

Polynomial Commitment Schemes

Tobias Rothmann

July 13, 2026

Abstract

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

Contents

1	Pairings	3
2	Mathematical Primitives	5
2.0.1	mod_ring operations on pow of Gp	5
2.0.2	mod_ring operations on pow of GT	6
2.0.3	bilinearity operations for mod_ring elements	6
3	The Discrete Logarithm Assumption	7
4	The t-Discrete Logarithm Assumption	8
5	The t-Strong Diffie-Hellman Assumption	9
6	The t-Bilinear Strong Diffie-Hellman Assumption	10
7	Polynomial Commitment Schemes	12

8	Restrictive Computations	15
8.1	Select	16
8.2	Constrain	17
8.3	restrict	19
8.4	Examples	19
9	The Algebraic Group Model	20
9.1	Example for the ML function	22
10	KZG function definitions	23
10.1	Setup	23
11	Correctness of the KZG	25
11.0.1	Helping lemmas for the computation of ψ	26
11.0.2	Helping lemmas about the public parameters PK	27
12	Extensions to CryptHOL	28
12.1	SPMF True	29
12.2	Sample uniform set	30
12.3	Sample uniform list	31
12.4	Sample uniform polynomial	32
13	Polynomial Binding of the KZG	33
13.1	t-DL game	33
13.2	Helping lemmas	35
13.2.1	Literal helping lemmas	35
13.3	KZG poly bind game to strong reduction game - equivalence theorem	37
14	Evaluation Binding of the KZG	37
14.1	t-SDH game	37
14.2	Defining a reduction adversary from evaluation binding to t-SDH	38
14.3	Helping definitions	38
14.3.1	Literal helping lemmas	39
14.4	Proof	39
15	Knowledge Soundness of the KZG	40
15.1	Helping definitions	42
15.2	Helping lemmas	42
16	Hiding of the KZG	44
16.1	Definitions for the weak hiding game	44
16.1.1	DL game	44
16.1.2	Reduction	45

16.2 Hiding proof	46
16.2.1 Helping lemmas	46
17 Batch Opening Definition	52
17.1 Polynomial operations and prerequisites	52
17.2 Function definitions	53
18 Correctness of the batched KZG	54
19 Polynomial Binding of the batched KZG	55
20 Evaluation Binding of the batched KZG	55
20.1 t-BSDH game	56
20.2 Binding for the batched evaluation according to KZG10 . . .	56
20.2.1 Defining a reduction function to t-BSDH	56
20.2.2 Helping lemmas	57
20.3 Literal helping lemma	58
20.4 KZG eval bind game to reduction game - equivalence theorem	59
21 Knowledge Soundness of the batched KZG	60
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:  $e \in \text{carrier } G_p \rightarrow \text{carrier } G_p \rightarrow \text{carrier } G_T$  and
   $e$ -bilinearity[simp]:  $\forall a b :: \text{int} . \forall P Q. P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \longrightarrow$ 
     $e (P [\bigwedge]_{G_p} a) (Q [\bigwedge]_{G_p} b)$ 
  =  $(e P Q) [\bigwedge]_{G_T} (a * b)$  and

```

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

The pairing function e is linear in the first component

lemma *e-linear-in-fst*:
assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
shows $e P [\cdot]_{G_p} (a::int) Q = (e P Q) [\cdot]_{G_T} a$
 $\langle \text{proof} \rangle$

The pairing function e is linear in the second component

lemma *e-linear-in-snd*:
assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
shows $e P (Q [\cdot]_{G_p} (a::int)) = (e P Q) [\cdot]_{G_T} a$
 $\langle \text{proof} \rangle$

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

this follows from non-degeneracy

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

lemma *e-g-g-in-carrier-GT[simp]*: $e \mathbf{g}_{G_p} \mathbf{g}_{G_p} \in \text{carrier } G_T$
 $\langle \text{proof} \rangle$

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

lemma *pow-on-eq-card-GT[simp]*: $(\mathbf{g}_{G_T} [\cdot]_{G_T} (x::int) = \mathbf{g}_{G_T} [\cdot]_{G_T} (y::int)) = ([x = y] \pmod{p})$
 $\langle \text{proof} \rangle$

lemma *pow-on-eq-card-GT-carrier-ext'[simp]*:
 $((e \mathbf{g}_{G_p} \mathbf{g}_{G_p})) [\cdot]_{G_T} x = ((e \mathbf{g}_{G_p} \mathbf{g}_{G_p})) [\cdot]_{G_T} y \longleftrightarrow [x = y] \pmod{p}$
 $\langle \text{proof} \rangle$

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  $p :: \text{int}$ 
  and  $e :: 'a \Rightarrow 'a \Rightarrow 'c$ 
+
fixes type-q ::  $('q :: \text{prime-card})$  itself
and max-deg :: nat
assumes
  p-gr-two:  $p > 2$  and
  d-pos:  $\text{max-deg} > 0$  and
  CARD-q:  $\text{int } (\text{CARD}('q)) = p$  and
  d-l-p:  $\text{max-deg} < p$ 
begin

abbreviation pow-mod-ring (infixr  $\hat{1}$  75)
  where  $x \hat{1} y \equiv x [\hat{1}]_1 (\text{to-int-mod-ring } (y :: 'q \text{ mod-ring}))$ 

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

```

2.0.1 mod_ring operations on pow of Gp

```

lemma carrier-pow-mod-order-Gp:
  assumes  $g \in \text{carrier } G_p$ 
  shows  $g [\hat{1}]_{G_p} x = g [\hat{1}]_{G_p} (x \text{ mod } p)$ 
<proof>

lemma pow-mod-order-Gp:  $\mathbf{g}_{G_p} [\hat{1}]_{G_p} x = \mathbf{g}_{G_p} [\hat{1}]_{G_p} (x \text{ mod } p)$ 
<proof>

lemma mod-ring-pow-mult-inv-Gp[symmetric]:  $(\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } (a :: 'q \text{ mod-ring}))) \otimes_{G_p} \text{inv}_{G_p} (\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } b))$ 
  =  $\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } (a - b))$ 
<proof>

lemma mod-ring-pow-mult-Gp[symmetric]:  $(\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } (a :: 'q \text{ mod-ring}))) \otimes_{G_p} (\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } b))$ 
  =  $\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } (a + b))$ 
<proof>

lemma mod-ring-pow-pow-Gp[symmetric]:  $(\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } (a :: 'q \text{ mod-ring}))) [\hat{1}]_{G_p} (\text{to-int-mod-ring } (b :: 'q \text{ mod-ring}))$ 
  =  $\mathbf{g}_{G_p} [\hat{1}]_{G_p} (\text{to-int-mod-ring } (a * b :: 'q \text{ mod-ring}))$ 

```

$\langle \text{proof} \rangle$

lemma *to-int-mod-ring-ge-0*: *to-int-mod-ring* $x \geq 0$
 $\langle \text{proof} \rangle$

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

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

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

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

2.0.2 mod_ring operations on pow of GT

lemma *pow-mod-order-GT*: $g \in \text{carrier } G_T \implies g [\]_{G_T} x = g [\]_{G_T} (x \text{ mod } p)$
 $\langle \text{proof} \rangle$

lemma *mod-ring-pow-mult-GT[symmetric]*: $x \in \text{carrier } G_T \implies (x [\]_{G_T} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) \otimes_{G_T} (x [\]_{G_T} (\text{to-int-mod-ring } b))$
 $= x [\]_{G_T} (\text{to-int-mod-ring } (a+b))$
 $\langle \text{proof} \rangle$

2.0.3 bilinearity operations for mod_ring elements

lemma *e-bilinear[simp]*: $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p \implies$
 $e (P [\]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) (Q [\]_{G_p} (\text{to-int-mod-ring } b))$
 $= (e P Q) [\]_{G_T} (\text{to-int-mod-ring } (a*b))$
 $\langle \text{proof} \rangle$

lemma *e-linear-in-fst*:
assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
shows $e (P [\]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) (Q) = (e P Q) [\]_{G_T} (\text{to-int-mod-ring } a)$
 $\langle \text{proof} \rangle$

lemma *e-linear-in-snd*:
assumes $P \in \text{carrier } G_p \wedge Q \in \text{carrier } G_p$
shows $e (P) (Q [\]_{G_p} (\text{to-int-mod-ring } (a::'q \text{ mod-ring}))) = (e P Q) [\]_{G_T} (\text{to-int-mod-ring } a)$
 $\langle \text{proof} \rangle$

```

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

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

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

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

end

end

theory DL-assumption

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

begin

```

3 The Discrete Logarithm Assumption

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

```

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

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

```

The discrete Logarithm game

```

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

```

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

```

definition advantage :: ('a, 'c) adversary  $\Rightarrow$  real

```

where *advantage* $\mathcal{A} = \text{spmf } (\text{game } \mathcal{A}) \text{ True}$

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  $\mathcal{A} = \text{TRY do } \{$ 
   $a \leftarrow \text{sample-uniform } (\text{Coset.order } G);$ 
   $a' \leftarrow \mathcal{A} (\text{exp } \mathbf{g} (\text{to-type } a));$ 
   $\text{--::unit} \leftarrow \text{assert-spmf } (\text{to-type } a = a');$ 
   $\text{return-spmf True}$ 
 $\} \text{ ELSE return-spmf False}$ 
 $(\text{is } ?lhs = ?rhs)$ 
 $\langle \text{proof} \rangle$ 
```

lemma *game-alt-def2*:

```
game  $\mathcal{A} = \text{TRY do } \{$ 
   $a \leftarrow \text{map-spmf to-type } (\text{sample-uniform } (\text{Coset.order } G));$ 
   $a' \leftarrow \mathcal{A} (\text{exp } \mathbf{g} a);$ 
   $\text{return-spmf } (a = a')$ 
 $\} \text{ ELSE return-spmf False}$ 
 $\langle \text{proof} \rangle$ 
```

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

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

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

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

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

definition *advantage* :: ('a,'c) adversary $\Rightarrow$ 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 {
 $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$;
 $\alpha' \leftarrow \mathcal{A} \text{ (map } (\lambda t'. \text{exp } \mathbf{g} \text{ ((to-type } \alpha) \hat{\sim} t')) [0..<t+1])$;
 $\text{--::unit} \leftarrow \text{assert-spmf } (\text{to-type } \alpha = \alpha')$;
 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 $\Rightarrow$ ('c::prime-card) mod-ring
and *exp* :: 'a $\Rightarrow$ 'c mod-ring $\Rightarrow$ 'a
begin

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

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

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

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

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

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

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^t . 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  $\Rightarrow$  ('e::prime-card) mod-ring
  and exp :: 'a  $\Rightarrow$  'e mod-ring  $\Rightarrow$  'a
  and exp-G_T :: 'c  $\Rightarrow$  'e mod-ring  $\Rightarrow$  'c
  and e :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'c — bilinear function from G to G_T
begin

