

# $\Sigma$ -protocols and Commitment Schemes

David Butler, Andreas Lochbihler

May 26, 2024

## Abstract

We use CryptHOL [2] to formalise commitment schemes and  $\Sigma$ -protocols. Both are widely used fundamental two party cryptographic primitives. Security for commitment schemes is considered using game-based definitions whereas the security of  $\Sigma$ -protocols is considered using both the game-based and simulation-based security paradigms. In this work we first define security for both primitives and then prove secure multiple examples namely; the Schnorr, Chaum-Pedersen and Okamoto  $\Sigma$ -protocols as well as a construction that allows for compound (AND and OR)  $\Sigma$ -protocols and the Pedersen and Rivest commitment schemes. We also prove that commitment schemes can be constructed from  $\Sigma$ -protocols. We formalise this proof at an abstract level, only assuming the existence of a  $\Sigma$ -protocol, consequently the instantiations of this result for the concrete  $\Sigma$ -protocols we consider come for free.

## Contents

<b>1</b>	<b>Commitment Schemes</b>	<b>2</b>
1.1	Defining security . . . . .	2
1.2	Pedersen Commitment Scheme . . . . .	13
1.3	Rivest Commitment Scheme . . . . .	17
<b>2</b>	<b><math>\Sigma</math>-Protocols</b>	<b>19</b>
2.1	Defining $\Sigma$ -protocols . . . . .	20
2.2	Commitments from $\Sigma$ -protocols . . . . .	23
2.3	Schnorr $\Sigma$ -protocol . . . . .	25
2.4	Chaum-Pedersen $\Sigma$ -protocol . . . . .	29
2.5	Okamoto $\Sigma$ -protocol . . . . .	34
2.6	$\Sigma$ -AND statements . . . . .	44
2.7	$\Sigma$ -OR statements . . . . .	47

# 1 Commitment Schemes

A commitment scheme is a two party Cryptographic protocol run between a committer and a verifier. They allow the committer to commit to a chosen value while at a later time reveal the value. A commitment scheme is composed of three algorithms, the key generation, the commitment and the verification algorithms.

The two main properties of commitment schemes are hiding and binding.

Hiding is the property that the commitment leaks no information about the committed value, and binding is the property that the committer cannot reveal their a different message to the one they committed to; that is they are bound to their commitment. We follow the game based approach [12] to define security. A game is played between an adversary and a challenger.

```
theory Commitment-Schemes imports  
  CryptHOL.CryptHOL  
begin
```

## 1.1 Defining security

Here we define the hiding, binding and correctness properties of commitment schemes.

We provide the types of the adversaries that take part in the hiding and binding games. We consider two variants of the hiding property, one stronger than the other — thus we provide two hiding adversaries. The first hiding property we consider is analogous to the IND-CPA property for encryption schemes, the second, weaker notion, does not allow the adversary to choose the messages used in the game, instead they are sampled from a set distribution.

```
type-synonym ('vk', 'plain', 'commit', 'state) hid-adv =  
  ('vk' ⇒ (('plain' × 'plain') × 'state) spmf)  
  × ('commit' ⇒ 'state ⇒ bool spmf)
```

```
type-synonym 'commit' hid = 'commit' ⇒ bool spmf
```

```
type-synonym ('ck', 'plain', 'commit', 'opening') bind-adversary =  
  'ck' ⇒ ('commit' × 'plain' × 'opening' × 'plain' × 'opening') spmf
```

We fix the algorithms that make up a commitment scheme in the locale.

```
locale abstract-commitment =  
  fixes key-gen :: ('ck × 'vk) spmf — outputs the keys received by the two parties  
  and commit :: 'ck ⇒ 'plain ⇒ ('commit × 'opening) spmf — outputs the  
  commitment as well as the opening values sent by the committer in the reveal  
  phase  
  and verify :: 'vk ⇒ 'plain ⇒ 'commit ⇒ 'opening ⇒ bool
```

**and** *valid-msg* :: 'plain  $\Rightarrow$  bool — checks whether a message is valid, used in the hiding game

**begin**

**definition** *valid-msg-set* = {*m*. *valid-msg* *m*}

**definition** *lossless* :: ('pub-key, 'plain, 'commit, 'state) hid-adv  $\Rightarrow$  bool

**where** *lossless*  $\mathcal{A} \longleftrightarrow$

(( $\forall$  *pk*. *lossless-spmf* (*fst*  $\mathcal{A}$  *pk*))  $\wedge$   
( $\forall$  *commit*  $\sigma$ . *lossless-spmf* (*snd*  $\mathcal{A}$  *commit*  $\sigma$ )))

The correct game runs the three algorithms that make up commitment schemes and outputs the output of the verification algorithm.

