,w)$
$\wedge$ *verify-eval* $(map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (\alpha \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1])$
$C\ i\ (v',w')$

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

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

14.4 Proof

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

theorem *eval-bind-game-eq-t-SDH-strong-ext-red*:
shows *eval-bind-game* $\mathcal{A}$ = *t-SDH-*G_p*.game* (*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 *?*α = $\lambda\alpha$. (*of-int-mod-ring* (*int* α)::'*e mod-ring*)
let *?PK* = $\lambda\alpha$. (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*?*α α)$\hat{}t$)) [0..<*max-deg*+1])

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

```
                    ∧ valid-eval (v,w)
                    ∧ valid-eval (v′,w′)
                    ∧ verify-eval (?PK α) C i (v,w)
                    ∧ verify-eval (?PK α) 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 {
   α ← sample-uniform (order G_p);
  (C, i, v, w, v′, w′) ← A (?PK α);
  - :: unit ← assert-spmf (v ≠ v′
                    ∧ valid-argument i
                    ∧ valid-eval (v,w)
                    ∧ valid-eval (v′,w′)
                    ∧ verify-eval (?PK α) C i (v,w)
                    ∧ verify-eval (?PK α) 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′) ← key-gen;
  (C, i, v, w, v′, w′) ← A PK;
  - :: unit ← assert-spmf (v ≠ v′
                    ∧ valid-argument i
                    ∧ valid-eval (v,w)
                    ∧ valid-eval (v′,w′)
                    ∧ verify-eval PK C i (v,w)
                    ∧ 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,-) ← key-gen;
 TRY do {
    (C, i, v, w, v′, w′) ← A PK;
    TRY do {
    - :: unit ← assert-spmf (v ≠ v′
                    ∧ valid-argument i
                    ∧ valid-eval (v,w)
                    ∧ valid-eval (v′,w′)
                    ∧ 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*,-) ← *key-gen*;
TRY do {
(*C*, *i*, *v*, *w*, *v'*, *w'*) ← $\mathcal{A}$ *PK*;
TRY do {
- :: *unit* ← *assert-spmf* (*v* ≠ *v'*$\wedge$ *valid-argument i* $\wedge$ *valid-eval* (*v*,*w*) $\wedge$ *valid-eval* (*v'*,*w'*));
- :: *unit* ← *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*,-) ← *key-gen*;
TRY do {
(*C*, *i*, *v*, *w*, *v'*, *w'*) ← $\mathcal{A}$ *PK*;
TRY do {
- :: *unit* ← *assert-spmf* (*v* ≠ *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*,-) ← *key-gen*;
(*C*, *i*, *v*, *w*, *v'*, *w'*) ← $\mathcal{A}$ *PK*;
- :: *unit* ← *assert-spmf* (*v* ≠ *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*: *spmf* (*t-SDH-*G_p*.game* (*ext-eval-bind-reduction* $\mathcal{A}$)) *True*
$\leq$ *spmf* (*t-SDH-*G_p*.game* (*eval-bind-reduction* $\mathcal{A}$)) *True*
(**is** *spmf ?lhgame True* $\leq$ *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 *?α* = $\lambda\alpha$. (*of-int-mod-ring* (*int* α)::'*e mod-ring*)
let *?PK* = $\lambda\alpha$. (*map* (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ ((*?α* α)^t)) [*0..<max-deg+1*])

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

have *bind-red-ext*: *t-SDH-*G_p*.game* (*ext-eval-bind-reduction* $\mathcal{A}$) = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
(C, i, v, w, v', w') $\leftarrow \mathcal{A}$ (*?PK* α);
- :: *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* α) C i (v,w)
$\wedge$ *verify-eval* (*?PK* α) C i (v',w'));
-::*unit* $\leftarrow$ *assert-spmf* ($\mathbf{g}_{G_p}$ ^$_{G_p}$ (1/(*?α* α + (−i))) = (w ÷$_{G_p}$ w') ^$_{G_p}$ (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* $\mathcal{A}$)])

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

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

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

have *ennreal* (*spmf* (*t-SDH-*G_p*.game* (*ext-eval-bind-reduction* $\mathcal{A}$)) *True*)
$\leq$ *ennreal* (*spmf* (*t-SDH-*G_p*.game* (*eval-bind-reduction* $\mathcal{A}$)) *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.emeasure-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-SDH-*G_p*.advantage* (*eval-bind-reduction* $\mathcal{A}$)
using *overestimate-reductions*
unfolding *t-SDH-*G_p*.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*

$= ('a'\ ck \times 'state') \Rightarrow ('e'\ argument \times ('e'\ evaluation \times ('a'\ witness \times int\ list)))$
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* =
$('a'\ commit \times int\ list) \Rightarrow$
$('e'\ mod\text{-}ring\ poly \times unit)\ spmf$

restrict for AGM adversaries 1 and 2

realized by the following two interpretations:

interpretation *AGM1*: *Algebraic-Algorithm* G_p *listS* G_p*.groupS prodC* G_p*.groupC noConstrain*
by (*unfold-locales*)

interpretation *AGM2*: *Algebraic-Algorithm* G_p *prodS* (*listS* G_p*.groupS*) *noSelect*

prodC noConstrain (*prodC noConstrain* G_p*.groupC*)
by (*unfold-locales*)

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

$\Rightarrow ('e, 'state, 'a)$ *knowledge-soundness-adversary2-AGM* $\Rightarrow ('a, 'e)$ *extractor* $\Rightarrow$ *bool spmf*
where *knowledge-soundness-game-AGM* $\mathcal{A}1\ \mathcal{A}2\ \mathcal{E}$ = *TRY do* {
(*ck*,*vk*) ← *key-gen*;
((*c*,*cvec*),σ) ← *AGM1.restrict* $\mathcal{A}1$ *ck*;
(*p*,*td*) ← $\mathcal{E}$ (*c*,*cvec*);
(*i*, *p-i*, *w*, *wvec*) ← *AGM2.restrict* $\mathcal{A}2$ (*ck*,σ);
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* $\Rightarrow$ $('state, 'a)$ *AGM-knowledge-soundness-adversary1*
$\Rightarrow ('e, 'state, 'a)$ *knowledge-soundness-adversary2-AGM*

⇒ ('a ck, 'a commit, 'e argument, 'e evaluation, 'a witness) eval-bind-adversary

where
knowledge-soundness-reduction $\mathcal{E}$ $\mathcal{A}1$ $\mathcal{A}2$ ck = do {
((c,cvec),σ) ← AGM1.restrict $\mathcal{A}1$ ck;
(p,td) ← $\mathcal{E}$ (c,cvec);
(i, p-i, w, wvec) ← AGM2.restrict $\mathcal{A}2$ (ck,σ);
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) extractor **where**
E (c,cvec) = return-spmf (Poly (map (of-int-mod-ring::int ⇒'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 ⇒ ('state, 'a) AGM-knowledge-soundness-adversary1
⇒ ('e, 'state, 'a) knowledge-soundness-adversary2-AGM
⇒ ('a ck, 'a commit, 'e argument, 'e evaluation, 'a witness) eval-bind-adversary

where
knowledge-soundness-reduction-ext $\mathcal{E}$ $\mathcal{A}1$ $\mathcal{A}2$ ck = do {
((c,cvec),σ) ← AGM1.restrict $\mathcal{A}1$ ck;
(p,td) ← $\mathcal{E}$ (c,cvec);
(i, p-i, w, wvec) ← AGM2.restrict $\mathcal{A}2$ (ck,σ);
- :: unit ← 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 (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^t)) [0..<max-deg+1];
vk = map (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^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)
∧ 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')
proof −
  define ck where ck-def: ck = map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1]
  define vk where vk-def: vk = map (λt. g_{G_p} ^_{G_p} (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)
        = g_{G_p} ^_{G_p} (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 ^_{G_p} α ^ t) [0..<max-deg + 1])
      (ψ-of (Poly ?cvec) i))
      = (g ^_{G_p} 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 ^_{G_p} α ^ t) [0..<max-deg + 1] ! 1 = g ^_{G_p} 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 ^_{G_p} poly (Poly ?cvec) α) = c
    proof −
      have (g ^_{G_p} poly (Poly ?cvec) α)
          = g-pow-PK-Prod (map (λt. g ^_{G_p} α ^ t) [0..<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 (λi acc. acc ⊗_{G_p} (g_{G_p} ^_{G_p} (α^i))
^_{G_p} (poly.coeff (Poly ?cvec) i))
```

```
    [0..<Suc (degree (Poly ?cvec))] 1_{G_p}
    by (rule g-pow-PK-Prod-to-fold)(fact deg-poly-calc-vec-le-max-deg)
  also have ...
  =fold (λ i acc. acc ⊗_{G_p} (g_{G_p} ^_{G_p} (α^i)) ^_{G_p} (?cvec!i)) [0..<max-deg+1]
1_{G_p}
  proof −
    have fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} ?cvec ! i) [0..<max-deg +
1] 1
      = fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} ?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 (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} ?cvec ! i)
                    [Suc (degree (Poly ?cvec))..<max-deg + 1]
                    (fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} ?cvec ! i)
                     [0..<Suc (degree (Poly ?cvec))] 1)
      by fastforce
    also have ... = fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} poly.coeff (Poly
?cvec) i)
                    [0..<Suc (degree (Poly ?cvec))]
                    1
    proof −
     have fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} ?cvec ! i) [0..<Suc (degree
(Poly ?cvec))] 1
       = fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} 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))] ⟹
             (λacc. acc ⊗ (g ^_{G_p} α ^ x) ^_{G_p} ?cvec ! x) =
             (λacc. acc ⊗ (g ^_{G_p} α ^ x) ^_{G_p} 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 ^_{G_p} α ^ x) ^_{G_p} ?cvec ! x = acc ⊗ (g ^_{G_p} α
^ x) ^_{G_p} poly.coeff (Poly ?cvec) x
            by presburger
        qed
```

