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 $e\ (P\ [\widehat{}]_{G_p}\ (a{::}\mathit{int}))\ Q = (e\ P\ Q)\ [\widehat{}]_{G_T}\ a$
⟨*proof*⟩

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 $e\ P\ (Q\ [\widehat{}]_{G_p}\ (a{::}\mathit{int})) = (e\ P\ Q)\ [\widehat{}]_{G_T}\ a$
⟨*proof*⟩

lemma *addition-in-exponents-on-e*[*simp*]:
assumes $x \in \mathit{carrier}\ G_p \wedge y \in \mathit{carrier}\ G_p$
shows $(e\ x\ y)\ [\widehat{}]_{G_T}\ (a{::}\mathit{int}) \otimes_{G_T} (e\ x\ y)\ [\widehat{}]_{G_T}\ (b{::}\mathit{int}) = (e\ x\ y)\ [\widehat{}]_{G_T}\ (a{+}b)$
⟨*proof*⟩

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*⟩

lemma *e-g-g-in-carrier-GT*[*simp*]: $e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p} \in \mathit{carrier}\ G_T$
⟨*proof*⟩

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

lemma *pow-on-eq-card-GT*[*simp*]: $(\mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ (x{::}\mathit{int}) = \mathbf{g}_{G_T}\ [\widehat{}]_{G_T}\ (y{::}\mathit{int})) = ([x{=}\ y]\ (\mathit{mod}\ p))$
⟨*proof*⟩

lemma *pow-on-eq-card-GT-carrier-ext'*[*simp*]:
$((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}))\ [\widehat{}]_{G_T}\ x = ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}))\ [\widehat{}]_{G_T}\ y \longleftrightarrow [x{=}\ y]\ (\mathit{mod}\ p)$
⟨*proof*⟩

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** ^1 75)
where x ^1 $y \equiv$ x [^]1 (*to-int-mod-ring* (y::'*q mod-ring*))

abbreviation *div-in-grp* (**infixr** ÷1 70)
where x ÷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*⟩

lemma *pow-mod-order-*G_p: $\mathbf{g}_{G_p}\ [\hat{}]_{G_p}\ x = \mathbf{g}_{G_p}\ [\hat{}]_{G_p}\ (x\ mod\ p)$
⟨*proof*⟩

lemma *mod-ring-pow-mult-inv-*G_p[*symmetric*]: ($\mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ (*to-int-mod-ring* (a::'*q mod-ring*))) $\otimes_{G_p}$ inv_{G_p} ($\mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ (*to-int-mod-ring* *b*))
= $\mathbf{g}_{G_p}\ [\hat{}]_{G_p}$ (*to-int-mod-ring* ($a-b$))
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *to-int-mod-ring-ge-0*: *to-int-mod-ring x* ≥ *0*
⟨*proof*⟩

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

lemma *g-pow-to-int-mod-ring-of-int-mod-ring*: $\mathbf{g} \hat{}_{G_p}$ *of-int-mod-ring x* = $\mathbf{g}\ [\widehat{}]$ *x*
⟨*proof*⟩

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

lemma *carrier-inj-on-multc*: *c* ≠ *0* ⟹ *inj-on* (λ*x*. *x* $\hat{}_{G_p}$ *c*) (*carrier* G_p)
⟨*proof*⟩

2.0.2 mod_ring operations on pow of GT

lemma *pow-mod-order-G*$_T$: *g* ∈ *carrier* G_T ⟹ *g* $[\widehat{}]_{G_T}$ *x* = *g* $[\widehat{}]_{G_T}$ (*x mod p*)
⟨*proof*⟩

lemma *mod-ring-pow-mult-G*$_T$[*symmetric*]: *x* ∈ *carrier* G_T ⟹ (*x* $[\widehat{}]_{G_T}$ (*to-int-mod-ring* (*a*::*'q mod-ring*))) ⊗$_{G_T}$ (*x* $[\widehat{}]_{G_T}$ (*to-int-mod-ring b*))
= *x* $[\widehat{}]_{G_T}$ (*to-int-mod-ring* (*a*+*b*))
⟨*proof*⟩

2.0.3 bilinearity operations for mod_ring elements

lemma *e-bilinear*[*simp*]: *P* ∈ *carrier* G_p ∧ *Q* ∈ *carrier* G_p ⟹
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*⟩

lemma *e-linear-in-fst*:
assumes *P* ∈ *carrier* G_p ∧ *Q* ∈ *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*⟩

lemma *e-linear-in-snd*:
assumes *P* ∈ *carrier* G_p ∧ *Q* ∈ *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*⟩

lemma *addition-in-exponents-on-e*[*simp*]:
assumes $x \in carrier\ G_p \wedge y \in carrier\ G_p$
shows $(e\ x\ y)\ \hat{}_{G_T}\ a \otimes_{G_T} (e\ x\ y)\ \hat{}_{G_T}\ b = (e\ x\ y)\ \hat{}_{G_T}\ (a+b)$
⟨*proof*⟩

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

lemma *pow-on-eq-card-GT*[*simp*]: $(\mathbf{g}_{G_T}\ \hat{}_{G_T}\ x = \mathbf{g}_{G_T}\ \hat{}_{G_T}\ y) = (x{=}y)$
⟨*proof*⟩

lemma *pow-on-eq-card-GT-carrier-ext'*[*simp*]:
$((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}))\hat{}_{G_T}\ x = ((e\ \mathbf{g}_{G_p}\ \mathbf{g}_{G_p}))\hat{}_{G_T}\ y \longleftrightarrow x{=}y$
⟨*proof*⟩

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*⟩

lemma *game-alt-def2*:
game $\mathcal{A}$ = *TRY do* {
a ← *map-spmf to-type* (*sample-uniform* (*Coset.order G*));
a' ← $\mathcal{A}$ (*exp* **g** *a*);
return-spmf (*a* = *a'*)
} *ELSE return-spmf False*
⟨*proof*⟩