**definition** *correct-game* :: 'plain  $\Rightarrow$  bool *spmf*

**where** *correct-game* *m* = do {

(*ck*, *vk*)  $\leftarrow$  *key-gen*;

(*c*, *d*)  $\leftarrow$  *commit* *ck* *m*;

*return-spmf* (*verify* *vk* *m* *c* *d*)}

**lemma**  $\llbracket$  *lossless-spmf* *key-gen*; *lossless-spmf* *TI*;

$\bigwedge$  *pk* *m*. *valid-msg* *m*  $\Longrightarrow$  *lossless-spmf* (*commit* *pk* *m*)  $\llbracket$   
 $\Longrightarrow$  *valid-msg* *m*  $\Longrightarrow$  *lossless-spmf* (*correct-game* *m*)

*<proof>*

**definition** *correct* **where** *correct*  $\equiv$  ( $\forall$  *m*. *valid-msg* *m*  $\longrightarrow$  *spmf* (*correct-game* *m*)  
*True* = 1)

The hiding property is defined using the hiding game. Here the adversary is asked to output two messages, the challenger flips a coin to decide which message to commit and hand to the adversary. The adversary's challenge is to guess which commitment it was handed. Note we must check the two messages outputted by the adversary are valid.

**primrec** *hiding-game-ind-cpa* :: ('vk, 'plain, 'commit, 'state) hid-adv  $\Rightarrow$  bool *spmf*

**where** *hiding-game-ind-cpa* ( $\mathcal{A}1$ ,  $\mathcal{A}2$ ) = *TRY* do {

(*ck*, *vk*)  $\leftarrow$  *key-gen*;

((*m0*, *m1*),  $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  *vk*;

- :: *unit*  $\leftarrow$  *assert-spmf* (*valid-msg* *m0*  $\wedge$  *valid-msg* *m1*);

*b*  $\leftarrow$  *coin-spmf*;

(*c*, *d*)  $\leftarrow$  *commit* *ck* (if *b* then *m0* else *m1*);

*b'* :: bool  $\leftarrow$   $\mathcal{A}2$  *c*  $\sigma$ ;

*return-spmf* (*b'* = *b*)} *ELSE* *coin-spmf*

The adversary wins the game if *b* = *b'*.

**lemma** *lossless-hiding-game*:

$\llbracket$  *lossless*  $\mathcal{A}$ ; *lossless-spmf* *key-gen*;

$\bigwedge$  *pk* *plain*. *valid-msg* *plain*  $\Longrightarrow$  *lossless-spmf* (*commit* *pk* *plain*)  $\llbracket$   
 $\Longrightarrow$  *lossless-spmf* (*hiding-game-ind-cpa*  $\mathcal{A}$ )

*<proof>*

To define security we consider the advantage an adversary has of winning the game over a tossing a coin to determine their output.

**definition** *hiding-advantage-ind-cpa* :: ('vk, 'plain, 'commit, 'state) hid-adv  $\Rightarrow$  real  
**where** *hiding-advantage-ind-cpa*  $\mathcal{A} \equiv | \text{spmf} (\text{hiding-game-ind-cpa } \mathcal{A}) \text{ True} - 1/2 |$

**definition** *perfect-hiding-ind-cpa* :: ('vk, 'plain, 'commit, 'state) hid-adv  $\Rightarrow$  bool  
**where** *perfect-hiding-ind-cpa*  $\mathcal{A} \equiv (\text{hiding-advantage-ind-cpa } \mathcal{A} = 0)$

The binding game challenges an adversary to bind two messages to the same committed value. Both opening values and messages are verified with respect to the same committed value, the adversary wins if the game outputs true. We must check some conditions of the adversaries output are met; we will always require that  $m \neq m'$ , other conditions will be dependent on the protocol for example we may require group or field membership.

**definition** *bind-game* :: ('ck, 'plain, 'commit, 'opening) bind-adversary  $\Rightarrow$  bool  
*spmf*  
**where** *bind-game*  $\mathcal{A} = \text{TRY do } \{$   
 $(ck, vk) \leftarrow \text{key-gen};$   
 $(c, m, d, m', d') \leftarrow \mathcal{A} ck;$   
 $- :: \text{unit} \leftarrow \text{assert-spmf} (m \neq m' \wedge \text{valid-msg } m \wedge \text{valid-msg } m');$   
 $\text{let } b = \text{verify } vk \ m \ c \ d;$   
 $\text{let } b' = \text{verify } vk \ m' \ c \ d';$   
 $\text{return-spmf } (b \wedge b') \}$  *ELSE return-spmf False*

We proof the binding game is equivalent to the following game which is easier to work with. In particular we assert b and b' in the game and return True.