qed *simp+*
moreover have $\forall$ *init* $\in$ *carrier* G_p.
fold (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?cvec* ! *i*)
[*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*]
init
= *init*
proof
fix *init* ::'*a*
assume *init-in-carrier*: *init* $\in$ *carrier* G_p
have *fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?cvec* ! *i*)
[*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*]
init = *fold* (λ*i acc. acc* $\otimes$ **1**)
[*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*]
init
proof (*rule List.fold-cong*)
show $\bigwedge$*x. x* $\in$ *set* [*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*] $\Longrightarrow$
(λ*acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *?cvec* ! *x*) = (λ*acc. acc* $\otimes$ **1**)
proof
fix *x*::*nat*
fix *acc* ::'*a*
assume *asm*: *x* $\in$ *set* [*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*]
show *acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *?cvec* ! *x* = *acc* $\otimes$ **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 order.refl trans-le-add2*)
then have (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *?cvec* ! *x* = **1** **by** *simp*
then show *?thesis* **by** *argo*
qed
qed
qed *simp+*
also have ... = *init*
proof (*induction max-deg*)
case *0*
then show *?case* **by** *fastforce*
next
case (*Suc max-deg*)
have *fold* (λ*i acc. acc* $\otimes$ **1**) [*Suc* (*degree* (*Poly ?cvec*))..<*Suc max-deg* + *1*] *init*
= *fold* (λ*i acc. acc* $\otimes$ **1**) ([*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*] @ [*Suc max-deg*]) *init*
by (*simp add*: *init-in-carrier*)
also have ... = *fold* (λ*i acc. acc* $\otimes$ **1**) [*Suc max-deg*] (*fold* (λ*i acc. acc* $\otimes$ **1**) [*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*] *init*)
by *force*
also have ... = *fold* (λ*i acc. acc* $\otimes$ **1**) [*Suc max-deg*] *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* (λ*i acc. acc* ⊗ (**g** ^$_{G_p}$ α ^ *i*) ^$_{G_p}$ *?cvec* ! *i*)
[*Suc* (*degree* (*Poly ?cvec*))..<*max-deg* + *1*]
init
= *init* **.**
qed
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) G_p.*generator-closed* G_p.*int-pow-closed* ‹**g** ^$_{G_p}$ *poly* (*Poly ?cvec*) α = *g-pow-PK-Prod* (*map* (λ*t*. **g** ^$_{G_p}$ α ^ *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* ⊗$_{G_p}$ (*map* (λ*t*. **g**$_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg+1*])!*i* ^$_{G_p}$ (*?cvec*!*i*)) [*0*..<*max-deg+1*] **1**$_{G_p}$
proof(*rule List.fold-cong*)
show 1 = **1** **by** *simp*
show [*0*..<*max-deg* + *1*] = [*0*..<*max-deg* + *1*] **by** *simp*
show ⋀*x*. *x* ∈ *set* [*0*..<*max-deg* + *1*] ⟹
(λ*acc. acc* ⊗ (**g** ^$_{G_p}$ α ^ *x*) ^$_{G_p}$ *?cvec* ! *x*) =
(λ*acc. acc* ⊗ *map* (λ*t*. **g** ^$_{G_p}$ α ^ *t*) [*0*..<*max-deg* + *1*] ! *x* ^$_{G_p}$ *?cvec* ! *x*)
proof
fix *x*::*nat*
fix *acc* :: *'a*
assume *asm*: *x* ∈ *set* [*0*..<*max-deg* + *1*]
show *acc* ⊗ (**g** ^$_{G_p}$ α ^ *x*) ^$_{G_p}$ *?cvec* ! *x*
= *acc* ⊗ *map* (λ*t*. **g** ^$_{G_p}$ α ^ *t*) [*0*..<*max-deg* + *1*] ! *x* ^$_{G_p}$ *?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* (λ *i acc. acc* ⊗$_{G_p}$ (*map* (λ*t*. **g**$_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg+1*])!*i* ^$_{G_p}$ (*of-int-mod-ring* (*cvec*!*i*))) [*0*..<*max-deg+1*] **1**$_{G_p}$
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 (λ*acc. acc* ⊗ *map* (λ*t*. **g** ^ α ^ *t*) [*0*..<*max-deg* + *1*] ! *x* ^ *map of-int-mod-ring cvec* ! *x*) =
(λ*acc. acc* ⊗ *map* (λ*t*. **g** ^ α ^ *t*) [*0*..<*max-deg* + *1*] ! *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 G_p ⟹  g [^]_{G_p}
(to-int-mod-ring (of-int-mod-ring x::'e mod-ring)) = g [^]_{G_p} x
          proof −
           fix g x
           assume g-in-carrier: g ∈ carrier G_p
          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 [^]_{G_p} (to-int-mod-ring (of-int-mod-ring x::'e mod-ring))
= g [^]_{G_p} x
             using carrier-pow-mod-order-G_p 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) G_p.generator-closed G_p.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 𝒜1 𝒜2 E =
  eval-bind-game (knowledge-soundness-reduction-ext E 𝒜1 𝒜2)
proof −
  note [simp] = Let-def split-def

  have knowledge-soundness-game-AGM 𝒜1 𝒜2 E =
    TRY do {
      (ck,vk) ← key-gen;
      ((c,cvec),σ) ← AGM1.restrict 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;
       return-spmf (verify-eval vk c i (p-i,w) ∧ p-i ≠ p-i' ∧ valid-argument i ∧
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) ← 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, p-i, (w, wvec)) ← AGM2.restrict 𝒜2 (ck,σ);
      let (p-i',w') = Eval ck td p i;
       return-spmf (verify-eval vk c i (p-i,w) ∧ p-i ≠ p-i' ∧ valid-argument i ∧
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) ← 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);
       return-spmf (verify-eval vk c i (p-i,w) ∧ p-i ≠ p-i' ∧ valid-argument i ∧
```

```
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..<length ck] 1);
              TRY do {
                return-spmf (verify-eval vk c i (p-i,w) ∧ p-i ≠ p-i' ∧ valid-argument
i ∧ valid-eval (p-i,w))
              } ELSE return-spmf False
            } ELSE return-spmf False
          } 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,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..<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
          } 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*,*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* G_p);
let (α::'*e mod-ring*) = *of-int-mod-ring* (*int x*);
let ck = *map* (λ*t*. $\mathbf{g}_{G_p}$ [^]$_{G_p}$ (α [^] *t*)) [*0*..<*max-deg*+*1*];
let vk = *map* (λ*t*. $\mathbf{g}_{G_p}$ [^]$_{G_p}$ (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 :: *nat* ← *sample-uniform* (*order* G_p);
let (α::′*e mod-ring*) = *of-int-mod-ring* (*int x*);
let ck = *map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg*+*1*];
let vk = *map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg*+*1*];
((*c*,*cvec*),σ) ← $\mathcal{A}$*1 ck*;
let (*p*,*td*) = (*Poly* (*map* (*of-int-mod-ring*::*int* ⇒′*e mod-ring*) *cvec*),());
(*i*, *p-i*, (*w*, *wvec*)) ← *AGM2.restrict* $\mathcal{A}$*2* (*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**
∧ *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(*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*) ← *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*,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**
∧ *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*
unfolding *key-gen-def Setup-def* **by** *force*
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);
    - :: 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) ← 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, p-i, (w, wvec)) ← AGM2.restrict A2 (ck,σ);
    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 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 (
```

```
        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
   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) ← key-gen;
    ((c,cvec),σ) ← AGM1.restrict 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 (
      valid-eval (p-i,w)
      ∧ 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(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) ← key-gen;
    ((c,cvec),σ) ← AGM1.restrict A1 ck;
   (p,td) ← E (c,cvec);
   (i, p-i, (w, wvec)) ← AGM2.restrict A2 (ck,σ);
   - :: unit ← assert-spmf ( valid-eval (p-i,w));
   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'));
```

```
    - :: 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] = 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 $\leftarrow$ *assert-spmf* $(v \neq v' \wedge$ *valid-argument* $i \wedge$ *valid-eval* $(v,\ w) \wedge$ *valid-eval* $(v',\ w'))$;
let $b = verify\text{-}eval\ vk\ c\ i\ (v,w)$;
let $b' = verify\text{-}eval\ vk\ c\ i\ (v',w')$;
return-spmf $(b \wedge b')\}$ *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* $\mathcal{A}$ $\mathcal{A}'$) =
TRY do {
$(ck,\ vk) \leftarrow$ *key-gen*;
$((c,cvec),\sigma) \leftarrow$ *AGM1.restrict* $\mathcal{A}$ *ck*;
$(p,td) \leftarrow E\ (c,cvec)$;
$(i,\ v,\ (w,\ wvec)) \leftarrow$ *AGM2.restrict* $\mathcal{A}'\ (ck,\sigma)$;
let $(v',w') = Eval\ ck\ td\ p\ i$;
- :: unit $\leftarrow$ *assert-spmf* $(v \neq v' \wedge$ *valid-argument* $i \wedge$ *valid-eval* $(v,\ w) \wedge$ *valid-eval* $(v',\ w'))$;
let $b = verify\text{-}eval\ vk\ c\ i\ (v,w)$;
let $b' = verify\text{-}eval\ vk\ c\ i\ (v',w')$;
return-spmf $(b \wedge b')\}$ *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* (*spmf* (*eval-bind-game* (*knowledge-soundness-reduction-ext E* $\mathcal{A}$ $\mathcal{A}'$)) *True*)
$\leq$ *ennreal* (*spmf* (*eval-bind-game* (*knowledge-soundness-reduction E* $\mathcal{A}$ $\mathcal{A}'$)) *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.emeasure-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* $\mathcal{A}1$ $\mathcal{A}2$ *E*) *True*
$\leq$ *t-SDH-*G_p*.advantage* (*eval-bind-reduction* (*knowledge-soundness-reduction E* $\mathcal{A}1$ $\mathcal{A}2$))
using *evaluation-binding*[*of knowledge-soundness-reduction E* $\mathcal{A}1$ $\mathcal{A}2$]

overestimate-reductions[*of* $\mathcal{A}1$ $\mathcal{A}2$]
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 ⇒ (*'a vk*, *'e argument*,*'state*) *eval-hiding-adversary1*
where
weak-eval-hiding-adversary1 $\mathcal{A}$ *vk* = *do* {
(*I*,σ) ← $\mathcal{A}$ *vk*;
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *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*
⇒
(*'e mod-ring*, *'a vk*, *'a commit*, *'e argument*, *'e evaluation*, *'a witness*, *'state*)
eval-hiding-adversary2 ⇒ *bool spmf*
where *weak-eval-hiding-game* $\mathcal{A}1$ $\mathcal{A}2$ =
TRY do {
p::*'e mod-ring poly* ← *sample-uniform-poly max-deg*;
eval-hiding-game p (*weak-eval-hiding-adversary1* $\mathcal{A}1$) $\mathcal{A}2$

} *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* ⇒
(*'e mod-ring*, *'a vk*, *'a commit*, *'e argument*, *'e evaluation*, *'a witness*, *'state*)
eval-hiding-adversary2 ⇒ *real*
where *weak-eval-hiding-advantage* $\mathcal{A}1$ $\mathcal{A}2$
≡ *spmf* (*weak-eval-hiding-game* $\mathcal{A}1$ $\mathcal{A}2$) *True*

16.1.1 DL game

We instantiate the DL game for the group Gp

sublocale *DL-*G_p: *DL* G_p *of-int-mod-ring* ∘ *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 Gp

sublocale *t-DL-*G_p: *t-DL* G_p *max-deg of-int-mod-ring* ∘ *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* × *'a*) *list* ⇒ *'e mod-ring* ⇒ *'a* **where**
interpolate-on xs-ys α= *do* {
let xs = *map fst xs-ys*;
let lagrange-exp = *map* (λ(*xj*,*yj*). *yj* $\hat{}_{G_p}$ (*poly* (*lagrange-basis-poly xs xj*) α)) *xs-ys*;
fold (λ*x 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* ⇒ *'e argument list* ⇒ *'e argument list*
where *filter-distinct xs ys* = *filter* (λ*x*. *find* ((=) *x*) *ys* = *None*) *xs*

lemma *set-filter-distinct*: *set* (*filter-distinct xs ys*) = {*x*. *x* ∈ *set xs* ∧ *x* ∉ *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* ∈ *set xs* ∧ *x* ∉ *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..<CARD('e)]) I)
```