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* (λ*t'*. *exp* **g** ((*to-type* α)^*t'*)) [*0*..<*t*+*1*]);
return-spmf (*to-type* α = α')
} *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* (λ*t'*. *exp* **g** ((*to-type* α)^*t'*)) [*0*..<*t*+*1*]);
-::*unit* ← *assert-spmf* (*to-type* α = α');
return-spmf True
} *ELSE return-spmf False*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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 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*) ← $\mathcal{A}$ (*map* (λ*t'*. *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*⟩

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* {
α ← *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* {
α ← *sample-uniform* (*Coset.order G*);
(*c*, *g*) ← $\mathcal{A}$ (*map* (λ*t'*. *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*⟩

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* ⟨*proof*⟩

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*)
⟨*proof*⟩

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* 𝒜 = *TRY do* {
(*ck*, *vk*) ← *key-gen*;
(*c*, *i*, *v*, *w*, *v'*, *w'*) ← 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* = []
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

8.3 restrict

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

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

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

end

8.4 Examples

locale *examples*
begin

examples for select

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

lemma ¤$_{intS}$ (*1*::*int*) = [*1*::*int*]
  ⟨*proof*⟩

lemma ¤$_{listS\ intS}$ [*1*] = [*1*]
  ⟨*proof*⟩

lemma ¤$_{listS\ (listS\ intS)}$ [[*1*]] = [*1*]
  ⟨*proof*⟩

lemma ¤$_{prodS\ (listS\ intS)\ intS}$ ([*1*],*0*) = [*1*,*0*]
  ⟨*proof*⟩

examples for constrain

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

lemma [*0*,*1*,*2*] ⊵$_{intC}$ *3*
  ⟨*proof*⟩

lemma [*0*,*1*,*2*] ⊵$_{listC\ intC}$ [*3*,*3*,*3*]

⟨*proof*⟩

lemma $[0,1,2] \trianglerighteq_{prodC\ (listC\ intC)\ intC} ([3,3,3],3)$
⟨*proof*⟩

end

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

9 The Algebraic Group Model

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

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

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

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

context *cyclic-group*
begin

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

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

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

check the rules of an algebraic algorithm i.e. given the elements g, a group element, and the vector 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*⟩

end

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

sublocale *Algebraic-Algorithm* ⊆ *Restrictive-Comp sel con* ⟨*proof*⟩

locale *algebraic-algorithm-examples* = *cyclic-group*
begin

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

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

For simplicity let the adversary be id

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

interpretation *Algebraic-Algorithm G groupS groupC*
  ⟨*proof*⟩

lemma
  assumes *g* ∈ *carrier G*
  shows *restrict A-id g*

$= \mathcal{A}\text{-}id\ g$
⟨*proof*⟩

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* ⟨*proof*⟩

lemma
assumes $g1 \in carrier\ G \wedge g2 \in carrier\ G \wedge g3 \in carrier\ G$
shows $list\text{-}id.restrict\ \mathcal{A}\text{-}list\text{-}fst\ [g1,g2,g3]$
$= \mathcal{A}\text{-}list\text{-}fst\ [g1,g2,g3]$
⟨*proof*⟩

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*⟩

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 $('a)\ adv\ \ G => algebraic\text{-}adv$

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: $\varphi\ x - \varphi\ u = (\sum i{\le}degree\ \varphi.\ coeff\ \varphi\ i * x\hat{}i) - (\sum i{\le}degree\ \varphi.\ coeff\ \varphi\ i * x\hat{}i) = (x-u)(\sum i{\le}degree\ \varphi.\ coeff\ \varphi\ i * (\sum j{<}i.\ u\hat{}(i - Suc\ j){*}x\hat{}j))$ (for the last step see the lemma power_diff_sumr2) Hence: $\psi\ x = (\sum i{\le}degree\ \varphi.\ coeff\ \varphi\ i * (\sum j{<}i.\ u\hat{}(i - Suc\ j){*}x\hat{}j))$ 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 $(\sum i{\le}degree\ \varphi.\ coeff\text{-}i{*}x\hat{}i)$ 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: $(\sum i{\le}degree\ \varphi.\ coeff\ \varphi\ i * (\sum j{<}i.\ u\hat{}(i - Suc\ j){*}x\hat{}j)) = (\sum i{\le}degree\ \varphi.\ (\sum j{\in}\{i{<}..{<}Suc\ (degree\ \varphi)\}.\ coeff\ \varphi\ j * u\hat{}(j{-}Suc\ i)) * x\hat{}i)$

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 $\varphi(x) - \varphi(i) = (x-i) * \psi(x)$
    in (*poly φ i*, *g-pow-PK-Prod PK ψ*) — $(\varphi(i), g\hat{}\psi(\alpha))$)

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* $\div_{G_p}$ ($\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *i*)) $\otimes_{G_T}$ ((*e* $\mathbf{g}_{G_p}$ $\mathbf{g}_{G_p}$) $\hat{}_{G_T}$ *eval*) = *e C* $\mathbf{g}_{G_p}$))
    — $e(g\hat{}\psi(\alpha), g\hat{}\alpha \,/\, g\hat{}i) \otimes e(g,g)\hat{}\varphi(i) = e(C, g)$

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 ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *coeffs-n-add-nth*[*simp*]: $\forall$ *i*<*n*. *coeffs-n* φ *u l n* ! *i* = *nth-default 0 l i* + *poly.coeff* φ *n* * *u* ^ (*n* − *Suc i*)
⟨*proof*⟩

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

lemma *sum-split*: *m*≤*n* ⟹ (∑ *i*≤*n*. *f i*) = (∑ *i*≤*m*. *f i*) + (∑ *i*∈{*m*<..<*Suc n*}. *f i*)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*≤*t* ⟹ *map* (λ*t*. **g** ^$_{G_p}$ α ^ *t*) [*0*..<*t* + *1*] ! *i* = **g**$_{G_p}$ ^$_{G_p}$ ($\alpha\hat{}i$)
⟨*proof*⟩

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

lemma *g-pow-PK-Prod-to-fold*[*simp*]: *degree* $\varphi \leq t \Longrightarrow$ *g-pow-PK-Prod* (*map* (λ*t*. **g** ^$_{G_p}$ α ^ *t*) [*0*..<*t* + *1*]) φ
= *fold* (λ*pk g*. *g* ⊗$_{G_p}$ (**g**$_{G_p}$ ^$_{G_p}$ ($\alpha\hat{}$*pk*)) ^$_{G_p}$ (*poly.coeff* φ *pk*)) [*0*..<*Suc* (*degree* φ)] **1**$_{G_p}$
⟨*proof*⟩