**lemma** *bind-game-alt-def*:  
*bind-game*  $\mathcal{A} = \text{TRY do } \{$   
 $(ck, vk) \leftarrow \text{key-gen};$   
 $(c, m, d, m', d') \leftarrow \mathcal{A} ck;$   
 $- :: \text{unit} \leftarrow \text{assert-spmf} (m \neq m' \wedge \text{valid-msg } m \wedge \text{valid-msg } m');$   
 $\text{let } b = \text{verify } vk \ m \ c \ d;$   
 $\text{let } b' = \text{verify } vk \ m' \ c \ d';$   
 $- :: \text{unit} \leftarrow \text{assert-spmf} (b \wedge b');$   
 $\text{return-spmf True} \}$  *ELSE return-spmf False*  
*(is ?lhs = ?rhs)*  
*<proof>*

**lemma** *lossless-binding-game*: *lossless-spmf* (*bind-game*  $\mathcal{A}$ )  
*<proof>*

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

**end**

```

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

context cyclic-group begin

lemma generator-pow-order:  $\mathbf{g} [ \wedge ] \text{order } G = \mathbf{1}$ 
  <proof>

lemma generator-pow-mult-order:  $\mathbf{g} [ \wedge ] (\text{order } G * \text{order } G) = \mathbf{1}$ 
  <proof>

lemma pow-generator-mod:  $\mathbf{g} [ \wedge ] (k \text{ mod } \text{order } G) = \mathbf{g} [ \wedge ] k$ 
  <proof>

lemma pow-carrier-mod:
  assumes  $g \in \text{carrier } G$ 
  shows  $g [ \wedge ] (k \text{ mod } \text{order } G) = g [ \wedge ] k$ 
  <proof>

lemma pow-generator-mod-int:  $\mathbf{g} [ \wedge ] ((k::\text{int}) \text{ mod } \text{order } G) = \mathbf{g} [ \wedge ] k$ 
  <proof>

lemma pow-generator-eq-iff-cong:
  finite (carrier G)  $\implies \mathbf{g} [ \wedge ] x = \mathbf{g} [ \wedge ] y \iff [x = y] (\text{mod } \text{order } G)$ 
  <proof>

lemma power-distrib:
  assumes  $h \in \text{carrier } G$ 
  shows  $\mathbf{g} [ \wedge ] (e :: \text{nat}) \otimes h [ \wedge ] e = (\mathbf{g} \otimes h) [ \wedge ] e$ 
  (is ?lhs = ?rhs)
  <proof>

lemma neg-power-inverse:
  assumes  $g \in \text{carrier } G$ 
  and  $x < \text{order } G$ 
  shows  $g [ \wedge ] (\text{order } G - (x :: \text{nat})) = \text{inv } (g [ \wedge ] x)$ 
  <proof>

lemma int-nat-pow: assumes  $a \geq 0$  shows  $(\mathbf{g} [ \wedge ] (\text{int } (a :: \text{nat}))) [ \wedge ] (b::\text{int}) =$ 
 $\mathbf{g} [ \wedge ] (a*b)$ 
  <proof>

lemma pow-gen-mod-mult:
  shows  $(\mathbf{g} [ \wedge ] (a::\text{nat}) \otimes \mathbf{g} [ \wedge ] (b::\text{nat})) [ \wedge ] ((c::\text{int})*\text{int } (d::\text{nat})) = (\mathbf{g} [ \wedge ] a \otimes \mathbf{g} [ \wedge ] b) [ \wedge ] ((c*\text{int } d) \text{ mod } (\text{order } G))$ 
  <proof>

```

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

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

**lemma** *l-cancel-inv*:  
**assumes**  $h \in \text{carrier } G$   
**shows**  $(\mathbf{g} [\wedge] (a :: \text{nat}) \otimes \text{inv } (\mathbf{g} [\wedge] a)) \otimes h = h$   
**(is ?lhs = ?rhs)**  
 $\langle \text{proof} \rangle$

**lemma** *inverse-split*:  
**assumes**  $a \in \text{carrier } G$  **and**  $b \in \text{carrier } G$   
**shows**  $\text{inv } (a \otimes b) = \text{inv } a \otimes \text{inv } b$   
 $\langle \text{proof} \rangle$