```

```

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

```

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

```

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

```

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

```

definition advantage :: ('a, 'e, 'c) adversary  $\Rightarrow$  real
  where advantage A = spmf (game 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 A = TRY do {
     $\alpha \leftarrow \text{sample-uniform } (\text{Coset.order } G)$ ;
     $(c, g) \leftarrow A (\text{map } (\lambda t'. \text{exp } \mathbf{g} ((\text{to-type } \alpha) \hat{\sim} t')) [0..<t+1])$ ;
     $\text{--::unit} \leftarrow \text{assert-spmf } (\text{exp-G}_T (e \ \mathbf{g} \ \mathbf{g}) (1/((\text{to-type } \alpha)+c)) = g)$ ;
     $\text{return-spmf True}$ 
  } 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 com-
    mitment
  and valid-poly :: 'r poly ⇒ bool — checks whether a polynomial is a valid
  message e.g. it’s degree is bounded
  and valid-argument :: 'argument ⇒ bool — checks whether an argument is a
  valid message e.g. set size in a batched version
  and valid-eval :: ('evaluation × 'witness) ⇒ bool — checks whether an evaluation
  is a valid message e.g. a valid group element
begin

```

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

```

sublocale cs: abstract-commitment key-gen commit verify-poly valid-poly ⟨proof⟩

```

```

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

```

definition *correct-cs*
where *correct-cs* $\equiv$ *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 $\Rightarrow$ 'argument $\Rightarrow$ bool spmf
where *correct-eval-game* *p i* = do {
 (*ck*, *vk*) $\leftarrow$ *key-gen*;
 (*c*, *d*) $\leftarrow$ *commit ck p*;
 let *w* = *eval ck d p i*;
 return-spmf (*verify-eval vk c i w*)
}

lemma *lossless-correct-eval-game*: $\llbracket$ lossless-spmf *key-gen*;
 $\bigwedge ck p. \text{valid-msg } p \implies \text{lossless-spmf } (\text{commit } ck p) \rrbracket$
 $\implies \text{valid-msg } p \implies \text{lossless-spmf } (\text{correct-eval-game } p i)$
<proof>

captures the perfect correctness property of eval

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

We again reuse the previous work on commitment schemes

definition *poly-bind-game*
where *poly-bind-game* $\equiv$ *cs.bind-game*

definition *poly-bind-advantage*
where *poly-bind-advantage* $\equiv$ *cs.bind-advantage*

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

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

definition *eval-bind-advantage* :: ('ck, 'commit, 'argument, 'evaluation, 'witness)
eval-bind-adversary $\Rightarrow$ *real*
where *eval-bind-advantage* $\mathcal{A} \equiv \text{spmf } (\text{eval-bind-game } \mathcal{A}) \text{ True}$

type-synonym ('vk', 'argument', 'state') *eval-hiding-adversary1* =
'vk' $\Rightarrow$ ('argument' list $\times$ 'state') *spmf*

type-synonym ('r', 'vk', 'commit', 'argument', 'evaluation', 'witness', 'state') *eval-hiding-adversary2*
= $(\text{'vk'} \Rightarrow \text{'state'} \Rightarrow \text{'commit'} \Rightarrow \text{'argument' list} \Rightarrow (\text{'evaluation'} \times \text{'witness'}) \text{ list} \Rightarrow (\text{'r' poly}) \text{ spmf})$

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

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

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

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

type-synonym ('r', 'commit', 'trapdoor') *extractor* = 'commit' $\Rightarrow$ ('r' poly $\times$ '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* *A1* *A2* *E* = *TRY* do {

(*ck*, *vk*) ← *key-gen*;

(*c*, *σ*) ← *A1* *ck*;

(*p*, *d*) ← *E* *c*;

(*i*, (*p-i*, *π*)) ← *A2* *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* *A1* *A2* *E* ≡ *spmf* (*knowledge-soundness-game* *A1* *A2* *E*) *True*

end

end

theory *Restrictive-Comp*

imports *CryptHOL.CryptHOL*

begin

8 Restrictive Computations

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

We formalize this notion in 3 components: 1. the Select record – purpose: select relevant elements from the input 2. the Constrain record – purpose:

constrain output elements to the list of relevant (selected) seen elements. 3. Restrictive__Comp locale – purpose: select relevant elements from the input and enforce constraints on the output.

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

We implement generalizations for model-type specific implementations to standard type constructors in Isabelle, including (e.g. group) lists, trees, multisets, finite sets, products, and finite maps. E.g. a model specific implementation of a group type can be generalized to a group list implementation. Additionally, we provide default implementations noSelect and noConstrain, which select no element and define no constraints on all inputs.

8.1 Select

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

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

We provide a few generalized type constructor implementations.

context

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

begin

definition *listS*::

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

where *listS* $\equiv$ ($\llbracket$ *select* = ($\lambda xs.$ concat ($\text{map } (\lambda x. \boxtimes_r x) xs$) $\rrbracket$)

definition *treeS* ::

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

where *treeS* $\equiv$ ($\llbracket$ *select* = ($\lambda x.$ $\boxtimes_{listS}(\text{inorder } x)$) $\rrbracket$)

end

context

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

begin

definition *multisetS* ::

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

where *multisetS* $\equiv$ ($\llbracket$ *select* = ($\lambda x.$ $\boxtimes_{listS\ r}(\text{sorted-list-of-multiset } x)$) $\rrbracket$)

definition *fsetS* ::

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

where *fsetS* $\equiv$ ($\llbracket$ *select* = ($\lambda x.$ $\boxtimes_{listS\ r}(\text{sorted-list-of-fset } x)$) $\rrbracket$)

end

context

fixes $ra :: ('a, 'c) \text{ Select}$
and $rb :: ('b, 'c) \text{ Select}$

begin

definition $prodS ::$

$(('a \times 'b), 'c) \text{ Select}$ **where**
 $prodS \equiv \llbracket select = (\lambda(a,b). \bowtie_{ra} a \ @ \ \bowtie_{rb} b) \rrbracket$

end

context

fixes $ra :: ('a::linorder, 'c) \text{ Select}$
and $rb :: ('b, 'c) \text{ Select}$

begin

definition $fmapS ::$

$(('a, 'b) \text{ fmap}, 'c) \text{ Select}$ **where**
 $fmapS \equiv \llbracket select = (\lambda fm. \bowtie_{listS} (prodS \ ra \ rb) \ (sorted\text{-}list\text{-}of\text{-}fmap \ fm)) \rrbracket$

end

definition $noSelect :: ('a, 'b) \text{ Select}$

where $noSelect \equiv \llbracket select = (\lambda x. []) \rrbracket$

lemma $noSelectEmpty[simp]: \bigwedge x. \bowtie_{noSelect} x = []$
 $\langle proof \rangle$

8.2 Constrain

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

record $('b, 'c) \text{ Constrain} = constrain :: 'b \text{ list} \Rightarrow 'c \Rightarrow bool(\mathbf{infixl} \ \langle \triangleright_1 \rangle \ 70)$

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

context

fixes $r :: ('b, 'c) \text{ Constrain}$

begin

definition $listC ::$

$('b, 'c \text{ list}) \text{ Constrain}$
where $listC \equiv \llbracket constrain = (\lambda ip \ op. list\text{-}all \ (\lambda x. ip \triangleright_r x) \ op) \rrbracket$

```

definition treeC ::
  ('b, 'c tree) Constrain
  where treeC  $\equiv \llbracket \text{constrain} = (\lambda ip\ op.\ ip \triangleright_{listC} (\text{inorder } op)) \rrbracket$ 

end

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

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

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

end

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

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

end

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

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

end

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

lemma noConstrainTrue[simp]:  $\bigwedge ip\ op.\ ip \triangleright_{noConstrain} op = True$ 
   $\langle proof \rangle$ 

```

8.3 restrict

```

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

```

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

```

definition restrict :: ('a  $\Rightarrow$  'c spmf)  $\Rightarrow$  'a  $\Rightarrow$  'c spmf where
  restrict  $\mathcal{A}$  arg  $\equiv$  do {
    res  $\leftarrow$   $\mathcal{A}$  arg;
    -::unit  $\leftarrow$  assert-spmf (( $\bowtie_{sel}$  arg)  $\triangleright_{con}$  res);
    return-spmf res
  }

```

end

8.4 Examples

```

locale examples
begin

```

examples for select

```

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

```

```

lemma  $\bowtie_{intS}$  (1::int) = [1::int]
   $\langle$ proof $\rangle$ 

```

```

lemma  $\bowtie_{listS\ intS}$  [1] = [1]
   $\langle$ proof $\rangle$ 

```

```

lemma  $\bowtie_{listS\ (listS\ intS)}$  [[1]] = [1]
   $\langle$ proof $\rangle$ 

```

```

lemma  $\bowtie_{prodS\ (listS\ intS)\ intS}$  ([1],0) = [1,0]
   $\langle$ proof $\rangle$ 

```

examples for constrain

```

definition intC :: (int, int) Constrain
  where intC  $\equiv$  ( $\Downarrow$ constrain = ( $\lambda ip\ op.$  sum-list ip = op))

```

```

lemma [0,1,2]  $\triangleright_{intC}$  3
   $\langle$ proof $\rangle$ 

```

```

lemma [0,1,2]  $\triangleright_{listC\ intC}$  [3,3,3]