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

lemma *poly-altdef-to-fold*[*symmetric*]: *n*≤*degree* φ ⟹ **g**$_{G_p}$ ^$_{G_p}$ ($\sum i{\leq}n$. *poly.coeff* φ *i* ∗ α ^ *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*⟩

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 $g\hat{}(\sum i{\leq}n.\ coeff\ \varphi\ i * \alpha\hat{}i)$, 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* ($\lambda t.\ \mathbf{g}\ \hat{}_{G_p}\ \alpha\ \hat{}\ t$) [*0..<t + 1*]) φ
$= \mathbf{g}_{G_p}\ \hat{}_{G_p}$ (*poly* $\varphi\ \alpha$)
⟨*proof*⟩

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

theorem *KZG-correct*: *correct-eval*
⟨*proof*⟩

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*) $\leq$ *ennreal* (*spmf game2 True*)
$\Longrightarrow$
spmf game1 True $\leq$ *spmf game2 True*
⟨*proof*⟩

lemma *unpack-bind-spmf*: $X = X' \Longrightarrow$ *bind-spmf X Y* = *bind-spmf X' Y*
⟨*proof*⟩

lemma *unpack-bind-spmf'*: $Y = Y' \Longrightarrow$ *bind-spmf X Y* = *bind-spmf X Y'*
⟨*proof*⟩

lemma *unpack-bind-spmf-fun*: $X = X' \Longrightarrow$ *bind-spmf X* ($\lambda y.$ *f y*) = *bind-spmf X'* ($\lambda y.$ *f y*)
⟨*proof*⟩

lemma *unpack-try-spmf*: $X = X' \Longrightarrow$ *TRY X ELSE Y* = *TRY X' ELSE Y*
⟨*proof*⟩

lemma *unpack-try-spmf'*: $Y = Y' \Longrightarrow$ *TRY X ELSE Y* = *TRY X ELSE Y'*
⟨*proof*⟩

lemma *spmf-eqI'*: $X = X' \Longrightarrow$ *spmf X Y* = *spmf X' Y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bind-spmf-independent-return-spmf*:
lossless-spmf x $\Longrightarrow$ *bind-spmf x* ($\lambda x.$ *return-spmf y*) = *return-spmf y*
⟨*proof*⟩

lemma *bind-spmf-le*:
($\bigwedge x.$ *spmf* (*f x*) *True* $\leq$ *spmf* (*f' x*) *True*) $\Longrightarrow$ *spmf* (*bind-spmf p f*) *True* $\leq$ *spmf* (*bind-spmf p f'*) *True*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *try-spmf-le*:
assumes *spmf x True* $\leq$ *spmf x' True*
shows *spmf* (*TRY x ELSE return-spmf False*) *True* $\leq$ *spmf* (*TRY x' ELSE return-spmf False*) *True*
⟨*proof*⟩

lemma *try-spmf-true-else-false-le*: *spmf* (*TRY X ELSE return-spmf False*) *True* $\leq$ *spmf X True*
⟨*proof*⟩

lemma *del-assert*: *spmf* (*bind-spmf* (*assert-spmf X*) (λ- . *Y*)) *True* $\leq$ *spmf Y True*

⟨*proof*⟩

lemma *assert-imp*: $(X \longrightarrow X') \Longrightarrow$
spmf (*bind-spmf* (*assert-spmf X*) (λ- . *Y*)) *True*
≤ *spmf* (*bind-spmf* (*assert-spmf X'*) (λ- . *Y*)) *True*
⟨*proof*⟩

lemma *assert-ret-unit*: *bind-spmf* (*assert-spmf x*) (λ*x* . *y*) = *bind-spmf* (*assert-spmf x*) (λ- . *y*)
⟨*proof*⟩

lemma *assert-based-eq*:
assumes $x \Longrightarrow y = y'$
shows *bind-spmf* (*assert-spmf x*) (λ- . *y*)
= *bind-spmf* (*assert-spmf x*) (λ- . *y'*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *assert-collapse*: *bind-spmf* (*assert-spmf X*) (λ-. *bind-spmf* (*assert-spmf Y*) (λ-. *Z*)) =
bind-spmf (*assert-spmf* (*X* ∧ *Y*)) (λ-. *Z*)
⟨*proof*⟩

lemma *assert-cong*: $X = Y \Longrightarrow$ *rel-spmf* (=) (*assert-spmf X*) (*assert-spmf Y*)
⟨*proof*⟩

lemma *rel-spmf-bind-assert-reflI*: $(Z \Longrightarrow$ *rel-spmf P X Y*$) \Longrightarrow$
rel-spmf P (*bind-spmf* (*assert-spmf Z*) (λ-. *X*)) (*bind-spmf* (*assert-spmf Z*) (λ-. *Y*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *weight-sample-uniform-set*: *weight-spmf* (*sample-uniform-set k n*) = *of-bool*

$(k \leq n)$
⟨*proof*⟩

lemma *weight-sample-uniform-set-k-0* [*simp*]: *weight-spmf* (*sample-uniform-set 0 n*) = *1*
⟨*proof*⟩

lemma *weight-sample-uniform-set-n-0* [*simp*]: *weight-spmf* (*sample-uniform-set k 0*) = *of-bool* (*k=0*)
⟨*proof*⟩

lemma *weight-sample-uniform-set-k-le-n* [*simp*]: $k \leq n \Longrightarrow$ *weight-spmf* (*sample-uniform-set k n*) = *1*
⟨*proof*⟩

lemma *lossless-sample-uniform-set* [*simp*]: *lossless-spmf* (*sample-uniform-set k n*) $\longleftrightarrow k \leq n$
⟨*proof*⟩

lemma *set-spmf-sample-uniform-set* [*simp*]: *set-spmf* (*sample-uniform-set k n*) = $\{x.\ x \subseteq \{..<n\} \wedge$ *card* $x = k\}$
⟨*proof*⟩