**lemma** *inverse-pow-pow*:  
**assumes**  $a \in \text{carrier } G$   
**shows**  $\text{inv } (a [\wedge] (r :: \text{nat})) = (\text{inv } a) [\wedge] r$   
 $\langle \text{proof} \rangle$

**lemma** *l-neq-1-exp-neq-0*:  
**assumes**  $l \in \text{carrier } G$   
**and**  $l \neq 1$   
**and**  $l = \mathbf{g} [\wedge] (t :: \text{nat})$   
**shows**  $t \neq 0$   
 $\langle \text{proof} \rangle$

**lemma** *order-gt-1-gen-not-1*:  
**assumes**  $\text{order } G > 1$   
**shows**  $\mathbf{g} \neq 1$   
 $\langle \text{proof} \rangle$

**lemma** *power-swap*:  $((\mathbf{g} [\wedge] (\alpha 0 :: \text{nat})) [\wedge] (r :: \text{nat})) = ((\mathbf{g} [\wedge] r) [\wedge] \alpha 0)$   
**(is ?lhs = ?rhs)**  
 $\langle \text{proof} \rangle$

**lemma** *gen-power-0*:  
**fixes**  $r :: \text{nat}$   
**assumes**  $\mathbf{g} [\wedge] r = 1$   
**and**  $r < \text{order } G$   
**shows**  $r = 0$

```

    <proof>

lemma group-eq-pow-eq-mod:
  fixes  $a\ b :: \text{nat}$ 
  assumes  $\mathbf{g} [\wedge] a = \mathbf{g} [\wedge] b$ 
    and  $\text{order } G > 0$ 
  shows  $[a = b] \text{ (mod order } G)$ 
  <proof>

end

end
theory Discrete-Log imports
  CryptHOL.CryptHOL
  Cyclic-Group-Ext
begin

locale dis-log =
  fixes  $\mathcal{G} :: \text{'grp cyclic-group (structure)}$ 
  assumes order-gt-0 [simp]:  $\text{order } \mathcal{G} > 0$ 
begin

type-synonym 'grp' dislog-adv = 'grp'  $\Rightarrow$  nat spmf

type-synonym 'grp' dislog-adv' = 'grp'  $\Rightarrow$  (nat  $\times$  nat) spmf

type-synonym 'grp' dislog-adv2 = 'grp'  $\times$  'grp'  $\Rightarrow$  nat spmf

definition dis-log :: 'grp dislog-adv  $\Rightarrow$  bool spmf
where dis-log  $\mathcal{A} = \text{TRY do \{$ 
   $x \leftarrow \text{sample-uniform (order } \mathcal{G})$ ;
   $\text{let } h = \mathbf{g} [\wedge] x$ ;
   $x' \leftarrow \mathcal{A} h$ ;
   $\text{return-spmf } ([x = x'] \text{ (mod order } \mathcal{G})) \}$  ELSE return-spmf False

definition advantage :: 'grp dislog-adv  $\Rightarrow$  real
where advantage  $\mathcal{A} \equiv \text{spmf (dis-log } \mathcal{A}) \text{ True}$ 

lemma lossless-dis-log:  $\llbracket 0 < \text{order } \mathcal{G}; \forall h. \text{lossless-spmf } (\mathcal{A} h) \rrbracket \implies \text{lossless-spmf}$ 
  (dis-log  $\mathcal{A}$ )
  <proof>

end

locale dis-log-alt =
  fixes  $\mathcal{G} :: \text{'grp cyclic-group (structure)}$ 
  and  $x :: \text{nat}$ 
  assumes order-gt-0 [simp]:  $\text{order } \mathcal{G} > 0$ 

```

**begin**

**sublocale** *dis-log*: *dis-log*  $\mathcal{G}$   
*<proof>*

**definition**  $g' = \mathbf{g} [\wedge] x$

**definition** *dis-log2* :: '*grp dis-log.dislog-adv*'  $\Rightarrow$  *bool* *spmf*  
**where** *dis-log2*  $\mathcal{A} = \text{TRY do}$  {  
   $w \leftarrow \text{sample-uniform (order } \mathcal{G}\text{)}$ ;  
   $\text{let } h = \mathbf{g} [\wedge] w$ ;  
   $(w1', w2') \leftarrow \mathcal{A} h$ ;  
   $\text{return-spmf } ([w = (w1' + x * w2')] \text{ (mod (order } \mathcal{G}\text{))})$  *ELSE*  $\text{return-spmf False}$

**definition** *advantage2* :: '*grp dis-log.dislog-adv*'  $\Rightarrow$  *real*  
**where** *advantage2*  $\mathcal{A} \equiv \text{spmf (dis-log2 } \mathcal{A}\text{) True}$