```

```

    <proof>

lemma  $[0,1,2] \triangleright_{prodC} (listC\ intC)\ intC\ ([3,3,3],3)$ 
    <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

$g = s_0 [\wedge] c_0 \otimes \dots \otimes s_n [\wedge] c_n$, where $[s_0, \dots, s_n] = \text{seen}$ are the group values the algorithm was supplied with.

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

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

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

lemma *constrain-grp-is-in-carrier*:
assumes $\forall g \in \text{set seen-vec. } g \in \text{carrier } G$
and *constrain-grp seen-vec (g, c-vec)*
shows $g \in \text{carrier } G$
 $\langle \text{proof} \rangle$

end

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

sublocale *Algebraic-Algorithm* $\subseteq$ *Restrictive-Comp sel con* $\langle \text{proof} \rangle$

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 $\Rightarrow$ ('a $\times$ int list) *spmf*
where *A-id* = $(\lambda x. \text{return-spmf } (x, [1::\text{int}]))$

interpretation *Algebraic-Algorithm* $G \text{ groupS } \text{groupC}$
 $\langle \text{proof} \rangle$

lemma
assumes $g \in \text{carrier } G$
shows *restrict A-id g*

$= \mathcal{A}\text{-id } g$
 $\langle \text{proof} \rangle$

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

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

interpretation list-id : *Restrictive-Comp* (listS groupS) listC groupC $\langle \text{proof} \rangle$

lemma

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

We define some useful ML functions for the AGM.

$\text{lift_to_algebraicT}$ 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.

$\langle ML \rangle$

9.1 Example for the ML function

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

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

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

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

end

end

theory *KZG-def*

imports *Polynomial-Commitment-Schemes Primitives*

begin

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

10 KZG function definitions

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

locale *KZG* = *math-primitives*
begin

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

10.1 Setup

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

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

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

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

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

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

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

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

definition *coeffs-n* :: 'e mod-ring poly $\Rightarrow$ 'e mod-ring $\Rightarrow$ 'e mod-ring list $\Rightarrow$ nat $\Rightarrow$ 'e mod-ring list

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

One can imagine the summation of power_diff_sumr2 to be horizontal, in the sense that it computes the coefficients from 0 to degree $\varphi = n$, and adds (or could add) to each coefficient in every iteration: 0: 0 + 1: f1 + 2: f2*u + f2*x + 3: f3*u^2 + f3*u*x + f3*x^2 ... n: fn*u^(n-1) + ... fn*u^((n-1)-i)*x^i ... + fn*x^(n-1) we order it to compute the coefficients one by one (to compute vertical) 1: 0 + f1 + ... + fn*u^(n-1) + 2: 0 + f2 * x + ... + fn*u^((n-1)-1) * x + ... n: 0 + fn * x^(n-1)

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

definition *psi-of* :: 'e mod-ring poly $\Rightarrow$ 'e mod-ring $\Rightarrow$ 'e mod-ring poly

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

a wrapper around Setup that hands the srs to both parties

definition *key-gen* :: ('a ck $\times$ 'a vk) spmf

where *key-gen* = do {
 (α , PK) $\leftarrow$ 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 $\Rightarrow$ 'e mod-ring poly $\Rightarrow$ ('a commit $\times$ trapdoor) spmf
where *commit* PK φ = return-spmf (g-pow-PK-Prod PK φ , ()) — $g^{\hat{\varphi}}(\alpha)$

definition *verify-poly* :: 'a vk $\Rightarrow$ 'e mod-ring poly $\Rightarrow$ 'a commit $\Rightarrow$ trapdoor $\Rightarrow$ 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 $\Rightarrow$ trapdoor $\Rightarrow$ 'e mod-ring poly $\Rightarrow$ 'e mod-ring $\Rightarrow$ ('e mod-ring $\times$ 'a witness)
where *Eval* PK td φ i = (let $\psi = \psi\text{-of } \varphi$ i — 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 $\Rightarrow$ 'a commit $\Rightarrow$ 'e mod-ring $\Rightarrow$ ('e mod-ring $\times$ 'a witness) $\Rightarrow$ bool
where *verify-eval* PK C i val = (let (eval, w) = val
in (e w (PK!1 $\div_{G_p}$ ($\mathbf{g}_{G_p}^{\hat{G_p}} i$)) $\otimes_{G_T}$ ((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 $\Rightarrow$ bool
where *valid-poly* φ = (degree $\varphi \leq \text{max-deg}$)

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

definition *valid-eval* :: ('e mod-ring $\times$ 'a witness) $\Rightarrow$ bool
where *valid-eval* val = (let (-, w) = val in w $\in$ carrier G_p)

the KZG is a polynomial commitment scheme

sublocale abstract-polynomial-commitment-scheme key-gen commit verify-poly Eval
verify-eval
valid-poly valid-argument valid-eval <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]*: $\text{length } (\text{coeffs-n } \varphi \text{ u } q\text{-co } n) = n$
 $\langle \text{proof} \rangle$

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

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

lemma *sum-split*: $m \leq n \implies (\sum i \leq n. f \text{ i}) = (\sum i \leq m. f \text{ i}) + (\sum i \in \{m < .. < \text{Suc } n\}. f \text{ i})$
 $\langle \text{proof} \rangle$

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

lemma *degree-q-le- φ* : $\text{degree } (\psi\text{-of } \varphi \text{ u}) \leq \text{degree } \varphi$
 $\langle \text{proof} \rangle$

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

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

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

lemma *sum-horiz-to-vert*: $n \leq \text{degree } (\varphi :: 'e \text{ mod-ring poly}) \implies$
 $(\sum i \leq n. \text{poly.coeff } \varphi \text{ i } * (\sum j < i. u ^ (i - \text{Suc } j) * x ^ j))$
 $= (\sum i \leq n. (\sum j \in \{i < .. < \text{Suc } n\}. \text{poly.coeff } \varphi \text{ j } * u ^ (j - \text{Suc } i)) * x ^ i)$
 $\langle \text{proof} \rangle$

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

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

<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 `$\psi$ -of-ith-coeff-eq-sum-ith-coeff` lemma, show the coefficients of the result of `sum_horiz_to_vert` equal to the coefficients calculated by ψ -of-poly and thus showing $\text{poly } (\psi\text{-of-poly } \varphi \ u) \ x$ equal to the result `sum` of `sum_horiz_to_vert`.

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

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

lemma `eq-on-e`: $(e \ (\mathbf{g}_{G_p} \ \widehat{G_p} \ (\text{poly } (\psi\text{-of } \varphi \ i) \ \alpha))) \ (\mathbf{g}_{G_p} \ \widehat{G_p} \ \alpha \div_{G_p} \mathbf{g}_{G_p} \ \widehat{G_p} \ i))$
 $\otimes_{G_T} (e \ (\mathbf{g}_{G_p} \ \mathbf{g}_{G_p}) \ \widehat{G_T} \ (\text{poly } \varphi \ i))$
 $= e \ (\mathbf{g}_{G_p} \ \widehat{G_p} \ (\text{poly } \varphi \ \alpha)) \ \mathbf{g}_{G_p}$
<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 i th public parameter is actually the i th public parameter $(g^{\wedge(\alpha^i)})$

lemma `PK-i`: $i \leq t \implies \text{map } (\lambda t. \mathbf{g} \ \widehat{G_p} \ \alpha \ \wedge^t) \ [0..<t+1] \ ! \ i = \mathbf{g}_{G_p} \ \widehat{G_p} \ (\alpha^i)$
<proof>

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

lemma `g-pow-PK-Prod-to-fold[simp]`: $\text{degree } \varphi \leq t \implies g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g} \ \widehat{G_p} \ \alpha \ \wedge^t) \ [0..<t+1]) \ \varphi$
 $= \text{fold } (\lambda pk \ g. g \otimes_{G_p} (\mathbf{g}_{G_p} \ \widehat{G_p} \ (\alpha^{\wedge pk})) \ \widehat{G_p} \ (\text{poly.coeff } \varphi \ pk)) \ [0..<\text{Suc } (\text{degree } \varphi)] \ \mathbf{1}_{G_p}$
<proof>

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

lemma `poly-altdef-to-fold[symmetric]`: $n \leq \text{degree } \varphi \implies \mathbf{g}_{G_p} \ \widehat{G_p} \ (\sum_{i \leq n. \text{poly.coeff } \varphi \ i * \alpha^i})$

$= \text{fold } (\lambda n \ g. \ g \otimes_{G_p} \mathbf{g}_{G_p}^{\wedge_{G_p}} (\text{poly.coeff } \varphi \ n * \alpha^{\wedge n}))$

$[0..< \text{Suc } n] \ \mathbf{1}_{G_p}$
 $\langle \text{proof} \rangle$

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

lemma *g-pow-PK-Prod-correct: degree $\varphi \leq t$*
 $\implies \text{g-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p}^{\wedge_{G_p}} \alpha^{\wedge t}) [0..<t+1]) \ \varphi$
 $= \mathbf{g}_{G_p}^{\wedge_{G_p}} (\text{poly } \varphi \ \alpha)$
 $\langle \text{proof} \rangle$

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

theorem *KZG-correct: correct-eval*
 $\langle \text{proof} \rangle$

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)*
 $\implies$
 $\text{spmf game1 True} \leq \text{spmf game2 True}$
 $\langle \text{proof} \rangle$

lemma *unpack-bind-spmf: $X = X' \implies \text{bind-spmf } X \ Y = \text{bind-spmf } X' \ Y$*
 $\langle \text{proof} \rangle$

lemma *unpack-bind-spmf'*: $Y = Y' \implies \text{bind-spmf } X \ Y = \text{bind-spmf } X \ Y'$
 $\langle \text{proof} \rangle$

lemma *unpack-bind-spmf-fun*: $X = X' \implies \text{bind-spmf } X \ (\lambda y. f \ y) = \text{bind-spmf } X' \ (\lambda y. f \ y)$
 $\langle \text{proof} \rangle$

lemma *unpack-try-spmf*: $X = X' \implies \text{TRY } X \ \text{ELSE } Y = \text{TRY } X' \ \text{ELSE } Y$
 $\langle \text{proof} \rangle$

lemma *unpack-try-spmf'*: $Y = Y' \implies \text{TRY } X \ \text{ELSE } Y = \text{TRY } X \ \text{ELSE } Y'$
 $\langle \text{proof} \rangle$

lemma *spm-f-eqI'*: $X = X' \implies \text{spm-f } X \ Y = \text{spm-f } X' \ Y$
 $\langle \text{proof} \rangle$

12.1 SPMF True

lemma *return-spmf-assert*: $\text{TRY } \text{return-spmf } X \ \text{ELSE } \text{return-spmf } \text{False} = \text{TRY } \text{bind-spmf } (\text{assert-spmf } X) \ (\lambda-. \text{return-spmf } \text{True}) \ \text{ELSE } \text{return-spmf } \text{False}$
 $\langle \text{proof} \rangle$

lemma *bind-spmf-independent-return-spmf*:
 $\text{lossless-spmf } x \implies \text{bind-spmf } x \ (\lambda x. \text{return-spmf } y) = \text{return-spmf } y$
 $\langle \text{proof} \rangle$

lemma *bind-spmf-le*:
 $(\bigwedge x. \text{spm-f } (f \ x) \ \text{True} \leq \text{spm-f } (f' \ x) \ \text{True}) \implies \text{spm-f } (\text{bind-spmf } p \ f) \ \text{True} \leq \text{spm-f } (\text{bind-spmf } p \ f') \ \text{True}$
 $\langle \text{proof} \rangle$

lemma *try-spmf-eq*:
assumes $\text{spm-f } x \ \text{True} = \text{spm-f } x' \ \text{True}$
shows $\text{spm-f } (\text{TRY } x \ \text{ELSE } \text{return-spmf } \text{False}) \ \text{True} = \text{spm-f } (\text{TRY } x' \ \text{ELSE } \text{return-spmf } \text{False}) \ \text{True}$
 $\langle \text{proof} \rangle$

lemma *try-spmf-le*:
assumes $\text{spm-f } x \ \text{True} \leq \text{spm-f } x' \ \text{True}$
shows $\text{spm-f } (\text{TRY } x \ \text{ELSE } \text{return-spmf } \text{False}) \ \text{True} \leq \text{spm-f } (\text{TRY } x' \ \text{ELSE } \text{return-spmf } \text{False}) \ \text{True}$
 $\langle \text{proof} \rangle$

lemma *try-spmf-true-else-false-le*: $\text{spm-f } (\text{TRY } X \ \text{ELSE } \text{return-spmf } \text{False}) \ \text{True} \leq \text{spm-f } X \ \text{True}$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *assert-imp*: $(X \longrightarrow X') \implies$
 $\text{spmf } (\text{bind-spmf } (\text{assert-spmf } X) (\lambda \cdot . Y)) \text{ True}$
 $\leq \text{spmf } (\text{bind-spmf } (\text{assert-spmf } X') (\lambda \cdot . Y)) \text{ True}$
 $\langle \text{proof} \rangle$

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

lemma *assert-based-eq*:
assumes $x \implies y = y'$
shows $\text{bind-spmf } (\text{assert-spmf } x) (\lambda \cdot . y)$
 $= \text{bind-spmf } (\text{assert-spmf } x) (\lambda \cdot . y')$
 $\langle \text{proof} \rangle$

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

lemma *assert-collapse*: $\text{bind-spmf } (\text{assert-spmf } X) (\lambda \cdot . \text{bind-spmf } (\text{assert-spmf } Y) (\lambda \cdot . Z))$
 $= \text{bind-spmf } (\text{assert-spmf } (X \wedge Y)) (\lambda \cdot . Z)$
 $\langle \text{proof} \rangle$

lemma *assert-cong*: $X = Y \implies \text{rel-spmf } (=) (\text{assert-spmf } X) (\text{assert-spmf } Y)$
 $\langle \text{proof} \rangle$

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

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

12.2 Sample uniform set

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

lemma *spmf-sample-uniform-set*: $\text{spmf } (\text{sample-uniform-set } k \ n) \ x$
 $= \text{indicator } \{x. x \subseteq \{..<n\} \wedge \text{card } x = k\} \ x / (n \text{ choose } k)$
 $\langle \text{proof} \rangle$

lemma *weight-sample-uniform-set*: $\text{weight-spmf } (\text{sample-uniform-set } k \ n) = \text{of-bool}$

$(k \leq n)$
 $\langle proof \rangle$

lemma *weight-sample-uniform-set-k-0* [simp]: *weight-spmf* (*sample-uniform-set* 0 *n*) = 1
 $\langle proof \rangle$

lemma *weight-sample-uniform-set-n-0* [simp]: *weight-spmf* (*sample-uniform-set* *k* 0) = of_bool (*k*=0)
 $\langle proof \rangle$

lemma *weight-sample-uniform-set-k-le-n* [simp]: $k \leq n \implies \text{weight-spmf } (\text{sample-uniform-set } k \ n) = 1$
 $\langle proof \rangle$

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

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

12.3 Sample uniform list

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

lemma *spmf-sample-uniform-list*: *spmf* (*sample-uniform-list* *k* *n*) *x*
= *indicator* $\{x. \text{set } x \subseteq \{..<n\} \wedge \text{length } x = k\}$ *x* / (*n*^*k*)
 $\langle proof \rangle$

lemma *weight-sample-uniform-list*: *weight-spmf* (*sample-uniform-list* *k* *n*) = of_bool
(*k*=0 $\vee$ 0<*n*)
 $\langle proof \rangle$

lemma *weight-sample-uniform-list-k-0* [simp]: *weight-spmf* (*sample-uniform-list* 0 *n*) = 1
 $\langle proof \rangle$

lemma *weight-sample-uniform-list-n-0* [simp]: *weight-spmf* (*sample-uniform-list* *k* 0) = of_bool (*k*=0)
 $\langle proof \rangle$

lemma *weight-sample-uniform-list-k-le-n* [simp]: $k < n \implies \text{weight-spmf } (\text{sample-uniform-list } k \ n) = 1$
 $\langle proof \rangle$

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

$\longleftrightarrow (k=0 \vee 0 < n)$
 $\langle proof \rangle$

lemma *set-spmf-sample-uniform-list* [simp]: $set\text{-}spmf\ (sample\text{-}uniform\text{-}list\ k\ n) = \{x. set\ x \subseteq \{..<n\} \wedge length\ x = k\}$
 $\langle proof \rangle$

the two following lemmas are helping lemmas for `Cons_random_list_split`

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

lemma *set-spmf-lhs-imp-length-gr-0*: $xs \in set\text{-}spmf\ (map\text{-}spmf\ ((\#)\ x)\ (sample\text{-}uniform\text{-}list\ k\ p)) \implies length\ xs > 0$
 $\langle proof \rangle$

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

lemma *Cons-random-list-split*:
assumes $p > 1$
shows $do\ \{x \leftarrow sample\text{-}uniform\ p;$
 $map\text{-}spmf\ ((\#)\ x)\ (sample\text{-}uniform\text{-}list\ k\ p)\} = sample\text{-}uniform\text{-}list\ (k+1)$
 p
(is ?lhs = ?rhs)
 $\langle proof \rangle$

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 $sample\text{-}uniform\text{-}list\ (k+1)\ p =$
 $do\ \{x \leftarrow sample\text{-}uniform\ p;$
 $xs \leftarrow (sample\text{-}uniform\text{-}list\ k\ p);$
 $return\text{-}spmf\ (x\#xs)\}$
(is ?lhs = ?rhs)
 $\langle proof \rangle$

12.4 Sample uniform polynomial

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

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

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

<proof>

end

theory *KZG-poly-bind*

imports *KZG-correct CryptHOL-ext tDL-assumption*

Berlekamp-Zassenhaus.Finite-Field-Factorization Elimination-Of-Repeated-Factors.ERF-Algorithm

begin

13 Polynomial Binding of the KZG

We show that the KZG is polynomial binding for every polynomial of degree $\leq \max_deg$. 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 $\circ$ int pow-mod-ring G_p*

<proof>

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

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