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* $x \subseteq \{..<n\} \wedge$ *length* $x = k\}$

lemma *spmf-sample-uniform-list*: *spmf* (*sample-uniform-list k n*) *x*
= *indicator* $\{x.$ *set* $x \subseteq \{..<n\} \wedge$ *length* $x = k\}$ *x* / $(n \hat{} k)$
⟨*proof*⟩

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

lemma *weight-sample-uniform-list-k-0* [*simp*]: *weight-spmf* (*sample-uniform-list 0 n*) = *1*
⟨*proof*⟩

lemma *weight-sample-uniform-list-n-0* [*simp*]: *weight-spmf* (*sample-uniform-list k 0*) = *of-bool* (*k=0*)
⟨*proof*⟩

lemma *weight-sample-uniform-list-k-le-n* [*simp*]: $k<n \Longrightarrow$ *weight-spmf* (*sample-uniform-list k n*) = *1*
⟨*proof*⟩

lemma *lossless-sample-uniform-list* [*simp*]: *lossless-spmf* (*sample-uniform-list k n*)

$\longleftrightarrow$ (k=0 $\vee$ 0<n)
⟨*proof*⟩

lemma *set-spmf-sample-uniform-list* [*simp*]: *set-spmf* (*sample-uniform-list k n*) = {*x*. *set x* ⊆ {..<*n*} ∧ *length x* = *k*}
⟨*proof*⟩

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*⟩

lemma *set-spmf-lhs-imp-length-gr-0*: *xs* ∈ *set-spmf* (*map-spmf* ((#) *x*) (*sample-uniform-list k p*))
⟹ *length xs* > *0*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

sampling a uniform random polynomial is equivalent to interpolating a polynomial 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*⟩

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$_p$*
⟨*proof*⟩

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*(*φ* ≠ *φ'*
∧ *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* (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^t)) [*0..<max-deg+1*]) φ
= *g-pow-PK-Prod* (*map* (λt. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^t)) [*0..<max-deg+1*]) φ'
⟷ *poly* (φ − φ') α = *0*
⟨*proof*⟩

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*⟩

lemma *root-imp-deg-1*:
assumes *monic* (*u*::'*e mod-ring poly*) ∧ *irreducible u*
shows (∃ *x*. *poly u x* = *0*) ⟷ *degree u* = *1*
⟨*proof*⟩

lemma *poly-eq0-is-find-α-sf-eq-α*:
assumes $\varphi \neq 0 \wedge$ *square-free* φ
shows *poly* φ α = *0* ⟹ *find-α-square-free* ($\mathbf{g}_{G_p}$ ^$_{G_p}$ α) φ = α
⟨*proof*⟩

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*: $\varphi \neq 0$ ⟹ *poly* φ α = *0* ⟹ *find-α* ($\mathbf{g}_{G_p}$ ^$_{G_p}$ α) φ = α
⟨*proof*⟩

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* ($\lambda t.$ $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ ((*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'))
= ($\varphi \neq \varphi' \wedge$ *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')
$\wedge$ (*poly* (φ − (φ')) (*of-int-mod-ring* (*int* α)::'*e mod-ring*) = *0*))
⟨*proof*⟩

lemma *helping-2-factorize-α*: $\varphi \neq \varphi' \wedge$ *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')
$\wedge$ (*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*⟩

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*⟩

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*⟩

lemma *t-DL-advantage-stronger-red-le-red*:
t-DL-G_p*.advantage* (*stronger-bind-reduction* $\mathcal{A}$) ≤ *t-DL-*G_p*.advantage* (*bind-reduction* $\mathcal{A}$)
⟨*proof*⟩

theorem *polynomial-binding*:
poly-bind-advantage $\mathcal{A}$ ≤ *t-DL-*G_p*.advantage* (*bind-reduction* $\mathcal{A}$)
⟨*proof*⟩

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
⟨*proof*⟩

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* ÷$_{G_p}$ *w'*) ^$_{G_p}$ (*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* ≠ φ'*-of-i* ∧ *w-i* ≠ *w'-i*
∧ *w-i* ∈ *carrier* G_p ∧ *w'-i* ∈ *carrier* G_p
∧ *verify-eval* ((*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ ((α)^*t*)) [*0*..<*max-deg+1*])) *C i* (φ*-of-i*, *w-i*)
∧ *verify-eval* ((*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ ((α)^*t*)) [*0*..<*max-deg+1*])) *C i* (φ'*-of-i*, *w'-i*)
shows $\mathbf{g}_{G_p}$ ^$_{G_p}$ (*1*/(α + (−*i*)))

$= (w\text{-}i \div_{G_p} w'\text{-}i) \mathbin{\hat{}}_{G_p} (1 \;/\; (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$

⟨*proof*⟩

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*⟩

lemma *helping-1*: $\varphi\text{-}of\text{-}i \neq \varphi'\text{-}of\text{-}i \wedge w\text{-}i \neq w'\text{-}i$
$\wedge$ *valid-argument i*
$\wedge$ $w\text{-}i \in carrier\ G_p \wedge\ w'\text{-}i \in carrier\ G_p$
$\wedge$ *verify-eval* $((map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} ((\alpha) \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1]))\ C\ i\ (\varphi\text{-}of\text{-}i, w\text{-}i)$
$\wedge$ *verify-eval* $((map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} ((\alpha) \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1]))\ C\ i\ (\varphi'\text{-}of\text{-}i, w'\text{-}i)$
$\wedge$ $\mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} (1/(\alpha + (-i)))$
$= (w\text{-}i \div_{G_p} w'\text{-}i) \mathbin{\hat{}}_{G_p} (1 \;/\; (\varphi'\text{-}of\text{-}i - \varphi\text{-}of\text{-}i))$
$\longleftrightarrow$
$\varphi\text{-}of\text{-}i \neq \varphi'\text{-}of\text{-}i \wedge w\text{-}i \neq w'\text{-}i$
$\wedge$ *valid-argument i*
$\wedge$ $w\text{-}i \in carrier\ G_p \wedge\ w'\text{-}i \in carrier\ G_p$
$\wedge$ *verify-eval* $((map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} ((\alpha) \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1]))\ C\ i\ (\varphi\text{-}of\text{-}i, w\text{-}i)$
$\wedge$ *verify-eval* $((map\ (\lambda t.\ \mathbf{g}_{G_p} \mathbin{\hat{}}_{G_p} ((\alpha) \mathbin{\hat{}} t))\ [0..<max\text{-}deg{+}1]))\ C\ i\ (\varphi'\text{-}of\text{-}i, w'\text{-}i)$
⟨*proof*⟩