**definition** *adversary2* :: ('*grp*  $\Rightarrow$  (*nat*  $\times$  *nat*) *spmf*)  $\Rightarrow$  '*grp*  $\Rightarrow$  *nat* *spmf*  
**where** *adversary2*  $\mathcal{A} h = \text{do}$  {  
   $(w1, w2) \leftarrow \mathcal{A} h$ ;  
   $\text{return-spmf } (w1 + x * w2)$ }

**definition** *dis-log3* :: '*grp dis-log.dislog-adv2*'  $\Rightarrow$  *bool* *spmf*  
**where** *dis-log3*  $\mathcal{A} = \text{TRY do}$  {  
   $w \leftarrow \text{sample-uniform (order } \mathcal{G}\text{)}$ ;  
   $\text{let } (h, w) = ((\mathbf{g} [\wedge] w, g' [\wedge] w), w)$ ;  
   $w' \leftarrow \mathcal{A} h$ ;  
   $\text{return-spmf } ([w = w'] \text{ (mod (order } \mathcal{G}\text{))})$  *ELSE*  $\text{return-spmf False}$

**definition** *advantage3* :: '*grp dis-log.dislog-adv2*'  $\Rightarrow$  *real*  
**where** *advantage3*  $\mathcal{A} \equiv \text{spmf (dis-log3 } \mathcal{A}\text{) True}$

**definition** *adversary3*:: '*grp dis-log.dislog-adv2*'  $\Rightarrow$  '*grp*  $\Rightarrow$  *nat* *spmf*  
**where** *adversary3*  $\mathcal{A} g = \text{do}$  {  
   $\mathcal{A} (g, g [\wedge] x)$ }

**end**

**locale** *dis-log-alt-reductions* = *dis-log-alt* + *cyclic-group*  $\mathcal{G}$   
**begin**

**lemma** *dis-log-adv3*:  
**shows** *advantage3*  $\mathcal{A} = \text{dis-log.} \text{advantage (adversary3 } \mathcal{A}\text{)}$   
*<proof>*

**lemma** *dis-log-adv2*:  
**shows** *advantage2*  $\mathcal{A} = \text{dis-log.} \text{advantage (adversary2 } \mathcal{A}\text{)}$   
*<proof>*