```
lemma card-map-card: card (set (map (of-int-mod-ring::int ⇒ 'e mod-ring) [0..<CARD('e)]))
= CARD('e)
proof −
  have card ((of-int-mod-ring ∘ int::nat ⇒ 'e mod-ring) ` {0..<CARD('e)}) =
card {0..<CARD('e)}
    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..<CARD('e)]) I) > 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..<CARD('e)]) I ≠ []
    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 ⇒
  ('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
  eval-hiding-adversary2 ⇒ ('a,'e) DL.adversary
where
  reduction 𝒜1 𝒜2 g-pow-a = do {
  (α, PK) ← Setup;
  (I,σ) ← 𝒜1 PK;
  let i = PickDistinct I;
   plain-evals ← map-spmf (map ((of-int-mod-ring::int ⇒ 'e mod-ring) ∘ int))
(sample-uniform-list max-deg (order G_p));
  let evals = g-pow-a # map (λi. g ^_{G_p} i) plain-evals ;
  let C = interpolate-on (zip (i#I) evals) α;
  let W = map (λ(x,y). (y,(C ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} ((1::'e mod-ring)/(α−x))))
(zip I plain-evals);
  φ' ← 𝒜2 PK σ C I W;
  return-spmf (poly φ' i)
  }

fun reduction-tDL :: ('a vk, 'e argument, 'state) eval-hiding-adversary1
⇒ ('a,'e) t-DL.adversary
  where reduction-tDL 𝒜1 PK = do {
  (I,σ) ← 𝒜1 PK;
  let (α'::'e mod-ring) = (case (find (λx. g ^_{G_p} x = PK!1) (PickDistinct I #I))
of Some x => x | None => 1);
  return-spmf α'
  }
```

16.2 Hiding proof

16.2.1 Helping lemmas

```
lemma PK1[simp]: map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1]!1 = g_{G_p} ^_{G_p} α
 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 ∈ set xs ⟹ Some(x) = find (λy. g ^_{G_p} y = g ^_{G_p} x)
xs
proof −
  assume asm: x ∈ set xs
```

```
  then have ∃ y. y ∈ set xs ∧ y=x
    by blast
  then have find (λy. y = x) xs = Some (x)
    by (smt (verit, ccfv-threshold) find-None-iff find-Some-iff2 not-None-eq)
  then show Some(x) = find (λy. g ^_{G_p} y =  g ^_{G_p} x) xs
    by (simp add: pow-on-eq-card)
qed

lemma exchange-lem: length I = max-deg ∧ distinct I ∧ α ∈ set (PickDistinct
I#I)
   ⟷ length I = max-deg ∧ distinct I ∧ α ∈ set (PickDistinct I#I)
  ∧ α = (case (find (λx. g ^_{G_p} x = map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1]!1)
(PickDistinct I#I)) of Some x => x | None => 1)
proof
  assume asm: length I = max-deg ∧ distinct I ∧ α ∈ set (PickDistinct I # I)
  then have some: Some α =
    find (λx. g ^ x = map (λt. g ^ α ^ t) [0..<max-deg + 1] ! 1) (PickDistinct I
# I)
    apply (simp only: PK1)
    apply (rule in-set-impl-find)
    apply simp
    done
  show length I = max-deg ∧
    distinct I ∧
    α ∈ set (PickDistinct I # I) ∧
   α = (case (find (λx. g ^_{G_p} x = map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1]!1)
(PickDistinct I#I)) of Some x => x | None => 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 ≡ (let
    xs = map fst xs-ys
    in sum-list (map (λ (xj,yj).  yj * (poly (lagrange-basis-poly xs xj) x)) xs-ys))
  (is ?lhs≡?rhs)
proof −
  have ?lhs≡let
    xs = map fst xs-ys
   in sum-list (map (λ p. poly p x) (map (λ (xj,yj). Polynomial.smult yj (lagrange-basis-poly
xs xj)) xs-ys))
    unfolding lagrange-interpolation-poly-def Let-def
    by (simp add: poly-sum-list poly-hom.hom-sum-list)
```

also have ...≡*let*
xs = *map fst xs-ys*
in sum-list (*map* ((λ (*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 ...≡*let*
xs = *map fst xs-ys*
in sum-list (*map* ((λ (*xj*,*yj*). *yj* ∗ (*poly* (*lagrange-basis-poly xs xj*) *x*))) *xs-ys*)
by *fastforce*
finally show *?lhs* ≡ *?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* (λ*x prod*. *prod* $\otimes_{G_p}$ *x*) (*map* (λ*x*. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *f x*) *xs*) ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *z*)
= $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *z* $\otimes_{G_p}$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ (*sum-list* (*map f xs*))
proof (*induction xs arbitrary*: *z*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
have *fold* (λ*x prod*. *prod* ⊗ *x*) (*map* (λ*x*. **g** $\hat{}_{G_p}$ *f x*) (*a* # *xs*)) (**g** $\hat{}_{G_p}$ *z*)
= *fold* (λ*x prod*. *prod* ⊗ *x*) (*map* (λ*x*. **g** $\hat{}_{G_p}$ *f x*) *xs*) (**g** $\hat{}_{G_p}$ (*z* + *f a*))
by (*simp add*: *mod-ring-pow-mult-*G_p)
also have ... = **g** $\hat{}_{G_p}$ (*z* + *f a*) ⊗ **g** $\hat{}_{G_p}$ *sum-list* (*map f xs*)
using *Cons.IH* **by** *blast*
also have ... = **g** $\hat{}_{G_p}$ *z* ⊗ **g** $\hat{}_{G_p}$(*f a*) ⊗ **g** $\hat{}_{G_p}$ *sum-list* (*map f xs*)
by (*simp add*: *mod-ring-pow-mult-*G_p)
also have ... = **g** $\hat{}_{G_p}$ *z* ⊗ **g** $\hat{}_{G_p}$ *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-*α*-is-*φ*-of-*α:
assumes *dist*: *distinct* (*map fst xs-ys*)
and *length-xs-ys*: *length xs-ys* ≤ *max-deg+1*
shows *interpolate-on* (*map* (λ(*x*,*y*).(*x*,$\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *y*)) *xs-ys*) α
= **g** $\hat{}_{G_p}$ (*poly* (*lagrange-interpolation-poly xs-ys*) α)
proof −
have *interpolate-on* (*map* (λ(*x*,*y*).(*x*,$\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *y*)) *xs-ys*) α =
(*let xs* = *map fst* (*map* (λ(*x*, *y*). (*x*, **g** $\hat{}_{G_p}$ *y*)) *xs-ys*);

```
      lagrange-exp =
        map (λ(xj, y). (g ^_{G_p} y) ^_{G_p} poly (lagrange-basis-poly xs xj) α) xs-ys
    in fold (λx prod. prod ⊗ 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 ^_{G_p} y)) xs-ys);
      lagrange-exp =
        map (λ(xj, y). g ^_{G_p} (y * poly (lagrange-basis-poly xs xj) α)) xs-ys
    in fold (λx prod. prod ⊗ x) lagrange-exp 1)
    using mod-ring-pow-pow-G_p by presburger
  also have ... = g ^_{G_p}
  (∑(xj,
    y)←xs-ys. y * poly (lagrange-basis-poly
                        (map fst (map (λ(x, y). (x, g ^_{G_p} y)) xs-ys)) xj)
                    α)
    using fold-on-G_p-is-sum-list[of (λ(xj, y). (y *
            poly
             (lagrange-basis-poly (map fst (map (λ(x, y). (x, g ^_{G_p} y)) xs-ys))
              xj)
             α)) xs-ys 0]
    unfolding Let-def split-def by force
  also have ... = g ^_{G_p} (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 ∧ distinct I
  and α ∉ set I
shows interpolate-on (zip (PickDistinct I # I) (map ((^_{G_p}) g) evals)) α
  = g ^_{G_p} (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 ≤ 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_{G_p} ^_{G_p} y)) xs-ys) α
 = g ^_{G_p} (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
```

Finnaly, conclude the statement that interpolate_on is equivalent to computing Commit. This is what the prior 3 lemmas where building up to.

```
lemma interpolate-on-is-Commit:
  assumes dist: distinct (map fst xs-ys)
```

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

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

```
    proof −
      fix x y
      assume asm: (x, y) ∈ set (tl xs-ys)
      let ?φ = lagrange-interpolation-poly xs-ys

      have y: poly (lagrange-interpolation-poly xs-ys) x = y
          by (rule lagrange-interpolation-poly[OF assms(1)])(simp add: asm calcu-
lation in-set-tlD)+

      have deg-ψ-le: degree (ψ-of (?φ) x) ≤ max-deg
      by (meson degree-lagrange-interpolation-poly degree-q-le-φ le-diff-conv le-trans
            length-xs-ys)

      have α-neg-x: α ≠ 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 ψ-unfold: poly (ψ-of ?φ x) α = (poly ?φ α − poly ?φ x) / (α − x)
        by (simp add: φx-m-φu-eq-xmu-ψx)