14.4 Proof

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

⟨*proof*⟩

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*⟩

lemma *overestimate-reductions*: *spmf* (*t-SDH-*G_p*.game* (*ext-eval-bind-reduction* $\mathcal{A}$)) *True*
≤ *spmf* (*t-SDH-*G_p*.game* (*eval-bind-reduction* $\mathcal{A}$)) *True*
(**is** *spmf ?lhgame True* ≤ *spmf ?rhgame True*)
⟨*proof*⟩

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* $\mathcal{A}$ ≤ *t-SDH-*G_p*.advantage* (*eval-bind-reduction* $\mathcal{A}$)
⟨*proof*⟩

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* × *'state'*) ⇒ (*'e' argument* × (*'e' 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* × *int list*) ⇒
(*'e' mod-ring poly* × *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*
⟨*proof*⟩

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

prodC noConstrain (*prodC noConstrain* G_p*.groupC*)
⟨*proof*⟩

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

⇒ (*'e*, *'state*, *'a*) *knowledge-soundness-adversary2-AGM* ⇒ (*'a*, *'e*) *extractor* ⇒ *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*) ∧ *p-i* ≠ *p-i'* ∧ *valid-argument i* ∧ *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* ⇒ ('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 $\mathcal{E}$ $\mathcal{A}1$ $\mathcal{A}2$ *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* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg+1*];
vk = *map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (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*)

$\wedge$ c = *fold* (λ i *acc*. *acc* $\otimes$ *ck*!i [^] (*cvec*!i)) [0..<*length ck*] **1**
$\wedge$ *verify-eval vk c i* (*p-i*,w)
$\wedge$ *p-i* $\neq$ *p-i*$'$
$\wedge$ *valid-argument i*
$\wedge$ *valid-eval* (*p-i*,w)
$\longleftrightarrow$
length ck = *length cvec*
$\wedge$ c = *fold* (λ i *acc*. *acc* $\otimes$ *ck*!i [^] (*cvec*!i)) [0..<*length ck*] **1**
$\wedge$ *p-i* $\neq$ *p-i*$'$
$\wedge$ *valid-argument i*
$\wedge$ *valid-eval* (*p-i*,w)
$\wedge$ *valid-eval* (*p-i*$'$, w')
$\wedge$ *verify-eval vk c i* (*p-i*, w)
$\wedge$ *verify-eval vk c i* (*p-i*$'$, w')
⟨*proof*⟩

lemma *knowledge-soundness-game-alt-def*:
knowledge-soundness-game-AGM $\mathcal{A}1$ $\mathcal{A}2$ *E* =
eval-bind-game (*knowledge-soundness-reduction-ext E* $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

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* $\mathcal{A}$ $\mathcal{A}'$)) *True*
≤ *spmf* (*eval-bind-game* (*knowledge-soundness-reduction E* $\mathcal{A}$ $\mathcal{A}'$)) *True*
⟨*proof*⟩

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*
≤ *t-SDH-G*$_p$*.advantage* (*eval-bind-reduction* (*knowledge-soundness-reduction E* $\mathcal{A}1$ $\mathcal{A}2$))
⟨*proof*⟩

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

⟨*proof*⟩

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
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *length-filter-distinct*:
assumes *card* (*set xs*) > *card* (*set ys*)
shows *length* (*filter-distinct xs ys*) > *0*
⟨*proof*⟩

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*⟩

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*⟩

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** $\hat{}_{G_p}$ *i*) *plain-evals* ;
let C = *interpolate-on* (*zip* (*i*#*I*) *evals*) *α*;
let W = *map* (*λ*(*x*,*y*). (*y*,(*C* $\div_{G_p}$ **g**$_{G_p}$ $\hat{}_{G_p}$ *y*) $\hat{}_{G_p}$ ((*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** $\hat{}_{G_p}$ *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**$_{G_p}$ $\hat{}_{G_p}$ (*α*^*t*)) [*0*..<*max-deg*+*1*]!*1* = **g**$_{G_p}$ $\hat{}_{G_p}$ *α*
⟨*proof*⟩

The "find" in the tDL reduction is successful.

lemma *in-set-impl-find*: *x* ∈ *set xs* ⟹ *Some*(*x*) = *find* (*λy*. **g** $\hat{}_{G_p}$ *y* = **g** $\hat{}_{G_p}$ *x*)
xs
⟨*proof*⟩

lemma *exchange-lem*: *length I* = *max-deg* ∧ *distinct I* ∧ *α* ∈ *set* (*PickDistinct*
I#*I*)

$\longleftrightarrow$ *length I = max-deg* $\wedge$ *distinct I* $\wedge$ $\alpha \in$ *set* (*PickDistinct I#I*)
$\wedge$ α = (*case* (*find* (λx. $\mathbf{g}\ \hat{}_{G_p}\ x$ = *map* (λt. $\mathbf{g}_{G_p}\ \hat{}_{G_p}\ (\alpha\hat{}t)$) [*0..<max-deg+1*]!*1*) (*PickDistinct I#I*)) *of Some x => x | None => 1*)
⟨*proof*⟩

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*⟩

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* (λx *prod*. *prod* $\otimes_{G_p}$ *x*) (*map* (λx. $\mathbf{g}_{G_p}\ \hat{}_{G_p}\ f$ *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*⟩

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
= $\mathbf{g}\ \hat{}_{G_p}$ (*poly* (*lagrange-interpolation-poly xs-ys*) α)
⟨*proof*⟩

corollary *interpolate-on-*α*-is-*φ*-of-*α*-helper*:
assumes *length I* = *max-deg* $\wedge$ *distinct I*
and $\alpha \notin$ *set I*
shows *interpolate-on* (*zip* (*PickDistinct I # I*) (*map* (($\hat{}_{G_p}$) $\mathbf{g}$) *evals*)) α
= $\mathbf{g}\ \hat{}_{G_p}$ (*poly* (*lagrange-interpolation-poly* (*zip* (*PickDistinct I # I*) *evals*)) α)
⟨*proof*⟩