**end**

**end**

**theory** *Number-Theory-Aux* **imports**

*HOL-Number-Theory.Cong*

*HOL-Number-Theory.Residues*

**begin**

**abbreviation** *inverse* **where** *inverse*  $x\ q \equiv (fst\ (bezw\ x\ q))$

**lemma** *inverse*: **assumes**  $gcd\ x\ q = 1$

**shows**  $[x * inverse\ x\ q = 1] \ (mod\ q)$

*<proof>*

**lemma** *prod-not-prime*:

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

**and** *prime*  $y$

**and**  $x > 2$

**and**  $y > 2$

**shows**  $\neg\ prime\ ((x-1)*(y-1))$

*<proof>*

**lemma** *ex-inverse*:

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

**and** *prime*  $P$

**and** *prime*  $Q$

**and**  $P \neq Q$

**shows**  $\exists\ d.\ [e*d = 1] \ (mod\ (P-1)) \wedge\ d \neq 0$

*<proof>*

**lemma** *ex-k1-k2*:

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

**and**  $[e*d = 1] \ (mod\ (P-1))$

**shows**  $\exists\ k1\ k2.\ e*d + k1*(P-1) = 1 + k2*(P-1)$

*<proof>*

**lemma**  $a > b \implies int\ a - int\ b = int\ (a - b)$

*<proof>*

**lemma** *ex-k-mod*:

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

**and**  $P \neq Q$

**and** *prime*  $P$

**and** *prime*  $Q$

**and**  $d \neq 0$

**and**  $[e*d = 1] \ (mod\ (P-1))$

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

*<proof>*

**lemma** *fermat-little-theorem*:

**assumes** *prime* ( $P :: \text{nat}$ )

**shows**  $[x^P = x] \pmod P$

$\langle \text{proof} \rangle$

**lemma** *prime-field*:

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

**and**  $a < q$

**and**  $a \neq 0$

**shows** *coprime*  $a\ q$

$\langle \text{proof} \rangle$

**end**

**theory** *Uniform-Sampling imports*

*CryptHOL.CryptHOL*

*HOL-Number-Theory.Cong*

**begin**

**definition** *sample-uniform-units*  $:: \text{nat} \Rightarrow \text{nat\ spmf}$

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

**lemma** *set-spmf-sample-uniform-units [simp]*:

*set-spmf* (*sample-uniform-units*  $q$ ) =  $\{..< q\} - \{0\}$

$\langle \text{proof} \rangle$

**lemma** *lossless-sample-uniform-units*:

**assumes** ( $p :: \text{nat}$ )  $> 1$

**shows** *lossless-spmf* (*sample-uniform-units*  $p$ )

$\langle \text{proof} \rangle$

**lemma** *weight-sample-uniform-units*:

**assumes** ( $p :: \text{nat}$ )  $> 1$

**shows** *weight-spmf* (*sample-uniform-units*  $p$ ) =  $1$

$\langle \text{proof} \rangle$

**lemma** *one-time-pad'*:

**assumes** *inj-on*: *inj-on*  $f$  ( $\{..< q\} - \{0\}$ )

**and** *sur*:  $f ' (\{..< q\} - \{0\}) = (\{..< q\} - \{0\})$

**shows** *map-spmf*  $f$  (*sample-uniform-units*  $q$ ) = (*sample-uniform-units*  $q$ )

(**is** *?lhs* = *?rhs*)

$\langle \text{proof} \rangle$

**lemma** *one-time-pad*:

**assumes** *inj-on*: *inj-on*  $f$   $\{..< q\}$

**and** *sur*:  $f ' \{..< q\} = \{..< q\}$

**shows** *map-spmf*  $f$  (*sample-uniform*  $q$ ) = (*sample-uniform*  $q$ )

(**is** *?lhs* = *?rhs*)

*<proof>*

**lemma** *plus-inj-eq*:

**assumes**  $x: x < q$

**and**  $x': x' < q$

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

**shows**  $x = x'$

*<proof>*

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

*<proof>*

**lemma** *surj-uni-samp-plus*:

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

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

*<proof>*

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

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

*<proof>*

**lemma** *mult-inj-eq*:

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

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

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

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

**shows**  $y = y'$

*<proof>*

**lemma** *inj-on-mult*:

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

**shows**  $inj-on (\lambda b. x*b \bmod q) \{..<q\}$

*<proof>*

**lemma** *surj-on-mult*:

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

**and**  $inj: inj-on (\lambda b. x*b \bmod q) \{..<q\}$

**shows**  $(\lambda b. x*b \bmod q) ' \{..<q\} = \{..<q\}$

*<proof>*

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

**assumes**  $coprime: coprime x q$

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

*<proof>*

**lemma** *inj-on-mult'*:

**assumes** *coprime*:  $\text{coprime } x \ (q::\text{nat})$   
**shows** *inj-on*  $(\lambda b. x*b \ \text{mod } q) \ (\{..<q\} - \{0\})$   
*<proof>*

**lemma** *surj-on-mult'*:

**assumes** *coprime*:  $\text{coprime } x \ (q::\text{nat})$   
**and** *inj*: *inj-on*  $(\lambda b. x*b \ \text{mod } q) \ (\{..<q\} - \{0\})$   
**shows**  $(\lambda b. x*b \ \text{mod } q) \ ' \ (\{..<q\} - \{0\}) = (\{..<q\} - \{0\})$   
*<proof>*

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

**assumes** *coprime*:  $\text{coprime } x \ q$   
**shows** *map-spmf*  $(\lambda b. x*b \ \text{mod } q) \ (\text{sample-uniform-units } q) = \text{sample-uniform-units } q$   
*<proof>*

**lemma** *samp-uni-add-mult*:

**assumes** *coprime*:  $\text{coprime } x \ (q::\text{nat})$   
**and** *x'*:  $x' < q$   
**and** *y'*:  $y' < q$   
**and** *map*:  $(y + x * x') \ \text{mod } q = (y + x * y') \ \text{mod } q$   
**shows**  $x' = y'$   
*<proof>*

**lemma** *inj-on-add-mult*:

**assumes** *coprime*:  $\text{coprime } x \ (q::\text{nat})$   
**shows** *inj-on*  $(\lambda b. (y + x*b) \ \text{mod } q) \ \{..<q\}$   
*<proof>*

**lemma** *surj-on-add-mult*:

**assumes** *coprime*:  $\text{coprime } x \ (q::\text{nat})$   
**and** *inj*: *inj-on*  $(\lambda b. (y + x*b) \ \text{mod } q) \ \{..<q\}$   
**shows**  $(\lambda b. (y + x*b) \ \text{mod } q) \ ' \ \{..<q\} = \{..<q\}$   
*<proof>*