```

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

```

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

```

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

```

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

```

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

```

where

```

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

```

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

```

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

```

where

```

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

```

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

lemma *finite-field-factorization-roots*:

fixes $\varphi :: 'e \text{ mod-ring poly}$
assumes *sf-f*: *square-free* φ
and *us*: *finite-field-factorization* $\varphi = (c, us)$
shows $\text{poly } \varphi \alpha = 0 \longleftrightarrow c=0 \vee (\exists u \in \text{set } us. \text{poly } u \alpha = 0)$
 $\langle \text{proof} \rangle$

lemma *root-imp-deg-1*:

assumes *monic* $(u :: 'e \text{ mod-ring poly}) \wedge \text{irreducible } u$
shows $(\exists x. \text{poly } u x = 0) \longleftrightarrow \text{degree } u = 1$
 $\langle \text{proof} \rangle$

lemma *poly-eq0-is-find- α -sf-eq- α* :

assumes $\varphi \neq 0 \wedge \text{square-free } \varphi$
shows $\text{poly } \varphi \alpha = 0 \implies \text{find-}\alpha\text{-square-free } (\mathbf{g}_{G_p} \hat{}_{G_p} \alpha) \varphi = \alpha$
 $\langle \text{proof} \rangle$

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

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

13.2.1 Literal helping lemmas

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

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

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

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

lemma *helping-3- α -is-found*: $\varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{t})) [0..<\text{max-deg}+1]) \varphi) \wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{t})) [0..<\text{max-deg}+1]) \varphi')$
 $\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}))) (\varphi - \varphi') =$
 $(\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring})$
 $\longleftrightarrow \varphi \neq \varphi' \wedge \text{valid-poly } \varphi \wedge \text{valid-poly } \varphi'$
 $\wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{t})) [0..<\text{max-deg}+1]) \varphi) \wedge (C = g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{t})) [0..<\text{max-deg}+1]) \varphi')$
 $\wedge (\text{find-}\alpha (\mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}))) (\varphi - \varphi') =$
 $(\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring})$
 $\wedge \text{of-int-mod-ring } (\text{int } \alpha) = \text{find-}\alpha ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} ((\text{of-int-mod-ring } (\text{int } \alpha))::'e \text{ mod-ring}) \hat{t})) [0..<\text{max-deg}+1])!1) (\varphi - \varphi')$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

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 $cs.bind\text{-}game\ \mathcal{A} = t\text{-}DL\text{-}G_p.game\ (stronger\text{-}bind\text{-}reduction\ \mathcal{A})$
<proof>

lemma *t-DL-advantage-stronger-red-le-red:*

$t\text{-}DL\text{-}G_p.advantage\ (stronger\text{-}bind\text{-}reduction\ \mathcal{A}) \leq t\text{-}DL\text{-}G_p.advantage\ (bind\text{-}reduction\ \mathcal{A})$
<proof>

theorem *polynomial-binding:*

$poly\text{-}bind\text{-}advantage\ \mathcal{A} \leq t\text{-}DL\text{-}G_p.advantage\ (bind\text{-}reduction\ \mathcal{A})$
<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 G_p

sublocale *t-SDH- G_p : t-SDH G_p max-deg of-int-mod-ring $\circ$ 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  $\Rightarrow$ 
  ('a, 'e) t-SDH.adversary
where
  eval-bind-reduction  $\mathcal{A}$  PK = do {
    (C, i,  $\varphi$ -of-i, w-i,  $\varphi'$ -of-i, w'-i)  $\leftarrow$   $\mathcal{A}$  PK;
    return-spmf (-i::'e mod-ring, (w-i  $\div_{G_p}$  w'-i)  $\hat{\wedge}_{G_p}$  (1 / ( $\varphi'$ -of-i -  $\varphi$ -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  $\Rightarrow$ 
  ('a, 'e) t-SDH.adversary
where
  ext-eval-bind-reduction  $\mathcal{A}$  PK = do {
    (C, i, v, w, v', w')  $\leftarrow$   $\mathcal{A}$  PK;
    - :: unit  $\leftarrow$  assert-spmf (v  $\neq$  v'
       $\wedge$  valid-argument i
       $\wedge$  valid-eval (v, w)
       $\wedge$  valid-eval (v', w')
       $\wedge$  verify-eval PK C i (v, w)
       $\wedge$  verify-eval PK C i (v', w')) ;
    return-spmf (-i::'e mod-ring, (w  $\div_{G_p}$  w')  $\hat{\wedge}_{G_p}$  (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  $\varphi$ -of-i  $\neq$   $\varphi'$ -of-i  $\wedge$  w-i  $\neq$  w'-i
   $\wedge$  w-i  $\in$  carrier  $G_p$   $\wedge$  w'-i  $\in$  carrier  $G_p$ 
   $\wedge$  verify-eval ((map ( $\lambda t$ .  $\mathbf{g}_{G_p} \hat{\wedge}_{G_p} ((\alpha) \hat{\wedge} t)$ ) [0.. $\text{max-deg}+1$ ])) C i ( $\varphi$ -of-i,
w-i)
   $\wedge$  verify-eval ((map ( $\lambda t$ .  $\mathbf{g}_{G_p} \hat{\wedge}_{G_p} ((\alpha) \hat{\wedge} t)$ ) [0.. $\text{max-deg}+1$ ])) C i ( $\varphi'$ -of-i,
w'-i)
shows  $\mathbf{g}_{G_p} \hat{\wedge}_{G_p} (1 / (\alpha + (-i)))$ 
```