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* ≤ *max-deg+1*

shows *interpolate-on* (*map* (λ(*x*,*y*).(*x*,$\mathbf{g}_{G_p}$ ^$_{G_p}$ *y*)) *xs-ys*) α = *g-pow-PK-Prod*
(*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg* + *1*]) (*lagrange-interpolation-poly* *xs-ys*)
⟨*proof*⟩

lemma *split-pow-div-*G_p:
shows **g** ^$_{G_p}$ (*y*/*x*) = (**g** ^$_{G_p}$ *y*) ^$_{G_p}$ (*1*/*x*)
⟨*proof*⟩

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* ≤ *max-deg* + *1*
and α*-nin-xs*: α ∉ *set* (*map fst* (*tl xs-ys*))
shows *map* (λ*i*. (*Eval* (*map* (λ*t*. **g** ^$_{G_p}$ (α ^*t*)) [*0*..<*max-deg* + *1*]) () (*lagrange-interpolation-poly* *xs-ys*) *i*)) (*map fst* (*tl xs-ys*))
= *map* (λ(*x*,*y*). (*y*,($\mathbf{g}_{G_p}$ ^$_{G_p}$ (*poly* (*lagrange-interpolation-poly xs-ys*) α) ÷$_{G_p}$ $\mathbf{g}_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (*1*/(α−*x*)))) (*tl xs-ys*)
⟨*proof*⟩

corollary *witness-calc-helper*:
assumes *length I* = *max-deg* ∧ *distinct I*
and α ∉ *set I*
and *evals*
∈ *set-spmf*
(*map-spmf* (*map* (λ*x*. *of-int-mod-ring* (*int x*)))
(*sample-uniform-list* (*max-deg* + *1*) (*order* G_p)))
shows *map* (λ*i*. (*Eval* (*map* (λ*t*. **g** ^$_{G_p}$ (α ^*t*)) [*0*..<*max-deg* + *1*]) () (*lagrange-interpolation-poly* (*zip* (*PickDistinct I* # *I*) *evals*)) *i*)) *I*
= *map2* (λ*x y*. (*y*,($\mathbf{g}_{G_p}$ ^$_{G_p}$ (*poly* (*lagrange-interpolation-poly* (*zip* (*PickDistinct* *I* # *I*) *evals*)) α) ÷$_{G_p}$ $\mathbf{g}_{G_p}$ ^$_{G_p}$ *y*) ^$_{G_p}$ (*1*/(α−*x*)))) *I* (*tl evals*)
⟨*proof*⟩

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*⟩

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 $\mathcal{A}1$ $\mathcal{A}2$ = *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*::*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' ← $\mathcal{A}2$ *PK* σ *C I W*;
return-spmf (φ = φ')} *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* {

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

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**
$game2\text{-}w\text{-}assert\ \mathcal{A}1\ \mathcal{A}2 = TRY\ do\ \{$
$(\alpha,\ PK) \leftarrow Setup;$
$(I,\sigma) \leftarrow \mathcal{A}1\ PK;$
$\text{-} :: unit \leftarrow assert\text{-}spmf\ (length\ I = max\text{-}deg \wedge distinct\ I);$
$let\ i = PickDistinct\ I;$
$evals \leftarrow map\text{-}spmf\ (map\ (of\text{-}int\text{-}mod\text{-}ring \circ int))\ (sample\text{-}uniform\text{-}list\ (max\text{-}deg{+}1)\ (order\ G_p));$
$let\ \varphi = lagrange\text{-}interpolation\text{-}poly\ (zip\ (i\#I)\ evals);$
$let\ C = interpolate\text{-}on\ (zip\ (i\#I)\ (map\ (\lambda i.\ \mathbf{g}\ \hat{}\ i)\ evals))\ \alpha;$
$let\ witn\text{-}tupel = map\ (\lambda(x,y).\ (y,(C\ \div_{G_p}\ \mathbf{g}_{G_p}\ \hat{}_{G_p}\ y)\ \hat{}_{G_p}\ (1/(\alpha{-}x))))\ (zip\ I\ (tl\ evals));$
$\varphi' \leftarrow \mathcal{A}2\ PK\ \sigma\ C\ I\ witn\text{-}tupel;$
$\text{-}::unit \leftarrow assert\text{-}spmf\ (\varphi = \varphi');$
$return\text{-}spmf\ (hd\ evals = poly\ \varphi'\ i)\}\ ELSE\ return\text{-}spmf\ False$

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**
$game2\text{-}wo\text{-}assert\ \mathcal{A}1\ \mathcal{A}2 = TRY\ do\ \{$
$(\alpha,\ PK) \leftarrow Setup;$
$(I,\sigma) \leftarrow \mathcal{A}1\ PK;$
$let\ i = PickDistinct\ I;$
$evals \leftarrow map\text{-}spmf\ (map\ (of\text{-}int\text{-}mod\text{-}ring \circ int))\ (sample\text{-}uniform\text{-}list\ (max\text{-}deg{+}1)\ (order\ G_p));$
$let\ \varphi = lagrange\text{-}interpolation\text{-}poly\ (zip\ (i\#I)\ evals);$
$(\alpha,\ PK) \leftarrow Setup;$
$let\ C = interpolate\text{-}on\ (zip\ (i\#I)\ (map\ (\lambda i.\ \mathbf{g}\ \hat{}\ i)\ evals))\ \alpha;$

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*));
$\varphi' \leftarrow \mathcal{A}2$ *PK* σ *C I witn-tupel*;
return-spmf (*hd evals = poly* φ' *i*)} *ELSE return-spmf False*

lemma *PickDistinct-Prop*: *length a = max-deg* $\wedge$ *distinct a*
$\Longrightarrow$ *distinct* (*PickDistinct a*#*a*) $\wedge$ *length* (*PickDistinct a*#*a*) = *max-deg + 1*
⟨*proof*⟩

corollary *PickDistinct-Prop'* : *length a = max-deg* $\wedge$ *distinct a* $\wedge$ *valid-poly* φ
$\Longrightarrow$ *distinct* (*PickDistinct a*#*a*) $\wedge$ *length* (*PickDistinct a*#*a*) = *max-deg + 1*
⟨*proof*⟩

lemma *distinct-map-fst-zip*: *distinct a* $\Longrightarrow$ *distinct* (*map fst* (*zip a b*))
⟨*proof*⟩

lemma *card-eq-ord*: *CARD*('*e*) = *order* G_p
⟨*proof*⟩

Step 1 of the proof.