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

**assumes** *coprime*:  $\text{coprime } x \ q$   
**shows** *map-spmf*  $(\lambda b. (y + x*b) \ \text{mod } q) \ (\text{sample-uniform } q) = (\text{sample-uniform } q)$   
*<proof>*

**lemma** *inj-on-minus*: *inj-on*  $(\lambda(b :: \text{nat}). (y + (q - b)) \ \text{mod } q) \ \{..<q\}$   
*<proof>*

**lemma** *surj-on-minus*:

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

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

**shows**  $\text{map-spmf}(\lambda b. (y + (q - b)) \bmod q) (\text{sample-uniform } q) = \text{sample-uniform } q$   
*<proof>*

**lemma** *not-coin-spmf*:  $\text{map-spmf} (\lambda a. \neg a) \text{coin-spmf} = \text{coin-spmf}$

*<proof>*

**lemma** *xor-uni-samp*:  $\text{map-spmf}(\lambda b. y \oplus b) (\text{coin-spmf}) = \text{map-spmf}(\lambda b. b)$   
*(coin-spmf)*

**(is ?lhs = ?rhs)**  
*<proof>*

**lemma** *ped-inv-mapping*:

**assumes**  $(a::nat) < q$   
**and**  $[m \neq 0] \bmod q$   
**shows**  $\text{map-spmf} (\lambda d. (d + a * (m::nat)) \bmod q) (\text{sample-uniform } q) = \text{map-spmf}$   
 $(\lambda d. (d + q * m - a * m) \bmod q) (\text{sample-uniform } q)$   
**(is ?lhs = ?rhs)**  
*<proof>*

**end**

## 1.2 Pedersen Commitment Scheme

The Pedersen commitment scheme [8] is a commitment scheme based on a cyclic group. We use the construction of cyclic groups from CryptHOL to formalise the commitment scheme. We prove perfect hiding and computational binding, with a reduction to the discrete log problem. We a proof of the Pedersen commitment scheme is realised in the instantiation of the Schnorr  $\Sigma$ -protocol with the general construction of commitment schemes from  $\Sigma$ -protocols. The commitment scheme that is realised there however take the inverse of the message in the commitment phase due to the construction of the simulator in the  $\Sigma$ -protocol proof. The two schemes are in some way equal however as we do not have a well defined notion of equality for commitment schemes we keep this section of the formalisation. This also serves as reference to the formal proof of the Pedersen commitment scheme we provide in [5].

**theory** *Pedersen imports*

*Commitment-Schemes*

*HOL-Number-Theory.Cong*

```

    Cyclic-Group-Ext
    Discrete-Log
    Number-Theory-Aux
    Uniform-Sampling
begin

locale pedersen-base =
  fixes  $\mathcal{G} :: 'grp$  cyclic-group (structure)
  assumes prime-order: prime (order  $\mathcal{G}$ )
begin

lemma order-gt-0 [simp]: order  $\mathcal{G} > 0$ 
  ⟨proof⟩

type-synonym 'grp' ck = 'grp'
type-synonym 'grp' vk = 'grp'
type-synonym plain = nat
type-synonym 'grp' commit = 'grp'
type-synonym opening = nat

definition key-gen :: ('grp ck × 'grp vk) spmf
where
  key-gen = do {
    x :: nat ← sample-uniform (order  $\mathcal{G}$ );
    let h = g [∧] x;
    return-spmf (h, h)
  }

definition commit :: 'grp ck ⇒ plain ⇒ ('grp commit × opening) spmf
where
  commit ck m = do {
    d :: nat ← sample-uniform (order  $\mathcal{G}$ );
    let c = (g [∧] d) ⊗ (ck [∧] m);
    return-spmf (c,d)
  }

definition commit-inv :: 'grp ck ⇒ plain ⇒ ('grp commit × opening) spmf
where
  commit-inv ck m = do {
    d :: nat ← sample-uniform (order  $\mathcal{G}$ );
    let c = (g [∧] d) ⊗ (inv ck [∧] m);
    return-spmf (c,d)
  }

definition verify :: 'grp vk ⇒ plain ⇒ 'grp commit ⇒ opening ⇒ bool
where
  verify v-key m c d = (c = (g [∧] d ⊗ v-key [∧] m))

definition valid-msg :: plain ⇒ bool

```

**where** *valid-msg*  $msg \equiv (msg < order \mathcal{G})$

**definition** *dis-log-A* :: ('grp ck, plain, 'grp commit, opening) bind-adversary  $\Rightarrow$   
'grp ck  $\Rightarrow$  nat spmf