    show Eval (map (λt. g ^α ^t) [0..<max-deg + 1]) () (lagrange-interpolation-poly
xs-ys) x
    =  (y, (g ^ poly (lagrange-interpolation-poly xs-ys) α ⊗ inv (g ^ y)) ^
              (1 / (α − 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-ψ-le])
      apply (simp only: ψ-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 length I = max-deg ∧ distinct I
  and α ∉ set I
  and evals
      ∈ set-spmf
          (map-spmf (map (λx. of-int-mod-ring (int x)))
            (sample-uniform-list (max-deg + 1) (order G_p)))
shows map (λi. (Eval (map (λt. g ^_{G_p} (α ^t)) [0..<max-deg + 1]) () (lagrange-interpolation-poly
(zip (PickDistinct I # I) evals)) i)) I
```

$=$ *map2* ($\lambda x\ y.$ (y,($\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ (*poly* (*lagrange-interpolation-poly* (*zip* (*PickDistinct* I # I) *evals*)) α) $\div_{G_p}$ $\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ y) $\widehat{}_{G_p}$ (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*: $\alpha \notin$ *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 ¬ $\alpha \notin$ *set* (*map fst* (*tl xs-ys*))
then have $\alpha \in$ *set* (*map fst* (*tl xs-ys*)) **by** *linarith*
then obtain y **where** y: (α,y) $\in$ *set* (*tl xs-ys*) **by** *fastforce*
then have $\alpha \in$ *set I* **by** (*simp only*: *tl-move set-zip-leftD*)
then show *False* **using** *assms*(2) **by** *blast*
qed

ultimately have *map* ($\lambda i.$ (*Eval* (*map* ($\lambda t.$ $\mathbf{g}$ $\widehat{}_{G_p}$ (α $\widehat{}$ t)) [0..<*max-deg* + 1]) () (*lagrange-interpolation-poly xs-ys*) i)) (*map fst* (*tl xs-ys*))
$=$ *map* (λ(x,y). (y,($\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ (*poly* (*lagrange-interpolation-poly xs-ys*) α) $\div_{G_p}$ $\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ y) $\widehat{}_{G_p}$ (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: $x \in$ {0..<*CARD*(*'e*)}
assume y: $y \in$ {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 probabillity 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_asert - game2_wo_assert

The proof is structered into 5 steps, each step targets one subgoal, except the fifth one, where the trarget 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 probabillity 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* ⇒
(*'e mod-ring*, *'a vk*, *'a commit*, *'e argument*, *'e evaluation*, *'a witness*, *'state*)
eval-hiding-adversary2 ⇒ *bool spmf* **where**
game1 A1 A2 = TRY do {
(α, *PK*) ← *Setup*;
(*I*,σ) ← *A1 PK*;
- :: *unit* ← *assert-spmf* (*length I = max-deg* ∧ *distinct I*);
let i = PickDistinct I;
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*::*nat* ⇒ *'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 (λ*i*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ *i*) *evals*;
let C = interpolate-on (*zip* (*i*#*I*) *exp-evals*) α;
let W = map (λ*j*. *Eval PK* () φ *j*) *I*;
φ' ← *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* ⇒
(*'e mod-ring*, *'a vk*, *'a commit*, *'e argument*, *'e evaluation*, *'a witness*, *'state*)
eval-hiding-adversary2 ⇒ *bool spmf* **where**
game2 $\mathcal{A}1$ $\mathcal{A}2$ = *TRY do* {
(α, *PK*) ← *Setup*;
(*I*,σ) ← $\mathcal{A}1$ *PK*;
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*)) (*sample-uniform-list* (*max-deg*+1) (*order* G_p));
let φ = *lagrange-interpolation-poly* (*zip* (*i*#*I*) *evals*);
let exp-evals = *map* ($\lambda i.$ $\mathbf{g}$ ^ *i*) *evals*;
let C = *interpolate-on* (*zip* (*i*#*I*) *exp-evals*) α;
let witn-tupel = *map* ($\lambda(x,y).$ (*y*,(*C* ÷$_{G_p}$ $\mathbf{g}_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (1/(α−*x*)))) (*zip I* (*tl evals*));
φ' ← $\mathcal{A}2$ *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* ⇒

(*'e mod-ring*, *'a vk*, *'a commit*, *'e argument*, *'e evaluation*, *'a witness*, *'state*)
eval-hiding-adversary2 ⇒ *bool spmf* **where**
game2-w-assert $\mathcal{A}1$ $\mathcal{A}2$ = *TRY do* {
(α, *PK*) ← *Setup*;
(*I*,σ) ← $\mathcal{A}1$ *PK*;
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *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* ($\lambda i.$ $\mathbf{g}$ ^ *i*) *evals*)) α;
let witn-tupel = *map* ($\lambda(x,y).$ (*y*,(*C* ÷$_{G_p}$ $\mathbf{g}_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (1/(α−*x*)))) (*zip I* (*tl evals*));
φ' ← $\mathcal{A}2$ *PK* σ *C I witn-tupel*;
-::*unit* ← *assert-spmf* ($\varphi = \varphi'$);
return-spmf (*hd evals* = *poly* φ' *i*)} *ELSE return-spmf False*

game2_w_asssert without the assert.

definition *game2-wo-assert* :: (*'a vk*, *'e argument*, *'state*) *eval-hiding-adversary1* ⇒
(*'e mod-ring*, *'a vk*, *'a commit*, *'e argument*, *'e evaluation*, *'a witness*, *'state*)
eval-hiding-adversary2 ⇒ *bool spmf* **where**
game2-wo-assert $\mathcal{A}1$ $\mathcal{A}2$ = *TRY do* {
(α, *PK*) ← *Setup*;
(*I*,σ) ← $\mathcal{A}1$ *PK*;

```
  let i = PickDistinct I;
  evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list (max-deg+1)
(order G_p));
  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 witn-tupel = map (λ(x,y). (y,(C ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (1/(α−x)))) (zip I
(tl evals));
  φ' ← A2 PK σ C I witn-tupel;
  return-spmf (hd evals = poly φ' i)} ELSE return-spmf False
```

lemma *PickDistinct-Prop*: *length a = max-deg ∧ distinct a*
⟹ 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*
⟹ distinct (PickDistinct a#a) ∧ length (PickDistinct a#a) = max-deg + 1
using *PickDistinct-Prop* **by** *presburger*

lemma *distinct-map-fst-zip*: *distinct a ⟹ 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
G_p));
let PK = map (λt. g_{G_p} ^_{G_p} (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
G_p));
  let PK = map (λt. g_{G_p} ^_{G_p} (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. g_{G_p} ^_{G_p} 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
```

del: *PickDistinct.simps*)
finally show *?thesis* **.**
qed

lemma *lossless-mod-ring-list*: *lossless-spmf* (*map-spmf* (*map* (λ*x*. *of-int-mod-ring* (*int x*)::′*e mod-ring*))
(*sample-uniform-list* (*max-deg* + *1*) (*order* G_p)))
by (*simp add*: G_p.*order-gt-0*)

Step 2 of the proof:

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

Step 3 of the proof:

lemma *game2-le-DL*: *spmf* (*game2* $\mathcal{A}$*1* $\mathcal{A}$*2*) *True* ≤ *DL-*G_p.*advantage* (*reduction* $\mathcal{A}$*1* $\mathcal{A}$*2*)
(**is** *?lhs* ≤ *?rhs*)
proof −
have *?lhs* = *spmf* (*TRY do* {
(α, *PK*) ← *Setup*;
TRY do{
(*I*,σ) ← $\mathcal{A}$*1 PK*;
TRY do{
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
TRY do{
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*)) (*sample-uniform-list* (*max-deg*+*1*) (*order* G_p));
let φ = *lagrange-interpolation-poly* (*zip* (*i*#*I*) *evals*);
let exp-evals = *map* (λ*i*. **g** ^ *i*) *evals*;
let C = *interpolate-on* (*zip* (*i*#*I*) *exp-evals*) α;
let witn-tupel = *map* (λ(*x*,*y*). (*y*,(*C* ÷$_{G_p}$ **g**$_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (*1*/(α−*x*)))) (*zip I* (*tl evals*));
TRY do{
φ′ ← $\mathcal{A}$*2 PK* σ *C I witn-tupel*;
TRY do{
- :: *unit* ← *assert-spmf* (φ = φ′);
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 *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* {
(α, *PK*) ← *Setup*;
(*I*,σ) ← $\mathcal{A}$*1 PK*;
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*)) (*sample-uniform-list* (*max-deg*+*1*) (*order* G_p));
let φ = *lagrange-interpolation-poly* (*zip* (*i*#*I*) *evals*);
let exp-evals = *map* (λ*i*. **g** ^ *i*) *evals*;
let C = *interpolate-on* (*zip* (*i*#*I*) *exp-evals*) α;
let witn-tupel = *map* (λ(*x*,*y*). (*y*,(*C* $\div_{G_p}$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *y*) $\hat{}_{G_p}$ (*1*/(α−*x*)))) (*zip I* (*tl evals*));
φ' ← $\mathcal{A}$*2 PK* σ *C I witn-tupel*;
- :: *unit* ← *assert-spmf* (φ = φ');
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* φ' (*PickDistinct I*), which is asserted by the Dl reduction game. Hence we add *hd evals* = *poly* φ' (*PickDistinct I*) to the assert without changing the game.

also have ... = *spmf* (*TRY do* {
(α, *PK*) ← *Setup*;
(*I*,σ) ← $\mathcal{A}$*1 PK*;
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*)) (*sample-uniform-list* (*max-deg*+*1*) (*order* G_p));
let φ = *lagrange-interpolation-poly* (*zip* (*i*#*I*) *evals*);
let exp-evals = *map* (λ*i*. **g** ^ *i*) *evals*;
let C = *interpolate-on* (*zip* (*i*#*I*) *exp-evals*) α;
let witn-tupel = *map* (λ(*x*,*y*). (*y*,(*C* $\div_{G_p}$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *y*) $\hat{}_{G_p}$ (*1*/(α−*x*)))) (*zip I* (*tl evals*));
φ' ← $\mathcal{A}$*2 PK* σ *C I witn-tupel*;
- :: *unit* ← *assert-spmf* (φ = φ' ∧ *hd evals* = *poly* φ' *i*);
return-spmf True
} *ELSE return-spmf False*) *True*
proof −
have *equi*: $\bigwedge$*I evals* φ'. *length I* = *max-deg* ∧ *distinct I*
⟹ *evals* ∈ *set-spmf* (*map-spmf* (*map* (*of-int-mod-ring* ∘ *int*))

```
      (sample-uniform-list (max-deg+1) (order G_p)))
    ⟹ ((lagrange-interpolation-poly (zip (PickDistinct I#I) evals) = φ'
    ⟷ (lagrange-interpolation-poly (zip (PickDistinct I#I) evals)) = φ'
        ∧ hd evals = poly φ' (PickDistinct I)))
  proof −
    fix I evals :: 'e mod-ring list
    fix φ'
    assume asm1: length I = max-deg ∧ distinct I
     assume asm2: evals ∈ set-spmf (map-spmf (map  (of-int-mod-ring ∘ 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) = φ'
   ⟷ lagrange-interpolation-poly (zip (PickDistinct I#I) evals) = φ' ∧ hd evals
= poly φ' (PickDistinct I))
    proof
      show lagrange-interpolation-poly (zip (PickDistinct I#I) evals) = φ' ⟹
lagrange-interpolation-poly (zip (PickDistinct I#I) evals) = φ' ∧ hd evals = poly
φ' (PickDistinct I)
      proof
       assume asm: lagrange-interpolation-poly (zip (PickDistinct I#I) evals) =
φ'
       show hd evals = poly φ' (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 φ' = lagrange-interpolation-poly (zip (PickDistinct I#I) evals)
          using asm[symmetric] .
        show (PickDistinct I, hd evals) ∈ 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 $\varphi = \varphi' \wedge hd\ evals = poly\ \varphi'\ (PickDistinct\ I)$ 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* {
(α, *PK*) ← *Setup*;
(*I*,σ) ← $\mathcal{A}1$ *PK*;
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*)) (*sample-uniform-list* (*max-deg*+1) (*order* G_p));
let φ = *lagrange-interpolation-poly* (*zip* (*i*#*I*) *evals*);
let exp-evals = *map* (λ*i*. **g** ^ *i*) *evals*;
let C = *interpolate-on* (*zip* (*i*#*I*) *exp-evals*) α;
let witn-tupel = *map* (λ(*x*,*y*). (*y*,(*C* $\div_{G_p}$ **g**$_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (1/(α−*x*)))) (*zip I* (*tl evals*));
φ' ← $\mathcal{A}2$ *PK* σ *C I witn-tupel*;
- :: *unit* ← *assert-spmf* (φ = φ');
- :: *unit* ← *assert-spmf* (*hd evals* = *poly* φ' *i*);
return-spmf True
} *ELSE return-spmf False*) *True*
by (*simp only*: *assert-collapse*)
also have ... = *spmf* (*TRY do* {
(α, *PK*) ← *Setup*;
TRY do {
(*I*,σ) ← $\mathcal{A}1$ *PK*;
TRY do {
- :: *unit* ← *assert-spmf* (*length I* = *max-deg* ∧ *distinct I*);
let i = *PickDistinct I*;
TRY do {
evals ← *map-spmf* (*map* (*of-int-mod-ring* ∘ *int*)) (*sample-uniform-list* (*max-deg*+1) (*order* G_p));
let φ = *lagrange-interpolation-poly* (*zip* (*i*#*I*) *evals*);
let exp-evals = *map* (λ*i*. **g** ^ *i*) *evals*;
let C = *interpolate-on* (*zip* (*i*#*I*) *exp-evals*) α;
let witn-tupel = *map* (λ(*x*,*y*). (*y*,(*C* $\div_{G_p}$ **g**$_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (1/(α−*x*)))) (*zip I* (*tl evals*));
TRY do {
φ' ← $\mathcal{A}2$ *PK* σ *C I witn-tupel*;
TRY do {
- :: *unit* ← *assert-spmf* (φ = φ');
TRY do {
return-spmf (*hd evals* = *poly* φ' *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) 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 {
  (α, PK) ← Setup;
  (I,σ) ← 𝒜1 PK;
  - :: unit ← assert-spmf (length I = max-deg ∧ distinct I);
  let i = PickDistinct I;
  evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list (max-deg+1)
(order G_p));
  let φ = lagrange-interpolation-poly (zip (i#I) evals);
  let exp-evals = map (λi. g ^ i) evals;
  let C = interpolate-on (zip (i#I) exp-evals) α;
  let witn-tupel = map (λ(x,y). (y,(C ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (1/(α−x)))) (zip I
(tl evals));
  φ' ← 𝒜2 PK σ C I witn-tupel;
  - :: unit ← assert-spmf (φ = φ');
  return-spmf (hd evals = poly φ' 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 ... ≤ spmf (TRY do {
  (α, PK) ← Setup;
  (I,σ) ← 𝒜1 PK;
  let i = PickDistinct I;
  evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list (max-deg+1)
(order G_p));
  let φ = lagrange-interpolation-poly (zip (i#I) evals);
  let exp-evals = map (λi. g ^ i) evals;
  let C = interpolate-on (zip (i#I) exp-evals) α;
  let witn-tupel = map (λ(x,y). (y,(C ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (1/(α−x)))) (zip I
(tl evals));
  φ' ← 𝒜2 PK σ C I witn-tupel;
  - :: unit ← assert-spmf (φ = φ');
  return-spmf (hd evals = poly φ' 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 ... ≤ spmf (TRY do {
```

```
  (α, PK) ← Setup;
  (I,σ) ← A1 PK;
  let i = PickDistinct I;
  evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list (max-deg+1)
(order G_p));
  let exp-evals = map (λi. g ^ i) evals;
  let C = interpolate-on (zip (i#I) exp-evals) α;
  let witn-tupel = map (λ(x,y). (y,(C ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (1/(α−x)))) (zip I
(tl evals));
  φ' ← A2 PK σ C I witn-tupel;
  return-spmf (hd evals = poly φ' 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 {
  (α, PK) ← Setup;
  (I,σ) ← A1 PK;
  let i = PickDistinct I;
  a ← map-spmf (of-int-mod-ring ∘ int) (sample-uniform (order G_p));
  plain-evals ← map-spmf (map (of-int-mod-ring ∘ int)) (sample-uniform-list max-deg
(order G_p));
  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 ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (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
    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 ← map-spmf (of-int-mod-ring ∘ int) (sample-uniform (order G_p));
  (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 G_p));
  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 ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (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 G_p));
  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 G_p));
  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 ÷_{G_p} g_{G_p} ^_{G_p} y) ^_{G_p} (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-G_p.advantage (reduction A1 A2)
    unfolding DL-G_p.advantage-def DL-G_p.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-G_p.advantage (reduction A1 A2) + t-DL-G_p.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-G_p.advantage (reduction-tDL
A1)
    using assms fundamental-lemma-game1-game2 by blast
   also have ... ≤ DL-G_p.advantage (reduction A1 A2) + t-DL-G_p.advantage
```