lemma *hiding-game-to-game1*:
shows *spmf* (*weak-eval-hiding-game* $\mathcal{A}1$ $\mathcal{A}2$) *True = spmf* (*game1* $\mathcal{A}1$ $\mathcal{A}2$) *True*
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

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)))
⟨*proof*⟩

Step 2 of the proof:

lemma *fundamental-lemma-game1-game2*:
assumes *lossless-*$\mathcal{A}1$: $\bigwedge$*PK . lossless-spmf* ($\mathcal{A}1$ *PK*)
and *lossless-*$\mathcal{A}2$: $\bigwedge$*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* $\leq$ *spmf* (*game2* $\mathcal{A}1$ $\mathcal{A}2$) *True + t-DL-*G_p*.advantage* (*reduction-tDL* $\mathcal{A}1$)
⟨*proof*⟩

Step 3 of the proof:

lemma *game2-le-DL*: *spmf* (*game2* $\mathcal{A}1$ $\mathcal{A}2$) *True* $\leq$ *DL-*G_p*.advantage* (*reduction* $\mathcal{A}1$ $\mathcal{A}2$)
(**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

Finally we assemble all proof steps for the weak hiding theorem

theorem *weak-hiding*:
assumes *lossless-*$\mathcal{A}1$: $\bigwedge$*PK . lossless-spmf* ($\mathcal{A}1$ *PK*)

and *lossless-$\mathcal{A}2$*: $\bigwedge PK\ \sigma\ C\ I\ W$. *lossless-spmf* ($\mathcal{A}2\ PK\ \sigma\ C\ I\ W$)
shows *weak-eval-hiding-advantage $\mathcal{A}1$ $\mathcal{A}2$*
$\leq$ *DL-G_p.advantage* (*reduction $\mathcal{A}1$ $\mathcal{A}2$*) + *t-DL-G_p.advantage* (*reduction-tDL $\mathcal{A}1$*)
⟨*proof*⟩

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*
⟨*proof*⟩

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**
ψ_B *B φ = do* {

let prod-B = prod ($\lambda i.$ [:−*i*,*1*:]) *B*;
(φ − (*r B* φ)) *div prod-B*}

$\varphi(x)$= ($\varphi(x)$/$\prod i{\in}B.(x{-}i)$) ∗ $\prod i{\in}B.(x{-}i)$ + φ *mod* $\prod i{\in}B.(x{-}i)$

lemma $\varphi = \psi_B\ B\ \varphi * (prod\ (\lambda i.\ [:{-}i,1:])\ \ B) + r\ B\ \varphi$
⟨*proof*⟩

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*⟩

lemma *deg-r-B-le*: *degree* (*r B* φ) ≤ *card B*
⟨*proof*⟩

lemma *deg-r-B-less*: *B* ≠ {} ⟹ *degree* φ > *card B* ⟹ *degree* (*r B* φ) < *card B*
⟨*proof*⟩

lemma *deg-div*: *degree* ((*x*::′*e mod-ring poly*) *div y*) ≤ *degree x*
⟨*proof*⟩

lemma *deg-*ψ_B: *degree* (ψ_B *B* φ) ≤ *degree* φ
⟨*proof*⟩

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

lemma *i-in-B-prod-B-zero*[*simp*]:
assumes(*i*::′*e argument*) ∈ *B*
shows *poly* (*prod* ($\lambda i.$ [:−*i*,*1*:]) *B*) *i* = *0*
⟨*proof*⟩

$r(i)$=$\varphi(i)$ 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*⟩

lemma ($\prod i{\in}B.$ [:− *i*, *1*:]) *dvd* φ ⟹ φ *mod* ($\prod i{\in}B.$ [:− *i*, *1*:]) = *0*
⟨*proof*⟩

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 $\varphi(x)/(\prod i{\in}B.\ (x{-}i))$ i.e. $\varphi(x)$ *mod* $(\prod i{\in}B.\ (x{-}i))$
ψ = ψ_B *B* φ; — $(\varphi(x) - r(x))\ /\ (\prod i{\in}B.\ (x{-}i))$
w-B = *g-pow-PK-Prod PK* ψ — $g\hat{}\psi(\alpha)$
in
(*r*, *w-B*) — (*B*, *r*(*x*), $g\hat{}\psi(\alpha)$)
)

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

definition *verify-eval-batch* :: *'a vk* ⇒ *'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 $\otimes_{G_T}$ (*e* **g** *g-pow-r*) = *e C* **g**))
— $e(g\hat{}(\prod i{\in}B.\ (\alpha{-}i)),\ g\hat{}\psi(\alpha)) \otimes e(g,g\hat{}r(\alpha)) = e(C,\ g)$

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 ⟨*proof*⟩

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* ($\prod i{\in}B$. [:− *i*, *1*:]) α) (**g** ^$_{G_p}$ *poly* (ψ_B *B* φ) α)
  $\otimes_{G_T}$ (*e* **g** (**g** ^$_{G_p}$ *poly* (*r B* φ) α))
  = *e* (**g** ^$_{G_p}$ *poly* φ α) **g**)
⟨*proof*⟩

theorem *KZG-correct*: *bKZG.correct-eval*
  ⟨*proof*⟩

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* $\mathcal{A}$ ≤ *t-DL-G$_p$.advantage* (*bind-reduction* $\mathcal{A}$)
  ⟨*proof*⟩

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*
⟨*proof*⟩

20.2 Binding for the batched evaluation according to KZG10

type-synonym $('a, 'e)$ *adversary* =
$'a$ *ck* $\Rightarrow$
($'a$ *commit* $\times$ $'e$ *argument* $\times$ $'e$ *evaluation* $\times$ $'a$ *witness* $\times$ $'e$ *argument set*
$\times$ $'a$ *witness* $\times$ $'e$ *batch-evaluation*) *spmf*