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

$\langle proof \rangle$

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*: $v \neq v'$

$$\begin{aligned} & \wedge \text{valid-argument } i \\ & \wedge \text{valid-eval } (v, w) \\ & \wedge \text{valid-eval } (v', w') \\ & \wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} (\alpha \hat{\wedge} t)) [0..<max-deg+1]) \\ C\ i\ (v, w) & \\ & \wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} (\alpha \hat{\wedge} t)) [0..<max-deg+1]) \\ C\ i\ (v', w') & \\ \longleftrightarrow & \\ & v \neq v' \wedge w \neq w' \\ & \wedge \text{valid-argument } i \\ & \wedge \text{valid-eval } (v, w) \\ & \wedge \text{valid-eval } (v', w') \\ & \wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} (\alpha \hat{\wedge} t)) [0..<max-deg+1]) \\ C\ i\ (v, w) & \\ & \wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} (\alpha \hat{\wedge} t)) [0..<max-deg+1]) \\ C\ i\ (v', w') & \\ \langle proof \rangle & \end{aligned}$$

lemma *helping-1*: $\varphi-of-i \neq \varphi'-of-i \wedge w-i \neq w'-i$

$$\begin{aligned} & \wedge \text{valid-argument } i \\ & \wedge w-i \in \text{carrier } G_p \wedge w'-i \in \text{carrier } G_p \\ & \wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} ((\alpha) \hat{\wedge} t)) [0..<max-deg+1]))\ C\ i\ (\varphi-of-i, \\ w-i) & \\ & \wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} ((\alpha) \hat{\wedge} t)) [0..<max-deg+1]))\ C\ i\ (\varphi'-of-i, \\ w'-i) & \\ & \wedge \mathbf{g}_{G_p} \hat{\wedge}_{G_p} (1/(\alpha + (-i))) \\ & = (w-i \div_{G_p} w'-i) \hat{\wedge}_{G_p} (1 / (\varphi'-of-i - \varphi-of-i)) \\ \longleftrightarrow & \\ & \varphi-of-i \neq \varphi'-of-i \wedge w-i \neq w'-i \\ & \wedge \text{valid-argument } i \\ & \wedge w-i \in \text{carrier } G_p \wedge w'-i \in \text{carrier } G_p \\ & \wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} ((\alpha) \hat{\wedge} t)) [0..<max-deg+1]))\ C\ i\ (\varphi-of-i, \\ w-i) & \\ & \wedge \text{verify-eval } ((\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\wedge}_{G_p} ((\alpha) \hat{\wedge} t)) [0..<max-deg+1]))\ C\ i\ (\varphi'-of-i, \\ w'-i) & \\ \langle proof \rangle & \end{aligned}$$

14.4 Proof

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

<proof>

theorem *eval-bind-game-eq-t-SDH-strong-ext-red:*

shows *eval-bind-game $\mathcal{A} = t\text{-SDH-}G_p.\text{game } (ext\text{-eval-bind-reduction } \mathcal{A})$*
<proof>

lemma *overestimate-reductions: $spmf (t\text{-SDH-}G_p.\text{game } (ext\text{-eval-bind-reduction } \mathcal{A}))$*

True

$\leq spmf (t\text{-SDH-}G_p.\text{game } (eval\text{-bind-reduction } \mathcal{A})) \text{ True}$

(is $spmf ?lhgame \text{ True} \leq spmf ?rhgame \text{ True}$)

<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: $eval\text{-bind-advantage } \mathcal{A} \leq t\text{-SDH-}G_p.\text{advantage } (eval\text{-bind-reduction } \mathcal{A})$*

<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*

$\Rightarrow AGM\text{-knowledge-soundness-adversary1}$

lift-to-algebraicT ('state', 'a ck', 'e mod-ring', 'e evaluation', 'a witness) *knowledge-soundness-adversary2*

$G_p \Rightarrow \text{AGM-knowledge-soundness-adversary2}$

type-synonym ('e', 'state', 'a') *knowledge-soundness-adversary2-AGM*
 $= ('a' \text{ ck} \times 'state') \Rightarrow ('e' \text{ argument} \times ('e' \text{ evaluation} \times ('a' \text{ witness} \times \text{int list})))$
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* =
 $('a' \text{ commit} \times \text{int list}) \Rightarrow$
 $('e' \text{ mod-ring poly} \times \text{unit}) \text{ spmf}$

restrict for AGM adversaries 1 and 2

realized by the following two interpretations:

interpretation *AGM1: Algebraic-Algorithm* $G_p \text{ listS } G_p.\text{groupS} \text{ prodC } G_p.\text{groupC}$
noConstrain
 $\langle \text{proof} \rangle$

interpretation *AGM2: Algebraic-Algorithm* $G_p \text{ prodS } (\text{listS } G_p.\text{groupS}) \text{ noSelect}$
 $\text{prodC } \text{noConstrain } (\text{prodC } \text{noConstrain } G_p.\text{groupC})$
 $\langle \text{proof} \rangle$

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

$\Rightarrow ('e', 'state', 'a') \text{ knowledge-soundness-adversary2-AGM} \Rightarrow ('a', 'e') \text{ extractor} \Rightarrow$
bool spmf

where *knowledge-soundness-game-AGM* $\mathcal{A1} \mathcal{A2} \mathcal{E} = \text{TRY do } \{$
 $(ck, vk) \leftarrow \text{key-gen};$
 $((c, cvec), \sigma) \leftarrow \text{AGM1.restrict } \mathcal{A1} \text{ ck};$
 $(p, td) \leftarrow \mathcal{E} (c, cvec);$
 $(i, p-i, w, wvec) \leftarrow \text{AGM2.restrict } \mathcal{A2} (ck, \sigma);$
 $\text{let } (p-i', w') = \text{Eval } ck \text{ td } p \text{ } i;$
 $\text{return-spmf } (\text{verify-eval } vk \text{ } c \text{ } i \text{ } (p-i, w) \wedge p-i \neq p-i' \wedge \text{valid-argument } i \wedge$
 $\text{valid-eval } (p-i, w))$
 $\} \text{ ELSE return-spmf False}$

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

definition *knowledge-soundness-reduction*

```

:: ('a, 'e) extractor  $\Rightarrow$  ('state, 'a) AGM-knowledge-soundness-adversary1
 $\Rightarrow$  ('e, 'state, 'a) knowledge-soundness-adversary2-AGM
 $\Rightarrow$  ('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), $\sigma$ )  $\leftarrow$  AGM1.restrict  $\mathcal{A}1$  ck;
  (p,td)  $\leftarrow$   $\mathcal{E}$  (c,cvec);
  (i, p-i, w, wvec)  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
  let (p-i',w') = Eval ck td p i;
  return-spmf (c, i, p-i, w, p-i', w')}

```

Extractor definition

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

```

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

```

15.1 Helping definitions

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

definition *knowledge-soundness-reduction-ext*

```

:: ('a, 'e) extractor  $\Rightarrow$  ('state, 'a) AGM-knowledge-soundness-adversary1
 $\Rightarrow$  ('e, 'state, 'a) knowledge-soundness-adversary2-AGM
 $\Rightarrow$  ('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), $\sigma$ )  $\leftarrow$  AGM1.restrict  $\mathcal{A}1$  ck;
  (p,td)  $\leftarrow$   $\mathcal{E}$  (c,cvec);
  (i, p-i, w, wvec)  $\leftarrow$  AGM2.restrict  $\mathcal{A}2$  (ck, $\sigma$ );
  - :: unit  $\leftarrow$  assert-spmf (valid-eval (p-i, w));
  let (p-i',w') = Eval ck td p i;
  return-spmf (c, i, p-i, w, p-i', w')}

```

15.2 Helping lemmas

proof related helping lemmas

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

```

let ck = map ( $\lambda t. \mathbf{g}_{G_p} \widehat{\phantom{t}}_{G_p} (\alpha \widehat{t})$ ) [0.. $\text{max-deg}+1$ ];
vk = map ( $\lambda t. \mathbf{g}_{G_p} \widehat{\phantom{t}}_{G_p} (\alpha \widehat{t})$ ) [0.. $\text{max-deg}+1$ ];
(p-i',w') = Eval ck td (Poly (map of-int-mod-ring cvec)) i
in
length ck = length (cvec::int list)

```

```

    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ verify-eval vk c i (p-i,w)
    ∧ p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
  ↔
    length ck = length cvec
    ∧ c = fold (λ i acc. acc ⊗ ck!i [⌈ (cvec!i)) [0..<length ck] 1
    ∧ p-i ≠ p-i'
    ∧ valid-argument i
    ∧ valid-eval (p-i,w)
    ∧ valid-eval (p-i', w')
    ∧ verify-eval vk c i (p-i, w)
    ∧ verify-eval vk c i (p-i', w')
⟨proof⟩

```

lemma *knowledge-soundness-game-alt-def:*
knowledge-soundness-game-AGM A1 A2 E =
eval-bind-game (knowledge-soundness-reduction-ext E A1 A2)
 ⟨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 A A')) True*
≤ spmf (eval-bind-game (knowledge-soundness-reduction E A 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 A1 A2 E) True
≤ t-SDH-G_p.advantage (eval-bind-reduction (knowledge-soundness-reduction E A1 A2))
 ⟨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 $\Rightarrow$ ('a vk, 'e argument, 'state) *eval-hiding-adversary1*
where
weak-eval-hiding-adversary1 $\mathcal{A}$ vk = do {
 (I, σ) $\leftarrow$ $\mathcal{A}$ vk;
 - :: unit $\leftarrow$ *assert-spmf* (length I = max-deg $\wedge$ distinct I);
 return-spmf (I, σ)
}

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

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

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

definition *weak-eval-hiding-advantage* :: ('a vk, 'e argument, 'state) *eval-hiding-adversary1*
 $\Rightarrow$
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ real
where *weak-eval-hiding-advantage* $\mathcal{A1}$ $\mathcal{A2}$
 $\equiv$ *spmf* (*weak-eval-hiding-game* $\mathcal{A1}$ $\mathcal{A2}$) 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 $\circ$ int pow-mod-ring G_p