(*reduction-tDL* $\mathcal{A}1$)
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. $r = \varphi \ mod \ \prod i{\in}B.(x{-}i)$

fun *r* :: *'e argument set* ⇒ *'e mod-ring poly* ⇒*'e batch-evaluation* **where**
r B φ = *do* {
let prod-B = *prod* (λ*i*. [:−*i*,*1*:]) *B*;
φ mod prod-B}

lemma *deg-r*: *degree* (*r B φ*) ≤ *degree φ*
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 ψ_B :: *'e argument set* ⇒ *'e mod-ring poly* ⇒ *'e mod-ring poly* **where**

$\psi_B\ B\ \varphi = do\ \{$
let prod-B = prod $(\lambda i.\ [:-i,1:])\ \ B$;
$(\varphi - (r\ B\ \varphi))$ *div 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* $\Longrightarrow$ *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*: $B \neq \{\} \Longrightarrow degree\ \varphi > card\ B \Longrightarrow degree\ (r\ B\ \varphi) < card\ B$
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 (∏ i∈B. [:− i, 1:]) dvd φ ⟹ φ mod (∏ i∈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 ck ⇒ trapdoor ⇒ 'e mod-ring poly ⇒ 'e argument set
 ⇒ ('e batch-evaluation × 'a witness)
where
 eval-batch PK td φ B =(
   let r = r B φ; — remainder of φ(x)/(∏ i∈B. (x−i)) i.e. φ(x) mod (∏ i∈B.
(x−i))
     ψ = ψ_B B φ; — (φ(x) − r(x)) / (∏ i∈B. (x−i))
     w-B = g-pow-PK-Prod PK ψ — g^ψ(α)
   in
   (r, w-B) — (B, r(x), g^ψ(α))
```

)

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* ⇒ *'a commit* ⇒ *'e argument set* ⇒ (*'e batch-evaluation* × *'a witness*)
⇒ *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* (λ*i*. [:−*i*,*1*:]) *B*);
g-pow-r = *g-pow-PK-Prod PK r-x in*
(*e g-pow-prod-B* w_B ⊗$_{G_T}$ (*e* **g** *g-pow-r*) = *e C* **g**))
— *e*(*g*^(∏ *i*∈*B*. (α−*i*)), *g*^ψ(α)) ⊗ *e*(*g*,*g*^*r*(α)) = *e*(*C*, *g*)

definition *valid-argument-batch* :: *'e argument set* ⇒ *bool*
where *valid-argument-batch B* = (*card B* ≤ *max-deg*)

definition *valid-eval-batch*::(*'e batch-evaluation* × *'a witness*) ⇒ *bool*
where *valid-eval-batch val* = (*let* (*r*,*w*) = *val in degree r* < *max-deg* ∧ *w* ∈ *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** ^$_{G_p}$ *poly* (∏ *i*∈*B*. [:− *i*, *1*:]) α) (**g** ^$_{G_p}$ *poly* (ψ$_B$ *B* φ) α)

$\otimes_{G_T}$ (e **g** (**g** $\hat{}_{G_p}$ *poly* (r B φ) α))
= e (**g** $\hat{}_{G_p}$ *poly* φ α) **g**)
proof −
have (*poly* ($\prod i{\in}B.$ [:− i, *1*:]) α) ∗ (*poly* (ψ_B B φ) α) + *poly* (r B φ) α = *poly* φ α
by (*metis* (*no-types*, *lifting*) ψ_B.*simps add.commute add-diff-cancel-right′ div-poly-eq-0-iff minus-mod-eq-mult-div mod-div-mult-eq nonzero-mult-div-cancel-left poly-hom.hom-add poly-mult r.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 (*intro allI*, *intro impI*)
fix φ::′*e mod-ring poly*
fix B :: ′*e mod-ring set*
assume *deg-*φ: *degree* $\varphi \leq$ *max-deg*
assume *cardB*: *card* $B \leq$ *max-deg*
show *spmf* (*bKZG.correct-eval-game* φ B) *True* = *1*
proof −

show that $g^{\psi_B(\alpha)}$ is correctly computed using a correct public key PK

have *g-pow-*ψB: ∀ x. *g-pow-PK-Prod*
(*map* (λ*t*. **g** $\hat{}_{G_p}$ *of-int-mod-ring* (*int x*) $\hat{}$ *t*) [*0*..<*max-deg* + *1*])
(ψ_B B φ) = **g** $\hat{}_{G_p}$ *poly* (ψ_B B φ) (*of-int-mod-ring* (*int x*))
using *deg-*ψ_B *g-pow-PK-Prod-correct le-trans deg-*φ **by** *blast*

show that $g^{r(\alpha)}$ is correctly computed using a correct public key PK

have *g-pow-rB*: ∀ x. *g-pow-PK-Prod*
(*map* (λ*t*. **g** $\hat{}_{G_p}$ *of-int-mod-ring* (*int x*) $\hat{}$ *t*) [*0*..<*max-deg* + *1*])
(r B φ) = **g** $\hat{}_{G_p}$ *poly* (r B φ) (*of-int-mod-ring* (*int x*))
using *deg-r g-pow-PK-Prod-correct le-trans deg-*φ **by** *blast*

show that $g^{\prod_{i\in B}(\alpha-i)}$ is correctly computed using a correct public key PK

have *g-ow-Prod*: ∀ x. *g-pow-PK-Prod*
(*map* (λ*t*. **g** $\hat{}_{G_p}$ *of-int-mod-ring* (*int x*) $\hat{}$ *t*) [*0*..<*max-deg* + *1*])
($\prod i{\in}B.$ [:− i, *1*:]) = **g** $\hat{}_{G_p}$ *poly* ($\prod i{\in}B.$ [:− i, *1*:]) (*of-int-mod-ring* (*int x*))
using *deg-Prod g-pow-PK-Prod-correct le-trans cardB 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 *spmf* (*bKZG.correct-eval-game* φ B) *True* =
spmf (*do*{
x :: *nat* ← *sample-uniform* (*order* G_p);

let α = *of-int-mod-ring* (*int x*);
let PK = *map* (λt. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ($\alpha \hat{} t$)) [*0..<max-deg+1*];
(*C*,*d*) ← *commit PK* φ;
let W = *eval-batch PK d* φ *B*;
return-spmf (*verify-eval-batch PK C B W*)
}) *True*
unfolding *bKZG.correct-eval-game-def key-gen-def Setup-def*
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 :: *nat* ← *sample-uniform* (*order* G_p);
return-spmf (
e (**g** $\hat{}_{G_p}$ *poly* ($\prod i \in B$. [:− *i*, *1*:]) (*of-int-mod-ring* (*int x*))) (**g** $\hat{}_{G_p}$ *poly* (ψ_B *B* φ) (*of-int-mod-ring* (*int x*)))
$\otimes_{G_T}$ (*e* **g** (**g** $\hat{}_{G_p}$ *poly* (*r B* φ) (*of-int-mod-ring* (*int x*))))
= *e* (**g** $\hat{}_{G_p}$ *poly* φ (*of-int-mod-ring* (*int x*))) **g**)
}) *True*
unfolding *eval-batch-def verify-eval-batch-def commit-def Let-def split-def*
g-pow-PK-Prod-correct[*OF deg-*φ]
using *g-pow-*ψ*B g-pow-rB 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 :: *nat* ← *sample-uniform* (*order* G_p);
return-spmf (*True*
)}) *True*
using *eq-on-e-Batch deg-Prod* **by** *algebra*
also have ... = *spmf* (*scale-spmf* (*weight-spmf* (*sample-uniform* (*Coset.order* G_p))) (*return-spmf True*)) *True*
using *bind-spmf-const*[*of sample-uniform* (*Coset.order* G_p) *return-spmf True*]
by *presburger*
also have ... = *1*
using *weight-sample-uniform-gt-0 CARD-*G_p *p-gr-two* **by** *simp*
finally show *?thesis* **.**
qed
qed

end

end
theory *BatchKZG-poly-bind*

imports *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-DL-*G_p*.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 Gp as preimage group and GT 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*.cyclic-group-axioms* G_p*.cyclic-group-axioms* **by** *presburger*

20.2 Binding for the batched evaluation according to KZG10

type-synonym ('*a*', '*e*') *adversary* =
'*a* *ck* ⇒
('*a* *commit* × '*e* *argument* × '*e* *evaluation* × '*a* *witness* × '*e* *argument set*
× '*a* *witness* × '*e* *batch-evaluation*) *spmf*

This is the formalized evaluation binding game

definition *bind-game* :: ('*a*, '*e*) *adversary* ⇒ *bool spmf*
where *bind-game* $\mathcal{A}$ = *TRY do* {
(*ck*,*vk*) ← *key-gen*;
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) ← $\mathcal{A}$ *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*);
return-spmf (*b* ∧ *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* ⇒ *real*
where *bind-advantage* $\mathcal{A}$ ≡ *spmf* (*bind-game* $\mathcal{A}$) *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* ⇒ ('*a*,'*e*,'*c*) *t-BSDH.adversary*
where
reduction $\mathcal{A}$ *PK* = *do* {
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) ← $\mathcal{A}$ *PK*;
let p' = *g-pow-PK-Prod PK* (*prod* (λ*i*. [:−*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* $\otimes_{G_T}$ *e* (*r'* $\div_{G_p}$ *w-i*) **g**) $\widehat{}_{G_T}$ (*1*/(φ-*i* − *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* ⇒ ('*a*,'*e*,'*c*) *t-BSDH.adversary*
where
ext-reduction $\mathcal{A}$ *PK* = *do* {
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) ← $\mathcal{A}$ *PK*;
- :: *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*)
∧ *verify-eval PK C i* (φ-*i*, *w-i*) ∧ *verify-eval-batch PK C B* (*r-x*, *w-B*));

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* $-$ [:*poly r-x i*:]) *div* [:$-i$,1:]);
return-spmf ($-i$::$'e$ *mod-ring*, (*e p' w-B* $\otimes_{G_T}$ *e* (*r'* $\div_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*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 $\mathcal{A}$ = *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*);
-::*unit* $\leftarrow$ *assert-spmf* (*b* $\wedge$ *b'*);
return-spmf True} *ELSE return-spmf False*
(**is** *?lhs* = *?rhs*)
proof $-$
have *?lhs* = *TRY do* {
(*ck*,*vk*) $\leftarrow$ *key-gen*;
TRY do {
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow$ $\mathcal{A}$ *ck*;
TRY do {
- :: *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*));

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* $\wedge$ *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*) $\leftarrow$ *key-gen*;
TRY do {
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow$ $\mathcal{A}$ *ck*;
TRY do {
- :: *unit* $\leftarrow$ *assert-spmf* (*i* $\in$ *B* $\wedge$ φ-*i* $\neq$ *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);
    -::unit ← assert-spmf (b ∧ 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 ... = ?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 φ-i such that φ-$i \neq r(x)$ 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 ∈ B ∧ φ-i ≠ poly r-x i
    ∧ valid-argument-batch B
    ∧ valid-eval (φ-i, w-i) ∧ valid-eval-batch (r-x, w-B)
    ∧ verify-eval (map (λt. g ^ α ^ t) [0..<max-deg + 1]) C i (φ-i, w-i)
    ∧ verify-eval-batch (map (λt. g ^ α ^ t) [0..<max-deg + 1]) C B (r-x, w-B)
  shows  e g g ^_{G_T} (1 / (α + - i))
        =
          (e (g-pow-PK-Prod (map (λt. g ^ α ^ t) [0..<max-deg + 1])
              ((∏ i∈B. [:- i, 1:]) div [:- i, 1:]))
           w-B
        ⊗_{G_T}
          e (g-pow-PK-Prod (map (λt. g ^ α ^ t) [0..<max-deg + 1])
              ((r-x - [:poly r-x i:]) div [:- i, 1:]) ⊗
             inv w-i)
           g) ^_{G_T}
          (1 / (φ-i - poly r-x i))
  (is ?goal = ?cmpt)
proof -
  obtain b where b: w-B = g ^ 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 = g ^ y
    apply (insert assms)
```

```
    apply (simp only: valid-eval-def Let-def split-def)
    apply (simp split: prod.split)
    by (metis G_p.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:]) 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 = (∏ i∈B. [:- i, 1:]) div [:- i, 1:] by blast
  then have p'-x-p-x: (∏ i∈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 (map (λt. g ^ α ^ t) [0..<max-deg + 1] ! 1 ⊗ inv (g ^
i)) ⊗_{G_T} e g g ^_{G_T} φ-i
    = e (g-pow-PK-Prod (map (λt. g ^ α ^ t) [0..<max-deg + 1]) (∏ i∈B. [:- i,
1:])) w-B ⊗_{G_T}
   e g (g-pow-PK-Prod (map (λt. g ^ α ^ t) [0..<max-deg + 1]) r-x) (is ?lhs =
?rhs)
    unfolding verify-eval-def verify-eval-batch-def Let-def split-def by auto
  moreover have ?lhs = e (g ^ y) (g ^ (α-i) ) ⊗_{G_T} e g g ^_{G_T} φ-i
    unfolding y using PK-i d-pos mod-ring-pow-mult-inv-G_p by auto
  moreover have ?rhs = e (g ^ poly (∏ i∈B. [:- i, 1:]) α) (g ^ b) ⊗_{G_T} e g (g
^ poly r-x α)
  proof -
    have g-pow-PK-Prod (map (λt. g ^ α ^ t) [0..<max-deg + 1]) (∏ i∈B. [:- i,
1:])
        = g ^ poly (∏ i∈B. [:- i, 1:]) α
      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-pow-PK-Prod (map (λt. g ^ α ^ t) [0..<max-deg + 1]) r-x
= g ^ poly r-x α
     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 (g ^ y) (g ^ (α-i) ) ⊗_{G_T} e g g ^_{G_T} φ-i = e (g ^ poly
(∏ i∈B. [:- i, 1:]) α) (g ^ b) ⊗_{G_T} e g (g ^ poly r-x α)
    by argo
  then have y*(α-i) + φ-i = (poly (∏ i∈B. [:- i, 1:]) α)*b + poly r-x α
    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 (∏ i∈B. [:- i, 1:]) α)*b - (α-i)*y = φ-i -  poly r-x α
```