This is the formalized evaluation binding game

definition *bind-game* :: $('a, 'e)$ *adversary* $\Rightarrow$ *bool spmf*
where *bind-game* $\mathcal{A}$ = *TRY do* {
(*ck*,*vk*) $\leftarrow$ *key-gen*;
(*C*, *i*, φ-*i*, *w-i*, *B*, *w-B*, *r-x*) $\leftarrow$ $\mathcal{A}$ *ck*;
- :: *unit* $\leftarrow$ *assert-spmf* (*i* $\in$ *B* $\wedge$ φ-*i* $\neq$ *poly r-x i*
$\wedge$ *valid-argument-batch B* $\wedge$ *valid-eval* (φ-*i*, *w-i*) $\wedge$ *valid-eval-batch* (*r-x*, *w-B*));

let b = *verify-eval vk C i* (φ-*i*, *w-i*);
let b' = *verify-eval-batch vk C B* (*r-x*,*w-B*);
return-spmf (*b* $\wedge$ *b'*)} *ELSE return-spmf False*

The advantage of the adversary over the evaluation binding game is the probability that it wins.

definition *bind-advantage* :: $('a, 'e)$ *adversary* $\Rightarrow$ *real*
where *bind-advantage* $\mathcal{A}$ $\equiv$ *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'* ÷$_{G_p}$ *w-i*) **g**) $\hat{}_{G_T}$ (*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* (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'* ÷$_{G_p}$ *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*) ← *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*);
-::*unit* ← *assert-spmf* (*b* ∧ *b'*);
return-spmf True} *ELSE return-spmf False*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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 $i \in B \wedge \varphi\text{-}i \neq$ *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* ($\lambda t.$ **g** $\hat{}\,\alpha$ $\hat{}\,t$) [*0..<max-deg + 1*]) *C i* (φ-*i*, *w-i*)
$\wedge$ *verify-eval-batch* (*map* ($\lambda t.$ **g** $\hat{}\,\alpha$ $\hat{}\,t$) [*0..<max-deg + 1*]) *C B* (*r-x*, *w-B*)
shows *e* **g** **g** $\hat{}_{G_T}$ (*1 / (α + − i)*)
=
(*e* (*g-pow-PK-Prod* (*map* ($\lambda t.$ **g** $\hat{}\,\alpha$ $\hat{}\,t$) [*0..<max-deg + 1*])
((∏ *i*∈*B*. [:− *i*, *1*:]) *div* [:− *i*, *1*:]))
w-B
$\otimes_{G_T}$
e (*g-pow-PK-Prod* (*map* ($\lambda t.$ **g** $\hat{}\,\alpha$ $\hat{}\,t$) [*0..<max-deg + 1*])
((*r-x* − [:*poly r-x i*:]) *div* [:− *i*, *1*:]) ⊗
inv w-i)
g) $\hat{}_{G_T}$
(*1 / (φ-i − poly r-x i*))
(**is** *?goal* = *?cmpt*)
⟨*proof*⟩

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*. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *α* $\hat{}$ *t*) [*0..<max-deg + 1*]) *C i* (*φ-i*,*w-i*) ∧
verify-eval-batch (*map* (λ*t*. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ (*α*::′*e mod-ring*) $\hat{}$ *t*) [*0..<max-deg + 1*]) *C B* (*r-x*,*w-B*) ∧
e **g** **g** $\hat{}_{G_T}$ (*1 / (α + − i)*) =
(*e* (*g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *α* $\hat{}$ *t*) [*0..<max-deg + 1*])
((∏ *i*∈*B*. [:− *i*, *1*:]) *div* [:− *i*, *1*:]))
w-B $\otimes_{G_T}$
e (*g-pow-PK-Prod* (*map* (λ*t*. $\mathbf{g}_{G_p}$ $\hat{}_{G_p}$ *α* $\hat{}$ *t*) [*0..<max-deg + 1*])
((*r-x* − [:*poly r-x i*:]) *div* [:− *i*, *1*:]) ⊗
inv w-i)
g) $\hat{}_{G_T}$
((*1*::′*e mod-ring*) / (*φ-i − poly r-x i*)))
⟷
(*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* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*]) *C i* (*φ-i*,*w-i*) ∧
verify-eval-batch (*map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ *α* ^ *t*) [*0*..<*max-deg* + *1*]) *C B* (*r-x*,*w-B*))
⟨*proof*⟩

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*⟩

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* $\mathcal{A}$ = *t-BSDH.advantage* (*ext-reduction* $\mathcal{A}$)
⟨*proof*⟩

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* $\mathcal{A}$)) *True* ≤ *spmf* (*t-BSDH.game* (*reduction* $\mathcal{A}$)) *True*
(**is** *spmf ?lhgame True* ≤ *spmf ?rhgame True*)
⟨*proof*⟩

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* $\mathcal{A}$ ≤ *t-BSDH.advantage* (*reduction* $\mathcal{A}$)
⟨*proof*⟩

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* × *int list*) ⇒
(*'e' mod-ring poly* × *unit*) *spmf*

restrict for AGM adversaries 1 and 2

interpretation *AGM1*: *Algebraic-Algorithm* G_p *listS* G_p*.groupS prodC* G_p*.groupC noConstrain*
⟨*proof*⟩

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

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

prodC noConstrain (*prodC noConstrain* (*prodC noConstrain* G_p*.groupC*))

⟨*proof*⟩

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

⇒ ('*e*, '*state*, '*a*) *knowledge-soundness-adversary2-AGM* ⇒ ('*a*, '*e*) *extractor* ⇒
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*, *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*

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

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

Extractor definition

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

lemma *reduction-map-imp*:
let ck = *map* (λ*t*. $\mathbf{g}_{G_p}$ ^$_{G_p}$ (α^*t*)) [*0*..<*max-deg+1*];
vk = *map* (λ*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* ⇒'*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*⟩

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*⟩

Finally we put everything together: we conclude that for every efficient adversary 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*))
⟨*proof*⟩

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.