<proof>

We instantiate the t-DL game for the group G_p

sublocale $t\text{-DL-}G_p$: $t\text{-DL } G_p \text{ max-deg of-int-mod-ring} \circ \text{int pow-mod-ring } G_p$
<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 $\text{interpolate-on} :: ('e \text{ mod-ring} \times 'a) \text{ list} \Rightarrow 'e \text{ mod-ring} \Rightarrow 'a$ **where**
 $\text{interpolate-on } xs\text{-}ys \alpha = \text{do } \{$
 $\text{let } xs = \text{map fst } xs\text{-}ys;$
 $\text{let } \text{lagrange-exp} = \text{map } (\lambda(xj, yj). yj \hat{=}_{G_p} (\text{poly } (\text{lagrange-basis-poly } xs \ xj) \ \alpha))$
 $xs\text{-}ys;$
 $\text{fold } (\lambda x \text{ prod. prod} \otimes_{G_p} x) \text{ lagrange-exp } \mathbf{1} \}$

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

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

lemma $\text{set-filter-distinct: set } (\text{filter-distinct } xs \ ys) = \{x. x \in \text{set } xs \wedge x \notin \text{set } ys\}$
<proof>

lemma $\text{length-filter-distinct:}$
assumes $\text{card } (\text{set } xs) > \text{card } (\text{set } ys)$
shows $\text{length } (\text{filter-distinct } xs \ ys) > 0$
<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 $\text{PickDistinct} :: 'e \text{ argument list} \Rightarrow 'e \text{ argument}$
where $\text{PickDistinct } I = \text{hd } (\text{filter-distinct } (\text{map } (\text{of-int-mod-ring}::\text{int} \Rightarrow 'e \text{ mod-ring}) [0..<\text{CARD}('e)]) \ I)$

lemma $\text{card-map-card: card } (\text{set } (\text{map } (\text{of-int-mod-ring}::\text{int} \Rightarrow 'e \text{ mod-ring}) [0..<\text{CARD}('e)]))$
 $= \text{CARD}('e)$
<proof>

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

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

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

```

16.2 Hiding proof

16.2.1 Helping lemmas

lemma PK1[simp]: map ($\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{} t)$) [$0..<\text{max-deg}+1$] $!1$ = $\mathbf{g}_{G_p} \hat{}_{G_p} \alpha$
 <proof>

The "find" in the tDL reduction is successful.

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

lemma exchange-lem: length I = max-deg $\wedge$ distinct $I \wedge \alpha \in \text{set } (\text{PickDistinct } I \# I)$

$\longleftrightarrow \text{length } I = \text{max-deg} \wedge \text{distinct } I \wedge \alpha \in \text{set } (\text{PickDistinct } I \# I)$
 $\wedge \alpha = (\text{case } (\text{find } (\lambda x. \mathbf{g}_{G_p} \hat{} x = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{} t)) [0..<\text{max-deg}+1]!1)$
 $(\text{PickDistinct } I \# I)) \text{ of Some } x \Rightarrow x \mid \text{None} \Rightarrow 1)$
 $\langle \text{proof} \rangle$

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

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

lemma *eval-on-lagrange-basis*: *poly (lagrange-interpolation-poly xs-ys) x* $\equiv$ *(let*
 $\text{xs} = \text{map fst xs-ys}$
 $\text{in sum-list } (\text{map } (\lambda (xj,yj). \text{yj} * (\text{poly } (\text{lagrange-basis-poly xs } xj) x)) \text{ xs-ys})$
 $(\text{is ?lhs} \equiv \text{?rhs})$
 $\langle \text{proof} \rangle$

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

lemma *fold-on- G_p -is-sum-list*: *fold* $(\lambda x \text{ prod. prod } \otimes_{G_p} x) (\text{map } (\lambda x. \mathbf{g}_{G_p} \hat{}_{G_p} f x) xs) (\mathbf{g}_{G_p} \hat{}_{G_p} z)$
 $= \mathbf{g}_{G_p} \hat{}_{G_p} z \otimes_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} (\text{sum-list } (\text{map } f xs))$
 $\langle \text{proof} \rangle$

The two prior lemmas are pulled together to show that `interpolate_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* $\leq \text{max-deg}+1$
shows *interpolate-on* $(\text{map } (\lambda(x,y).(x, \mathbf{g}_{G_p} \hat{}_{G_p} y)) \text{ xs-ys}) \alpha$
 $= \mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly } (\text{lagrange-interpolation-poly xs-ys}) \alpha)$
 $\langle \text{proof} \rangle$

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

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

lemma *interpolate-on-is-Commit*:
assumes *dist*: *distinct (map fst xs-ys)*
and *length-xs-ys*: *length xs-ys* $\leq \text{max-deg}+1$

shows *interpolate-on* (*map* ($\lambda(x,y).(x, \mathbf{g}_{G_p} \hat{}_{G_p} y)$) *xs-ys*) $\alpha = g\text{-pow-PK-Prod}$
 (*map* ($\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{t})$) [$0..<max\text{-deg} + 1$]) (*lagrange-interpolation-poly*
xs-ys)
<proof>

lemma *split-pow-div- G_p* :
shows $\mathbf{g}_{G_p} \hat{}_{G_p} (y/x) = (\mathbf{g}_{G_p} \hat{}_{G_p} y) \hat{}_{G_p} (1/x)$
<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* $\leq max\text{-deg} + 1$
and $\alpha \notin \text{set } (\text{map fst } (tl \text{ xs-ys}))$
shows *map* ($\lambda i. (\text{Eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{t})) [0..<max\text{-deg} + 1]) () (\text{lagrange-interpolation-poly}$
xs-ys) *i*)) (*map fst (tl xs-ys)*)
 = *map* ($\lambda(x,y). (y, (\mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly } (\text{lagrange-interpolation-poly } \text{xs-ys}) \alpha) \div_{G_p}$
 $\mathbf{g}_{G_p} \hat{}_{G_p} y) \hat{}_{G_p} (1/(\alpha - x)))$) (*tl xs-ys*)
<proof>

corollary *witness-calc-helper*:
assumes *length I* = *max-deg* $\wedge$ *distinct I*
and $\alpha \notin \text{set } I$
and *evals*
 $\in \text{set-spmf}$
 (*map-spmf* (*map* ($\lambda x. \text{of-int-mod-ring } (\text{int } x)$))
 (*sample-uniform-list* (*max-deg* + 1) (*order G_p*)))
shows *map* ($\lambda i. (\text{Eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{}_{G_p} (\alpha \hat{t})) [0..<max\text{-deg} + 1]) () (\text{lagrange-interpolation-poly}$
 (*zip* (*PickDistinct I # I*) *evals*)) *i*)) *I*
 = *map2* ($\lambda x y. (y, (\mathbf{g}_{G_p} \hat{}_{G_p} (\text{poly } (\text{lagrange-interpolation-poly } (\text{zip } (\text{PickDistinct}$
I # I) *evals*)) $\alpha) \div_{G_p} \mathbf{g}_{G_p} \hat{}_{G_p} y) \hat{}_{G_p} (1/(\alpha - x)))$) *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* $\circ$ *int:: nat $\Rightarrow$ 'e mod-ring*)
 $\{0..<CARD ('e)\}$
<proof>

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

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

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

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

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

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

definition *game1* :: ('a vk, 'e argument, 'state) eval-hiding-adversary1 $\Rightarrow$
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ bool spmf **where**
game1 $\mathcal{A}1 \mathcal{A}2 = \text{TRY do } \{$
 $(\alpha, PK) \leftarrow \text{Setup};$
 $(I, \sigma) \leftarrow \mathcal{A}1 PK;$
 $- :: \text{unit} \leftarrow \text{assert-spmf } (\text{length } I = \text{max-deg} \wedge \text{distinct } I);$
 $\text{let } i = \text{PickDistinct } I;$
 $\text{evals} \leftarrow \text{map-spmf } (\text{map } (\text{of-int-mod-ring} \circ \text{int}::\text{nat} \Rightarrow 'e \text{ mod-ring})) (\text{sample-uniform-list}$
 $(\text{max-deg}+1) (\text{order } G_p));$
 $\text{let } \varphi = \text{lagrange-interpolation-poly } (\text{zip } (i\#I) \text{ evals});$
 $\text{let } \text{exp-evals} = \text{map } (\lambda i. \mathbf{g}_{G_p} \hat{\mathbf{g}}_{G_p} i) \text{ evals};$
 $\text{let } C = \text{interpolate-on } (\text{zip } (i\#I) \text{ exp-evals}) \alpha;$
 $\text{let } W = \text{map } (\lambda j. \text{Eval } PK () \varphi j) I;$
 $\varphi' \leftarrow \mathcal{A}2 PK \sigma C I W;$
 $\text{return-spmf } (\varphi = \varphi') \}$ **ELSE** *return-spmf False*

game1 with the reduction adversaries emulation instead of CreateWitness

definition *game2* :: ('a vk, 'e argument, 'state) eval-hiding-adversary1 $\Rightarrow$
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2 $\Rightarrow$ bool spmf **where**
game2 $\mathcal{A}1 \mathcal{A}2 = \text{TRY do } \{$

```

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

```

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

definition game2-w-assert :: ('a vk, 'e argument, 'state) eval-hiding-adversary1 $\Rightarrow$

```

('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2  $\Rightarrow$  bool spmf where
game2-w-assert  $\mathcal{A}1$   $\mathcal{A}2 = \text{TRY do } \{$ 
( $\alpha$ , PK)  $\leftarrow$  Setup;
( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$  PK;
- :: unit  $\leftarrow$  assert-spmf (length  $I = \text{max-deg} \wedge \text{distinct } I$ );
let  $i = \text{PickDistinct } I$ ;
evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
(order  $G_p$ ));
let  $\varphi = \text{lagrange-interpolation-poly } (\text{zip } (i\#I) \text{ evals})$ ;
let  $C = \text{interpolate-on } (\text{zip } (i\#I) (\text{map } (\lambda i. \mathbf{g} \hat{\ } i) \text{ evals})) \ \alpha$ ;
let withn-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  $\sigma$   $C$   $I$  withn-tupel;
-::unit  $\leftarrow$  assert-spmf ( $\varphi = \varphi'$ );
return-spmf (hd evals = poly  $\varphi' i$ ) } ELSE return-spmf False

```

game2_w_assert without the assert.

definition game2-wo-assert :: ('a vk, 'e argument, 'state) eval-hiding-adversary1