by (*metis* (*no-types*, *lifting*) *add-diff-cancel-left'* *add-diff-cancel-right'* *add-diff-eq* *mult.commute*)
then have (*poly* (∏ *i*∈*B*. [:− *i*, *1*:]) *α*)∗*b* − (*α*−*i*)∗*y* = *φ-i* − (*poly r'-x α*)∗(*α*−*i*) − *poly* [:*poly r-x i*:] *α*
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 (*α*−*i*)∗(*poly p'-x α*)∗*b* − (*α*−*i*)∗*y* = *φ-i* − (*poly r'-x α*)∗(*α*−*i*) − *poly* [:*poly r-x i*:] *α*
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 (*α*−*i*)∗((*poly p'-x α*)∗*b* − *y* + *poly r'-x α*) = *φ-i* − *poly* [:*poly r-x i*:] *α*
by (*simp add*: *Groups.mult-ac*(*2*) *Groups.mult-ac*(*3*) *Rings.ring-distribs*(*1*) *Rings.ring-distribs*(*4*))
then have (*α*−*i*)∗((*poly p'-x α*)∗*b* − *y* + *poly r'-x α*) = *φ-i* − *poly r-x i*
by *auto*
then have *poly-eq-res*: *1*/(*α*−*i*) = ((*poly p'-x α*)∗*b* − *y* + *poly r'-x α*)/(*φ-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 *?cmpt* = (*e* (**g** ^$_{G_p}$ (*poly p'-x α*)) (**g** ^$_{G_p}$ *b*) ⊗$_{G_T}$ *e* (**g** ^$_{G_p}$ (*poly r'-x α*) ÷$_{G_p}$ (**g** ^$_{G_p}$ *y*)) **g**) ^$_{G_T}$ (*1* / (*φ-i* − *poly r-x i*))
proof −
have *g-pow-PK-Prod* (*map* (λ*t*. **g** ^ *α* ^ *t*) [*0*..<*max-deg* + *1*]) ((∏ *i*∈*B*. [:− *i*, *1*:]) *div* [:− *i*, *1*:])
= **g** ^$_{G_p}$ (*poly p'-x α*)
unfolding *p'-x*
by (*rule g-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-pow-PK-Prod* (*map* (λ*t*. **g** ^ *α* ^ *t*) [*0*..<*max-deg* + *1*]) ((*r-x* − [:*poly r-x i*:]) *div* [:− *i*, *1*:])
= **g** ^$_{G_p}$ (*poly r'-x α*)
unfolding *r'-x*
apply (*rule g-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-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 ... = *e* **g** **g** ^$_{G_T}$ (((*poly p'-x α*)∗*b* − *y* + *poly r'-x α*)/(*φ-i* −

poly r-x i))
proof −
have (*e* (**g** ^$_{G_p}$ *poly p'-x α*) (**g** ^$_{G_p}$ *b*) ⊗$_{G_T}$
e (**g** ^$_{G_p}$ *poly r'-x α* ⊗ *inv* (**g** ^$_{G_p}$ *y*)) **g**) ^$_{G_T}$ (*1* / (*φ-i* − *poly r-x i*))
= (*e* (**g** ^$_{G_p}$ *poly p'-x α*) (**g** ^$_{G_p}$ *b*) ⊗$_{G_T}$
e (**g** ^$_{G_p}$ (*poly r'-x α* − *y*)) **g**) ^$_{G_T}$ (*1* / (*φ-i* − *poly r-x i*))
using *mod-ring-pow-mult-inv-G$_p$* **by** *presburger*
also have ... = (*e* **g g** ^$_{G_T}$ ((*poly p'-x α*)∗*b*) ⊗$_{G_T}$
e **g g** ^$_{G_T}$ (*poly r'-x α* − *y*)) ^$_{G_T}$ (*1* / (*φ-i* − *poly r-x i*))
using *e-linear-in-fst G$_p$.generator-closed e-bilinear* **by** *presburger*
also have ... = (*e* **g g** ^$_{G_T}$ ((*poly p'-x α*)∗*b* + *poly r'-x α* − *y*)) ^$_{G_T}$ (*1* / (*φ-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** ^$_{G_T}$ (((*poly p'-x α*)∗*b* + *poly r'-x α* − *y*)/(*φ-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* ∈ (*B*::'*e argument set*) ∧
(*φ-i*:: '*e evaluation*) ≠ (*poly r-x i*:: '*e evaluation*) ∧
valid-argument-batch B ∧
valid-eval (*φ-i*,*w-i*) ∧ *valid-eval-batch* (*r-x*,*w-B*) ∧
verify-eval (*map* (λ*t*. **g**$_{G_p}$ ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*]) *C i* (*φ-i*,*w-i*) ∧
verify-eval-batch (*map* (λ*t*. **g**$_{G_p}$ ^$_{G_p}$ (*α*::'*e mod-ring*) ^ *t*) [*0*..<*max-deg* + *1*]) *C B* (*r-x*,*w-B*) ∧
e **g g** ^$_{G_T}$ (*1* / (*α* + − *i*)) =
(*e* (*g-pow-PK-Prod* (*map* (λ*t*. **g**$_{G_p}$ ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*])
((∏ *i*∈*B*. [:− *i*, *1*:]) *div* [:− *i*, *1*:]))
w-B ⊗$_{G_T}$
e (*g-pow-PK-Prod* (*map* (λ*t*. **g**$_{G_p}$ ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*])
((*r-x* − [:*poly r-x i*:]) *div* [:− *i*, *1*:]) ⊗
inv w-i)
g) ^$_{G_T}$
((*1*::'*e mod-ring*) / (*φ-i* − *poly r-x i*)))
⟷
(*i* ∈ *B* ∧

$\varphi\text{-}i \neq$ *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 (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ α $\widehat{}$ *t*) [*0..<max-deg + 1*]) *C i* (φ-*i*,*w-i*) $\wedge$
verify-eval-batch (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ α $\widehat{}$ *t*) [*0..<max-deg + 1*]) *C B* (*r-x*,*w-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*: *bind-game* $\mathcal{A}$ = *t-BSDH.game* (*ext-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 *?mr* = $\lambda\alpha$. (*of-int-mod-ring* (*int* α)::'*e mod-ring*)
let *?PK* = $\lambda\alpha$. (*map* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\widehat{}_{G_p}$ ((*?mr* α)$\widehat{}$*t*)) [*0..<max-deg+1*])

Firstly, unfold the t-BSDH game and the reduction adversary

have *t-BSDH.game* (*ext-reduction* $\mathcal{A}$) = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
(*c*, *g*) $\leftarrow$ (*ext-reduction* $\mathcal{A}$) (*?PK* α);
-::*unit* $\leftarrow$ *assert-spmf*((*e* **g** **g**) $\widehat{}_{G_T}$ (*1*/((*?mr* α)+*c*)) = *g*);
return-spmf True
} *ELSE return-spmf False*
unfolding *t-BSDH.game-alt-def* **by** (*metis o-def*)
also have ... = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow$ $\mathcal{A}$ (*?PK* α);
- :: *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* α) *C i* (φ-*i*, *w-i*) $\wedge$ *verify-eval-batch* (*?PK* α) *C B* (*r-x*, *w-B*));
let p' = *g-pow-PK-Prod* (*?PK* α) (*prod* ($\lambda i.$ [:−*i*,*1*:]) *B div* [:−*i*,*1*:]);
let r' = *g-pow-PK-Prod* (*?PK* α) ((*r-x* − [:*poly r-x i*:]) *div* [:−*i*,*1*:]);
-::*unit* $\leftarrow$ *assert-spmf* ((*e* **g** **g**) $\widehat{}_{G_T}$ (*1*/((*?mr* α)+(−*i*))) = (*e* p' *w-B* $\otimes_{G_T}$ *e* (r' $\div_{G_p}$ *w-i*) **g**) $\widehat{}_{G_T}$ (*1*/(φ-*i* − *poly r-x i*)));
return-spmf True
} *ELSE return-spmf False*
by *force*

fold the two asserts together so we can reason about their joined content.

also have ... = *TRY do* {
α ← *sample-uniform* (*order* G_p);
TRY do {
(*C*, *i*, *φ-i*, *w-i*, *B*, *w-B*, *r-x*) ← $\mathcal{A}$ (*?PK α*);
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*)
∧ *verify-eval* (*?PK α*) *C i* (*φ-i*, *w-i*) ∧ *verify-eval-batch* (*?PK α*) *C B* (*r-x*, *w-B*));
let p′ = *g-pow-PK-Prod* (*?PK α*) (*prod* (λ*i*. [:−*i*,*1*:]) *B div* [:−*i*,*1*:]);
let r′ = *g-pow-PK-Prod* (*?PK α*) ((*r-x* − [:*poly r-x i*:]) *div* [:−*i*,*1*:]);
-::*unit* ← *assert-spmf* ((*e* **g** **g**) $\hat{}_{G_T}$ (*1*/((*?mr α*)+(−*i*))) = (*e p′ w-B* ⊗$_{G_T}$ *e* (*r′* ÷$_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*1*/(*φ-i* − *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* {
α ← *sample-uniform* (*order* G_p);
TRY do {
(*C*, *i*, *φ-i*, *w-i*, *B*, *w-B*, *r-x*) ← $\mathcal{A}$ (*?PK α*);
TRY do {
let p′ = *g-pow-PK-Prod* (*?PK α*) (*prod* (λ*i*. [:−*i*,*1*:]) *B div* [:−*i*,*1*:]);
let r′ = *g-pow-PK-Prod* (*?PK α*) ((*r-x* − [:*poly r-x i*:]) *div* [:−*i*,*1*:]);
- :: *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*)
∧ *verify-eval* (*?PK α*) *C i* (*φ-i*, *w-i*) ∧ *verify-eval-batch* (*?PK α*) *C B* (*r-x*, *w-B*)
∧ (*e* **g** **g**) $\hat{}_{G_T}$ (*1*/((*?mr α*)+(−*i*)))
= (*e p′ w-B* ⊗$_{G_T}$ *e* (*r′* ÷$_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*1*/(*φ-i* − *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* {
α ← *sample-uniform* (*order* G_p);
(*C*, *i*, *φ-i*, *w-i*, *B*, *w-B*, *r-x*) ← $\mathcal{A}$ (*?PK α*);
let p′ = *g-pow-PK-Prod* (*?PK α*) (*prod* (λ*i*. [:−*i*,*1*:]) *B div* [:−*i*,*1*:]);
let r′ = *g-pow-PK-Prod* (*?PK α*) ((*r-x* − [:*poly r-x i*:]) *div* [:−*i*,*1*:]);
- :: *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*)

$\wedge$ *verify-eval* (*?PK* α) *C i* (φ-*i*, *w-i*) $\wedge$ *verify-eval-batch* (*?PK* α) *C B* (*r-x*, *w-B*)
$\wedge$ (*e* **g** **g**) $\hat{}_{G_T}$ (*1*/((*?mr* α)+(−*i*)))
= (*e p′ w-B* $\otimes_{G_T}$ *e* (*r′* $\div_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*1*/(φ-*i* − *poly r-x i*)));
return-spmf True
} *ELSE return-spmf False*
unfolding *split-def Let-def*
by(*fold try-bind-spmf-lossless2*[*OF lossless-return-spmf*]) *simp*

erase the pairing check with the literal helping lemma to get closer to the eval bind game

also have ... = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow \mathcal{A}$ (*?PK* α);
- :: *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* α) *C i* (φ-*i*, *w-i*) $\wedge$ *verify-eval-batch* (*?PK* α) *C B* (*r-x*, *w-B*));
return-spmf True
} *ELSE return-spmf False*
unfolding *Let-def* **using** *literal-helping* **by** *algebra*

Split the message checks off the Evaluations checks, as it is done in the eval bind game

also have ... = *TRY do* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
TRY do {
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow \mathcal{A}$ (*?PK* α);
TRY do {
- :: *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* α) *C i* (φ-*i*, *w-i*) $\wedge$ *verify-eval-batch* (*?PK* α) *C B* (*r-x*, *w-B*));
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* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
TRY do {
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow \mathcal{A}$ (*?PK* α);
TRY do {
- :: *unit* $\leftarrow$ *assert-spmf* (*i* $\in$ *B* $\wedge$ φ-*i* $\neq$ *poly r-x i*
$\wedge$ *valid-argument-batch B*

```
                    ∧ valid-eval (φ-i, w-i) ∧ valid-eval-batch (r-x, w-B));
    - :: unit ← assert-spmf (verify-eval (?PK α) C i (φ-i, w-i) ∧ verify-eval-batch
(?PK α) 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 {
    α ← sample-uniform (order G_p);
    (C, i, φ-i, w-i, B, w-B, r-x) ← A (?PK α);
    - :: 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));
    - :: unit ← assert-spmf (verify-eval (?PK α) C i (φ-i, w-i) ∧ verify-eval-batch
(?PK α) 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 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: bind-advantage A = t-BSDH.advantage (ext-reduction
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: spmf (t-BSDH.game (ext-reduction A)) True
  ≤ spmf (t-BSDH.game (reduction A)) True
  (is spmf ?lhgame True ≤ 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 ?α = λα. (of-int-mod-ring (int α)::'e mod-ring)
  let ?PK = λα. (map (λt. g_{G_p} ^_{G_p} ((?α α)^t)) [0..<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* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
(C, i, φ-i, *w-i*, B, *w-B*, *r-x*) $\leftarrow \mathcal{A}$ (*?PK* α);
- :: *unit* $\leftarrow$ *assert-spmf* ($i \in B \wedge \varphi$-$i \neq$ *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* α) C i (φ-i, *w-i*) $\wedge$ *verify-eval-batch* (*?PK* α) C B (*r-x*, *w-B*));
let p' = *g-pow-PK-Prod* (*?PK* α) (*prod* (λi. [:−i,*1*:]) B *div* [:−i,*1*:]);
let r' = *g-pow-PK-Prod* (*?PK* α) ((*r-x* − [:*poly r-x i*:]) *div* [:−i,*1*:]);
-::*unit* $\leftarrow$ *assert-spmf* ((*e* **g** **g**) $\hat{}_{G_T}$ (*1*/((*?*α α)+(−i))) = (*e* p' *w-B* $\otimes_{G_T}$ *e* ($r' \div_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*1*/(φ-i − *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* {
$\alpha \leftarrow$ *sample-uniform* (*order* G_p);
(C, i, φ-i, *w-i*, B, *w-B*, *r-x*) $\leftarrow \mathcal{A}$ (*?PK* α);
let p' = *g-pow-PK-Prod* (*?PK* α) (*prod* (λi. [:−i,*1*:]) B *div* [:−i,*1*:]);
let r' = *g-pow-PK-Prod* (*?PK* α) ((*r-x* − [:*poly r-x i*:]) *div* [:−i,*1*:]);
-::*unit* $\leftarrow$ *assert-spmf* ((*e* **g** **g**) $\hat{}_{G_T}$ (*1*/((*?*α α)+(−i))) = (*e* p' *w-B* $\otimes_{G_T}$ *e* ($r' \div_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*1*/(φ-i − *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*)
$\leq$ *ennreal* (*spmf* (*t-BSDH.game* (*reduction* $\mathcal{A}$)) *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.emeasure-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* $\mathcal{A} \leq$ *t-BSDH.advantage* (*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* × *'state'*) ⇒ (*'e' mod-ring* × *'e' argument set* × (*'e' batch-evaluation* × (*'a' witness* × *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* =

$('a'\ commit \times int\ list) \Rightarrow$
$('e'\ mod\text{-}ring\ poly \times unit)\ spmf$

restrict for AGM adversaries 1 and 2

interpretation *AGM1*: *Algebraic-Algorithm* G_p *listS* G_p*.groupS prodC* G_p*.groupC noConstrain*
by (*unfold-locales*)

$'ck' \Rightarrow 'state' \Rightarrow ('argument' \times ('evaluation' \times 'witness'))\ spmf$

interpretation *AGM2*: *Algebraic-Algorithm* G_p *prodS* (*listS* G_p*.groupS*) *noSelect*

prodC noConstrain (*prodC noConstrain* (*prodC noConstrain* G_p*.groupC*))
by (*unfold-locales*)

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

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

definition *ks-to-eval-bind-reduction* :: ('*state*, '*a*) *AGM-knowledge-soundness-adversary1*

$\Rightarrow$ ('*e*, '*state*, '*a*) *knowledge-soundness-adversary2-AGM* $\Rightarrow$ ('*a*, '*e*) *extractor*
$\Rightarrow$ ('*a*, '*e*) *adversary* **where**
ks-to-eval-bind-reduction $\mathcal{A}1\ \mathcal{A}2\ \mathcal{E}$ *ck* = *do* {
((*c*,*cvec*),σ) $\leftarrow$ *AGM1.restrict* $\mathcal{A}1$ *ck*;
(*p*,*td*) $\leftarrow$ $\mathcal{E}$ (*c*,*cvec*);
(*i*, *I*, (*r-x*, (*w*, *wvec*))) $\leftarrow$ *AGM2.restrict* $\mathcal{A}2$ (*ck*,σ);
let (*p-i*,*w'*) = *Eval ck td p i*;
return-spmf (*c*,*i*, *p-i*, *w'*, *I*, *w*, *r-x*)
}

Extractor definition

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

lemma *reduction-map-imp*:
let ck = *map* ($\lambda t.$ $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^t)) [*0*..<*max-deg+1*];
vk = *map* ($\lambda t.$ $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^t)) [*0*..<*max-deg+1*];
(*p-i*,*w-i*) = *Eval ck td* (*Poly* (*map* (*of-int-mod-ring*::*int* $\Rightarrow$'*e mod-ring*) *cvec*)) *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. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1]
  define vk where vk-def: vk = map (λt. g_{G_p} ^_{G_p} (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)
        = g_{G_p} ^_{G_p} (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-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*)) ≤ *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 le-diff-conv*)

have *1*: (*g-pow-PK-Prod* (*map* (*λt*. **g** ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*])
(*ψ-of* (*Poly ?cvec*) *i*))
= (**g** ^$_{G_p}$ *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** ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*] ! *1* = **g** ^$_{G_p}$ *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** ^$_{G_p}$ *poly* (*Poly ?cvec*) *α*) = *c*
proof −
have (**g** ^$_{G_p}$ *poly* (*Poly ?cvec*) *α*)
= *g-pow-PK-Prod* (*map* (*λt*. **g** ^$_{G_p}$ *α* ^ *t*) [*0*..<*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* (*λi acc*. *acc* ⊗$_{G_p}$ (**g**$_{G_p}$ ^$_{G_p}$ (*α*^*i*))

$\hat{}_{G_p}$ (*poly.coeff* (*Poly ?cvec*) *i*))
[*0..<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* (λ *i acc. acc* $\otimes_{G_p}$ ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ($\alpha\hat{}$*i*)) $\hat{}_{G_p}$ (*?cvec*!*i*)) [*0..<max-deg+1*] $\mathbf{1}_{G_p}$
proof −
have *fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?cvec* ! *i*) [*0..<max-deg* + *1*] **1**
= *fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?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* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?cvec* ! *i*)
[*Suc* (*degree* (*Poly ?cvec*))*..<max-deg* + *1*]
(*fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?cvec* ! *i*)
[*0..<Suc* (*degree* (*Poly ?cvec*))] **1**)
by *fastforce*
also have ... = *fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *poly.coeff* (*Poly ?cvec*) *i*)
[*0..<Suc* (*degree* (*Poly ?cvec*))]
1
proof −
have *fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *?cvec* ! *i*) [*0..<Suc* (*degree* (*Poly ?cvec*))] **1**
= *fold* (λ*i acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *i*) $\hat{}_{G_p}$ *poly.coeff* (*Poly ?cvec*) *i*) [*0..<Suc* (*degree* (*Poly ?cvec*))] **1**
proof (*rule List.fold-cong*)
show $\bigwedge$*x*. *x* ∈ *set* [*0..<Suc* (*degree* (*Poly ?cvec*))] ⟹
(λ*acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *?cvec* ! *x*) =
(λ*acc. acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *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 coeff-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* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *?cvec* ! *x* = *acc* $\otimes$ (**g** $\hat{}_{G_p}$ α $\hat{}$ *x*) $\hat{}_{G_p}$ *poly.coeff* (*Poly ?cvec*) *x*
by *presburger*