```

 $\Rightarrow$ 
('e mod-ring, 'a vk, 'a commit, 'e argument, 'e evaluation, 'a witness, 'state)
eval-hiding-adversary2  $\Rightarrow$  bool spmf where
game2-wo-assert  $\mathcal{A}1$   $\mathcal{A}2 = \text{TRY do } \{$ 
( $\alpha$ , PK)  $\leftarrow$  Setup;
( $I, \sigma$ )  $\leftarrow$   $\mathcal{A}1$  PK;
let  $i = \text{PickDistinct } I$ ;
evals  $\leftarrow$  map-spmf (map (of-int-mod-ring  $\circ$  int)) (sample-uniform-list (max-deg+1)
(order  $G_p$ ));
let  $\varphi = \text{lagrange-interpolation-poly } (\text{zip } (i\#I) \text{ evals})$ ;
( $\alpha$ , PK)  $\leftarrow$  Setup;
let  $C = \text{interpolate-on } (\text{zip } (i\#I) (\text{map } (\lambda i. \mathbf{g} \hat{\ } i) \text{ evals})) \ \alpha$ ;

```

$\text{let } \text{witn-tupel} = \text{map } (\lambda(x,y). (y, (C \div_{G_p} \mathbf{g}_{G_p} \hat{G}_p y) \hat{G}_p (1/(\alpha-x)))) (\text{zip } I$
 $(\text{tl evals}));$
 $\varphi' \leftarrow \mathcal{A}2 \text{ PK } \sigma \ C \ I \ \text{witn-tupel};$
 $\text{return-spmf } (\text{hd evals} = \text{poly } \varphi' \ i) \} \text{ ELSE return-spmf False}$

lemma *PickDistinct-Prop*: $\text{length } a = \text{max-deg} \wedge \text{distinct } a$
 $\implies \text{distinct } (\text{PickDistinct } a \# a) \wedge \text{length } (\text{PickDistinct } a \# a) = \text{max-deg} + 1$
 $\langle \text{proof} \rangle$

corollary *PickDistinct-Prop'*: $\text{length } a = \text{max-deg} \wedge \text{distinct } a \wedge \text{valid-poly } \varphi$
 $\implies \text{distinct } (\text{PickDistinct } a \# a) \wedge \text{length } (\text{PickDistinct } a \# a) = \text{max-deg} + 1$
 $\langle \text{proof} \rangle$

lemma *distinct-map-fst-zip*: $\text{distinct } a \implies \text{distinct } (\text{map fst } (\text{zip } a \ b))$
 $\langle \text{proof} \rangle$

lemma *card-eq-ord*: $\text{CARD}('e) = \text{order } G_p$
 $\langle \text{proof} \rangle$

Step 1 of the proof.

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

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

Step 2 of the proof:

lemma *fundamental-lemma-game1-game2*:
assumes $\text{lossless-}\mathcal{A}1: \bigwedge PK . \text{lossless-spmf } (\mathcal{A}1 \ \text{PK})$
and $\text{lossless-}\mathcal{A}2: \bigwedge PK \ \sigma \ C \ I \ W . \text{lossless-spmf } (\mathcal{A}2 \ \text{PK } \sigma \ C \ I \ W)$
shows $\text{spmf } (\text{game1 } \mathcal{A}1 \ \mathcal{A}2) \ \text{True} \leq \text{spmf } (\text{game2 } \mathcal{A}1 \ \mathcal{A}2) \ \text{True} + t\text{-DL-}G_p.\text{advantage}$
 $(\text{reduction-tDL } \mathcal{A}1)$
 $\langle \text{proof} \rangle$

Step 3 of the proof:

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

Finally we assemble all proof steps for the weak hiding theorem

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

```

and lossless-A2:  $\bigwedge PK \ \sigma \ C \ I \ W . \text{lossless-spmf} \ (A2 \ PK \ \sigma \ C \ I \ W)$ 
shows weak-eval-hiding-advantage A1 A2
   $\leq DL\text{-}G_p.\text{advantage} \ (\text{reduction} \ A1 \ A2) + t\text{-}DL\text{-}G_p.\text{advantage} \ (\text{reduction-}tDL$ 
A1)
 $\langle \text{proof} \rangle$ 

end

end
theory BatchKZG-def

imports KZG-correct

begin

```

17 Batch Opening Definition

```

locale BatchEvalKZG = KZG
begin

```

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

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

```

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

```

17.1 Polynomial operations and prerequisites

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

```

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

```

```

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

```

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

```

fun  $\psi_B$  :: 'e argument set  $\Rightarrow$  'e mod-ring poly  $\Rightarrow$  'e mod-ring poly where
   $\psi_B$  B  $\varphi$  = do {

```

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

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

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

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

lemma *deg-Prod*: $\text{degree } (\prod_{i \in B} [-i, 1:]) = \text{card } (B::'e \ \text{argument set})$
 $\langle \text{proof} \rangle$

lemma *deg-r-B-le*: $\text{degree } (r \ B \ \varphi) \leq \text{card } B$
 $\langle \text{proof} \rangle$

lemma *deg-r-B-less*: $B \neq \{\} \implies \text{degree } \varphi > \text{card } B \implies \text{degree } (r \ B \ \varphi) < \text{card } B$
 $\langle \text{proof} \rangle$

lemma *deg-div*: $\text{degree } ((x::'e \ \text{mod-ring poly}) \ \text{div} \ y) \leq \text{degree } x$
 $\langle \text{proof} \rangle$

lemma *deg-ψ_B*: $\text{degree } (\psi_B \ B \ \varphi) \leq \text{degree } \varphi$
 $\langle \text{proof} \rangle$

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

lemma *i-in-B-prod-B-zero[simp]*:
assumes $(i::'e \ \text{argument}) \in B$
shows $\text{poly } (\text{prod } (\lambda i. [-i, 1:]) \ B) \ i = 0$
 $\langle \text{proof} \rangle$

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

lemma *r-eq-φ-on-B*:
assumes $(i::'e \ \text{argument}) \in B$
shows $\text{poly } (r \ B \ \varphi) \ i = \text{poly } \varphi \ i$
 $\langle \text{proof} \rangle$

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

17.2 Function definitions

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

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

where

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

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* :: $'a \text{ vk} \Rightarrow 'a \text{ commit} \Rightarrow 'e \text{ argument set} \Rightarrow ('e \text{ batch-evaluation} \times 'a \text{ witness}) \Rightarrow \text{bool}$

where

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

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

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

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

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

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 com-mitter 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*: $(e(\mathbf{g} \hat{_{G_p}} \text{poly}(\prod_{i \in B} [:- i, 1:])) \alpha) (\mathbf{g} \hat{_{G_p}} \text{poly}(\psi_B B \varphi) \alpha)$
 $\otimes_{G_T} (e \mathbf{g} (\mathbf{g} \hat{_{G_p}} \text{poly}(r B \varphi) \alpha))$
 $= e(\mathbf{g} \hat{_{G_p}} \text{poly} \varphi \alpha) \mathbf{g}$
 $\langle \text{proof} \rangle$

theorem *KZG-correct*: *bKZG.correct-eval*
 $\langle \text{proof} \rangle$

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 stan-dard KZG.

theorem *bKZG.poly-bind-advantage* $\mathcal{A} \leq t\text{-DL-}G_p.\text{advantage}(\text{bind-reduction } \mathcal{A})$
 $\langle \text{proof} \rangle$

end

end

theory *BatchKZG-eval-bind*

imports *BatchKZG-correct tBS DH-assumption CryptHOL-ext*
begin

20 Evaluation Binding of the batched KZG

locale *BatchEvalKZG-PCS-eval-bind* = *BatchEvalKZG-PCS-correct*
begin

We prove two evaluation binding properties: 1. the classical evaluation binding game from PCS 2. the binding game from the original KZG paper [8] The difference is that 1. shows that the batched evaluation function is binding following the classical definition (i.e. two batched evaluations for the same argument must be equal) and 2. shows that the batched evaluation function is binding with respect to single evaluation function.

20.1 t-BSDH game

We instantiate the t-BSDH game for the pairing e , with the group G_p as preimage group and G_T as target group.

sublocale *t-BSDH*: *t-BSDH* G_p G_T *max-deg of-int-mod-ring* $\circ$ *int pow-mod-ring* G_p *pow-mod-ring* G_T e
 $\langle proof \rangle$

20.2 Binding for the batched evaluation according to KZG10

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

This is the formalized evaluation binding game

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

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

definition *bind-advantage* :: $(\text{'a'}, \text{'e'})$ *adversary* $\Rightarrow$ *real*
where *bind-advantage* $\mathcal{A} \equiv \text{spmf } (\text{bind-game } \mathcal{A}) \text{ True}$

20.2.1 Defining a reduction function to t-BSDH

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

```

fun reduction
  :: ('a, 'e) adversary  $\Rightarrow$  ('a, 'e, 'c) t-BSDH.adversary
where
  reduction  $\mathcal{A}$  PK = do {
    (C, i,  $\varphi$ -i, w-i, B, w-B, r-x)  $\leftarrow$   $\mathcal{A}$  PK;
    let p' = g-pow-PK-Prod PK (prod ( $\lambda i$ . [-i, 1:]) B div [-i, 1:]);
    let r' = g-pow-PK-Prod PK ((r-x - [:poly r-x i:]) div [-i, 1:]);
    return-spmf (-i::'e mod-ring, (e p' w-B  $\otimes_{G_T}$  e (r'  $\div_{G_p}$  w-i) g)  $\hat{\phantom{g}}_{G_T}$  (1/(( $\varphi$ -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  $\Rightarrow$  ('a, 'e, 'c) t-BSDH.adversary
where
  ext-reduction  $\mathcal{A}$  PK = do {
    (C, i,  $\varphi$ -i, w-i, B, w-B, r-x)  $\leftarrow$   $\mathcal{A}$  PK;
    - :: unit  $\leftarrow$  assert-spmf (i  $\in$  B  $\wedge$   $\varphi$ -i  $\neq$  poly r-x i
       $\wedge$  valid-argument-batch B
       $\wedge$  valid-eval ( $\varphi$ -i, w-i)  $\wedge$  valid-eval-batch (r-x, w-B)
       $\wedge$  verify-eval PK C i ( $\varphi$ -i, w-i)  $\wedge$  verify-eval-batch PK C B
(r-x, w-B));
    let p' = g-pow-PK-Prod PK (prod ( $\lambda i$ . [-i, 1:]) B div [-i, 1:]);
    let r' = g-pow-PK-Prod PK ((r-x - [:poly r-x i:]) div [-i, 1:]);
    return-spmf (-i::'e mod-ring, (e p' w-B  $\otimes_{G_T}$  e (r'  $\div_{G_p}$  w-i) g)  $\hat{\phantom{g}}_{G_T}$  (1/(( $\varphi$ -i -
poly r-x i)))) }

```

20.2.2 Helping lemmas

An alternative but equivalent game for the evaluation binding game. This alternative game encapsulates the event that the Adversary wins in the assert_spmf statement. It's a closely adapted proof from Sigma_Commit_Crypto.Commitment_Schemes bind_game_alt_def