```
    qed
  qed simp+
  moreover have ∀ init ∈ carrier G_p.
        fold (λi acc. acc ⊗ (g ^_G_p α ^ i) ^_G_p ?cvec ! i)
          [Suc (degree (Poly ?cvec))..<max-deg + 1]
          init
        = init
  proof
    fix init ::'a
    assume init-in-carrier: init ∈ carrier G_p
    have fold (λi acc. acc ⊗ (g ^_G_p α ^ i) ^_G_p ?cvec ! i)
          [Suc (degree (Poly ?cvec))..<max-deg + 1]
          init = fold (λi acc. acc ⊗ 1)
          [Suc (degree (Poly ?cvec))..<max-deg + 1]
          init
    proof (rule List.fold-cong)
      show ⋀x. x ∈ set [Suc (degree (Poly ?cvec))..<max-deg + 1] ⟹
          (λacc. acc ⊗ (g ^_G_p α ^ x) ^_G_p ?cvec ! x) = (λacc. acc ⊗ 1)
      proof
        fix x::nat
        fix acc ::'a
        assume asm: x ∈ set [Suc (degree (Poly ?cvec))..<max-deg + 1]
        show acc ⊗ (g ^_G_p α ^ x) ^_G_p ?cvec ! x = acc ⊗ 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 (g ^_G_p α ^ x) ^_G_p ?cvec ! x = 1 by simp
          then show ?thesis by argo
        qed
      qed
    qed simp+
    also have ... = init
    proof (induction max-deg)
      case 0
      then show ?case by fastforce
    next
      case (Suc max-deg)
    have fold (λi acc. acc ⊗ 1) [Suc (degree (Poly ?cvec))..<Suc max-deg
+ 1] init
        = fold (λi acc. acc ⊗ 1) ([Suc (degree (Poly ?cvec))..<max-deg +
1] @ [Suc max-deg]) init
        by (simp add: init-in-carrier)
      also have ... = fold (λi acc. acc ⊗ 1) [Suc max-deg] (fold (λi acc.
acc ⊗ 1) [Suc (degree (Poly ?cvec))..<max-deg + 1] init)
        by force
        also have ... = fold (λi acc. acc ⊗ 1) [Suc max-deg] init using
Suc.IH by argo
```

```
      also have ... = init ⊗ 1 by force
      also have ... = init by (simp add: init-in-carrier)
      finally show ?case .
    qed
    finally show fold (λi acc. acc ⊗ (g ^_{G_p} α ^ i) ^_{G_p} ?cvec ! i)
          [Suc (degree (Poly ?cvec))..<max-deg + 1]
          init
        = init .
  qed
  ultimately show ?thesis
   by (metis (no-types, lifting) G_p.generator-closed G_p.int-pow-closed ‹g
^_{G_p} poly (Poly ?cvec) α = g-pow-PK-Prod (map (λt. g ^_{G_p} α ^ 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 ⊗_{G_p} (map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1])!i
^_{G_p} (?cvec!i)) [0..<max-deg+1] 1_{G_p}
proof(rule List.fold-cong)
 show 1 = 1 by simp
 show [0..<max-deg + 1] = [0..<max-deg + 1] by simp
 show ⋀x. x ∈ set [0..<max-deg + 1] ⟹
  (λacc. acc ⊗ (g ^_{G_p} α ^ x) ^_{G_p} ?cvec ! x) =
  (λacc. acc ⊗ map (λt. g ^_{G_p} α ^ t) [0..<max-deg + 1] ! x ^_{G_p} ?cvec !
x)
 proof
  fix x::nat
  fix acc :: 'a
  assume asm: x ∈ set [0..<max-deg + 1]
  show  acc ⊗ (g ^_{G_p} α ^ x) ^_{G_p} ?cvec ! x
     = acc ⊗ map (λt. g ^_{G_p} α ^ t) [0..<max-deg + 1] ! x ^_{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 (λ i acc. acc ⊗_{G_p} (map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1])!i
^_{G_p} (of-int-mod-ring (cvec!i))) [0..<max-deg+1] 1_{G_p}
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 ^ α ^ t) [0..<max-deg + 1] ! x ^
map of-int-mod-ring cvec ! x) =
 (λacc. acc ⊗ map (λt. g ^ α ^ t) [0..<max-deg + 1] ! 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 G_p ⟹  g [^]_{G_p}
(to-int-mod-ring (of-int-mod-ring x::'e mod-ring)) = g [^]_{G_p} x
           proof −
             fix g x
             assume g-in-carrier: g ∈ carrier G_p
            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 [^]_{G_p} (to-int-mod-ring (of-int-mod-ring x::'e mod-ring))
= g [^]_{G_p} x
              using carrier-pow-mod-order-G_p 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) G_p.generator-closed G_p.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*
≤ *spmf* (*bind-game* (*ks-to-eval-bind-reduction* $\mathcal{A}1$ $\mathcal{A}2$ *E*)) *True*
(**is** *?lhs* ≤ *?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*) ← *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*, *I*, (*r*, (*w*, *wvec*))) ← *AGM2.restrict* $\mathcal{A}2$ (*ck*,σ);
let (*p-i*,*w*′) = *Eval ck td p i*;
return-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*)
} *ELSE return-spmf False*) *True*
unfolding *knowledge-soundness-game-AGM-def 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
by *force*
also have ...= *spmf* (*TRY do* {
(*ck*,*vk*) ← *key-gen*;
TRY do {
((*c*,*cvec*),σ) ← $\mathcal{A}1$ *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* $\mathcal{A}2$ (*ck*,σ);
let (*p-i*,*w*′) = *Eval ck td p i*;
TRY do {
return-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*)
} *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*) ← *key-gen*;
TRY do {
((*c*,*cvec*),σ) ← $\mathcal{A}1$ *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* $\mathcal{A}2$ (*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) ← $\mathcal{A}1$ *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*, *I*, (*r*, (*w*, *wvec*))) ← *AGM2.restrict* $\mathcal{A}2$ (*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) ← $\mathcal{A}1$ *ck*;
(*p*,*td*) ← *E* (*c*,*cvec*);
(*i*, *I*, (*r*, (*w*, *wvec*))) ← *AGM2.restrict* $\mathcal{A}2$ (*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 G_p);
  let (α::'e mod-ring) = of-int-mod-ring (int x);
  let ck = map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1];
  let vk = map (λt. g_{G_p} ^_{G_p} (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 G_p);
  let (α::'e mod-ring) = of-int-mod-ring (int x);
  let ck = map (λt. g_{G_p} ^_{G_p} (α^t)) [0..<max-deg+1];
  let vk = map (λt. g_{G_p} ^_{G_p} (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)
```

```
                ∧ 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) 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;
      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 {
            return-spmf (verify-eval vk c i (p-i, w-i)
              ∧ 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) ← 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));
    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
```

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*) ← $\mathcal{A}1$ *ck*;
(*p*,*td*) ← *E* (*c*,*cvec*);
(*i*, *I*, (*r-x*, (*w*, *wvec*))) ← *AGM2.restrict* $\mathcal{A}2$ (*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*) ← $\mathcal{A}1$ *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*, *I*, (*r-x*, (*w*, *wvec*))) ← *AGM2.restrict* $\mathcal{A}2$ (*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* $\mathcal{A}1$ $\mathcal{A}2$ *E*)) *True*
unfolding *bind-game-def ks-to-eval-bind-reduction-def*
unfolding *knowledge-soundness-game-AGM-def 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 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.