```

lemma bind-game-alt-def:
  bind-game  $\mathcal{A}$  = TRY do {
    (ck, vk)  $\leftarrow$  key-gen;
    (C, i,  $\varphi$ -i, w-i, B, w-B, r-x)  $\leftarrow$   $\mathcal{A}$  ck;
    - :: unit  $\leftarrow$  assert-spmf (i  $\in$  B  $\wedge$   $\varphi$ -i  $\neq$  poly r-x i
       $\wedge$  valid-argument-batch B  $\wedge$  valid-eval ( $\varphi$ -i, w-i)  $\wedge$  valid-eval-batch (r-x, w-B));

    let b = verify-eval vk C i ( $\varphi$ -i, w-i);
    let b' = verify-eval-batch vk C B (r-x, w-B);
    -::unit  $\leftarrow$  assert-spmf (b  $\wedge$  b');
    return-spmf True} ELSE return-spmf False
    (is ?lhs = ?rhs)
  <proof>

```

show that `VerifyEvalBatch` of `B` and `r(x)` and `VerifEval` on $i \in B$ and $\varphi-i$ such that $\varphi-i \neq r(x)$ implies that the t-BSDH is broken. This lemma captures that the adversaries messages already break the t-BSDH assumption.

lemma *verifys-impl-t-BSDH-break*:

assumes $i \in B \wedge \varphi-i \neq \text{poly } r-x \ i$
 $\wedge \text{valid-argument-batch } B$
 $\wedge \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B)$
 $\wedge \text{verify-eval } (\text{map } (\lambda t. \mathbf{g} \wedge \alpha \wedge t) [0..<\text{max-deg} + 1]) \ C \ i \ (\varphi-i, w-i)$
 $\wedge \text{verify-eval-batch } (\text{map } (\lambda t. \mathbf{g} \wedge \alpha \wedge t) [0..<\text{max-deg} + 1]) \ C \ B \ (r-x, w-B)$
shows $e \ \mathbf{g} \ \mathbf{g} \wedge_{G_T} (1 / (\alpha + - \ i))$
 $=$
 $(e \ (g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g} \wedge \alpha \wedge t) [0..<\text{max-deg} + 1])$
 $\quad ((\prod_{i \in B.} [- \ i, 1:]) \text{ div } [- \ i, 1:])))$
 $w-B$
 $\otimes_{G_T}$
 $e \ (g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g} \wedge \alpha \wedge t) [0..<\text{max-deg} + 1])$
 $\quad ((r-x - [: \text{poly } r-x \ i:]) \text{ div } [- \ i, 1:]) \otimes$
 $\quad \text{inv } w-i)$
 $\mathbf{g}) \wedge_{G_T}$
 $(1 / (\varphi-i - \text{poly } r-x \ i))$
 $(\text{is } ?\text{goal} = ?\text{cmpt})$
 $\langle \text{proof} \rangle$

20.3 Literal helping lemma

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

lemma *literal-helping*:

$(i \in (B::'e \text{ argument set}) \wedge$
 $\quad (\varphi-i::'e \text{ evaluation}) \neq (\text{poly } r-x \ i::'e \text{ evaluation}) \wedge$
 $\quad \text{valid-argument-batch } B \wedge$
 $\quad \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B) \wedge$
 $\quad \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \wedge_{G_p} \alpha \wedge t) [0..<\text{max-deg} + 1]) \ C \ i$
 $(\varphi-i, w-i) \wedge$
 $\quad \text{verify-eval-batch } (\text{map } (\lambda t. \mathbf{g}_{G_p} \wedge_{G_p} (\alpha::'e \text{ mod-ring}) \wedge t)$
 $\quad [0..<\text{max-deg} + 1]) \ C \ B \ (r-x, w-B) \wedge$
 $\quad e \ \mathbf{g} \ \mathbf{g} \wedge_{G_T} (1 / (\alpha + - \ i)) =$
 $\quad (e \ (g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \wedge_{G_p} \alpha \wedge t) [0..<\text{max-deg} + 1])$
 $\quad ((\prod_{i \in B.} [- \ i, 1:]) \text{ div } [- \ i, 1:])))$
 $w-B \otimes_{G_T}$
 $e \ (g\text{-pow-PK-Prod } (\text{map } (\lambda t. \mathbf{g}_{G_p} \wedge_{G_p} \alpha \wedge t) [0..<\text{max-deg} + 1])$
 $\quad ((r-x - [: \text{poly } r-x \ i:]) \text{ div } [- \ i, 1:]) \otimes$
 $\quad \text{inv } w-i)$
 $\mathbf{g}) \wedge_{G_T}$
 $((1::'e \text{ mod-ring}) / (\varphi-i - \text{poly } r-x \ i)))$
 $\longleftrightarrow$
 $(i \in B \wedge$
 $\quad \varphi-i \neq \text{poly } r-x \ i \wedge$

$$\begin{aligned}
& \text{valid-argument-batch } B \wedge \\
& \text{valid-eval } (\varphi-i, w-i) \wedge \text{valid-eval-batch } (r-x, w-B) \wedge \\
& \quad \text{verify-eval } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} t) [0..<\text{max-deg} + 1]) \ C \ i \\
& (\varphi-i, w-i) \wedge \\
& \quad \text{verify-eval-batch } (\text{map } (\lambda t. \mathbf{g}_{G_p} \hat{\alpha}_{G_p} t) [0..<\text{max-deg} + 1]) \ C \\
& B \ (r-x, w-B)) \\
& \langle \text{proof} \rangle
\end{aligned}$$

20.4 KZG eval bind game to reduction game - equivalence theorem

We show that the binding game is equivalent to the t-BSDH game with the extended reduction adversary.

lemma *bind-game-eq-t-BSDH-red*: $\text{bind-game } \mathcal{A} = t\text{-BSDH.game } (\text{ext-reduction } \mathcal{A})$
 $\langle \text{proof} \rangle$

From the previous lemma we conclude that the adversary's advantage on winning the evaluation binding game is the same as winning the t-BSDH game with the extended reduction adversary.

lemma *evaluation-binding-ext-red*: $\text{bind-advantage } \mathcal{A} = t\text{-BSDH.advantage } (\text{ext-reduction } \mathcal{A})$
 $\langle \text{proof} \rangle$

Now we use overestimation to show that the advantage of winning the t-BSDH game with the extended reduction adversary is less than or equal to winning it with the normal reduction adversary.

lemma *overestimate-reductions*: $\text{spmf } (t\text{-BSDH.game } (\text{ext-reduction } \mathcal{A})) \ \text{True} \leq \text{spmf } (t\text{-BSDH.game } (\text{reduction } \mathcal{A})) \ \text{True}$
 $(\text{is_spmf } ?\text{lhgame } \text{True} \leq \text{spmf } ?\text{rhgame } \text{True})$
 $\langle \text{proof} \rangle$

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*: $\text{bind-advantage } \mathcal{A} \leq t\text{-BSDH.advantage } (\text{reduction } \mathcal{A})$
 $\langle \text{proof} \rangle$

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
 $\Rightarrow$ *AGM-knowledge-soundness-adversary1*
lift-to-algebraicT ('state, 'a ck, 'e mod-ring, 'e evaluation, 'a witness) *knowledge-soundness-adversary2*
 $G_p \Rightarrow$ *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*
 $= ('a' \text{ ck} \times 'state') \Rightarrow ('e' \text{ mod-ring} \times 'e' \text{ argument set} \times ('e' \text{ batch-evaluation} \times ('a' \text{ witness} \times \text{int list}))) \text{ spmf}$

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

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

restrict for AGM adversaries 1 and 2

interpretation *AGM1: Algebraic-Algorithm* $G_p \text{ listS } G_p.\text{groupS} \text{ prodC } G_p.\text{groupC}$
noConstrain
 $\langle \text{proof} \rangle$

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

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

$\langle proof \rangle$

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

$\Rightarrow ('e, 'state, 'a) \text{ knowledge-soundness-adversary2-AGM} \Rightarrow ('a, 'e) \text{ extractor} \Rightarrow$
bool spmf

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

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

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

$\Rightarrow ('a, 'e) \text{ adversary where}$

ks-to-eval-bind-reduction $\mathcal{A}1 \mathcal{A}2 \mathcal{E} ck = \text{do}$ {
 $((c, cvec), \sigma) \leftarrow \text{AGM1.restrict } \mathcal{A}1 \text{ } ck;$
 $(p, td) \leftarrow \mathcal{E} (c, cvec);$
 $(i, I, (r-x, (w, wvec))) \leftarrow \text{AGM2.restrict } \mathcal{A}2 (ck, \sigma);$
 $\text{let } (p-i, w') = \text{Eval } ck \text{ } td \text{ } p \text{ } i;$
 $\text{return-spmf } (c, i, p-i, w', I, w, r-x)$
 $\}$

Extractor definition

fun *E* :: ('a, 'e) extractor **where**

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

lemma *reduction-map-imp:*

$\text{let } ck = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1];$
 $vk = \text{map } (\lambda t. \mathbf{g}_{G_p} \hat{G_p} (\alpha \hat{t})) [0..<\text{max-deg}+1];$
 $(p-i, w-i) = \text{Eval } ck \text{ } td \text{ } (\text{Poly } (\text{map } (\text{of-int-mod-ring}::\text{int} \Rightarrow 'e \text{ mod-ring})$
 $cvec)) \text{ } i$
 in
 $\text{length } ck = \text{length } cvec$
 $\wedge c = \text{fold } (\lambda i \text{ acc. } acc \otimes ck!i [\hat{\cdot}] (cvec!i)) [0..<\text{length } ck] \mathbf{1}$
 $\wedge \text{verify-eval-batch } vk \text{ } c \text{ } I (r-x, w) \wedge \text{poly } r-x \text{ } i \neq p-i$
 $\wedge \text{valid-argument-batch } I \wedge \text{valid-eval-batch } (r-x, w) \wedge i \in I$
 $\longrightarrow$
 $\text{length } ck = \text{length } cvec$
 $\wedge c = \text{fold } (\lambda i \text{ acc. } acc \otimes ck!i [\hat{\cdot}] (cvec!i)) [0..<\text{length } ck] \mathbf{1}$
 $\wedge i \in I \wedge p-i \neq \text{poly } r-x \text{ } i$
 $\wedge \text{valid-argument-batch } I \wedge \text{valid-eval } (p-i, w-i) \wedge \text{valid-eval-batch } (r-x, w)$
 $\wedge \text{verify-eval } vk \text{ } c \text{ } i \text{ } (p-i, w-i)$
 $\wedge \text{verify-eval-batch } vk \text{ } c \text{ } I (r-x, w)$

$\langle proof \rangle$

theorem *ks-game-to-eval-bind-game*: $spmf\ (knowledge-soundness-game-AGM\ \mathcal{A}1\ \mathcal{A}2\ E)\ True$
 $\leq\ spmf\ (bind-game\ (ks-to-eval-bind-reduction\ \mathcal{A}1\ \mathcal{A}2\ E))\ True$
 $(is\ ?lhs\ \leq\ ?rhs)$
 $\langle proof \rangle$

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*:
 $spmf\ (knowledge-soundness-game-AGM\ \mathcal{A}1\ \mathcal{A}2\ E)\ True$
 $\leq\ t\text{-}BSDH.\text{advantage}\ (reduction\ (ks-to-eval-bind-reduction\ \mathcal{A}1\ \mathcal{A}2\ E))$
 $\langle proof \rangle$

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>.