**where** *dis-log-A*  $\mathcal{A} h = do$  {  
 (c, m, d, m', d')  $\leftarrow \mathcal{A} h$ ;  
 - :: unit  $\leftarrow assert-spmf (m \neq m' \wedge valid\text{-}msg\ m \wedge valid\text{-}msg\ m')$ ;  
 - :: unit  $\leftarrow assert-spmf (c = \mathbf{g} [\uparrow] d \otimes h [\uparrow] m \wedge c = \mathbf{g} [\uparrow] d' \otimes h [\uparrow] m')$ ;  
 return-spmf (if (m > m') then (nat ((int d' - int d) \* inverse (m - m') (order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ )) else  
 (nat ((int d - int d') \* inverse (m' - m) (order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ )))}

**sublocale** *ped-commit*: abstract-commitment key-gen commit verify valid-msg <proof>

**sublocale** *discrete-log*: *dis-log* -  
<proof>

**end**

**locale** *pedersen* = *pedersen-base* + *cyclic-group*  $\mathcal{G}$   
**begin**

**lemma** *mod-one-cancel*: **assumes** [int y \* z \* x = y' \* x] (mod order  $\mathcal{G}$ ) **and** [z \* x = 1] (mod order  $\mathcal{G}$ )  
**shows** [int y = y' \* x] (mod order  $\mathcal{G}$ )  
<proof>

**lemma** *dis-log-break*:  
**fixes** d d' m m' :: nat  
**assumes** c:  $\mathbf{g} [\uparrow] d' \otimes (\mathbf{g} [\uparrow] y) [\uparrow] m' = \mathbf{g} [\uparrow] d \otimes (\mathbf{g} [\uparrow] y) [\uparrow] m$   
**and** *y-less-order*: y < order  $\mathcal{G}$   
**and** *m-ge-m'*: m > m'  
**and** m: m < order  $\mathcal{G}$   
**shows** y = nat ((int d' - int d) \* (fst (bezw ((m - m') (order  $\mathcal{G}$ )))) mod order  $\mathcal{G}$ )  
<proof>

**lemma** *dis-log-break'*:  
**assumes** y < order  $\mathcal{G}$   
**and**  $\neg m' < m$   
**and** m  $\neq$  m'  
**and** m: m' < order  $\mathcal{G}$   
**and**  $\mathbf{g} [\uparrow] d \otimes (\mathbf{g} [\uparrow] y) [\uparrow] m = \mathbf{g} [\uparrow] d' \otimes (\mathbf{g} [\uparrow] y) [\uparrow] m'$   
**shows** y = nat ((int d - int d') \* fst (bezw ((m' - m)) (order  $\mathcal{G}$ ))) mod int (order  $\mathcal{G}$ )  
<proof>

**lemma** *set-spmf-samp-uni* [*simp*]: *set-spmf* (sample-uniform (order  $\mathcal{G}$ )) = {x. x < order  $\mathcal{G}$ }

$\langle \text{proof} \rangle$

**lemma** *correct*:  
**shows**  $\text{spm}f (\text{ped-commit.correct-game } m) \text{ True} = 1$   
 $\langle \text{proof} \rangle$

**theorem** *abstract-correct*:  
**shows**  $\text{ped-commit.correct}$   
 $\langle \text{proof} \rangle$

**lemma** *perfect-hiding*:  
**shows**  $\text{spm}f (\text{ped-commit.hiding-game-ind-cpa } \mathcal{A}) \text{ True} - 1/2 = 0$   
**including** *monad-normalisation*  
 $\langle \text{proof} \rangle$

**theorem** *abstract-perfect-hiding*:  
**shows**  $\text{ped-commit.perfect-hiding-ind-cpa } \mathcal{A}$   
 $\langle \text{proof} \rangle$

**lemma** *bind-output-cong*:  
**assumes**  $x < \text{order } \mathcal{G}$   
**shows**  $(x = \text{nat } ((\text{int } b - \text{int } ab) * \text{fst } (\text{bezw } (aa - ac) (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})))$   
 $\longleftrightarrow [x = \text{nat } ((\text{int } b - \text{int } ab) * \text{fst } (\text{bezw } (aa - ac) (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G}))] (\text{mod order } \mathcal{G})$   
 $\langle \text{proof} \rangle$

**lemma** *bind-game-eq-dis-log*:  
**shows**  $\text{ped-commit.bind-game } \mathcal{A} = \text{discrete-log.dis-log } (\text{dis-log-}\mathcal{A} \ \mathcal{A})$   
 $\langle \text{proof} \rangle$

**theorem** *pedersen-bind*:  $\text{ped-commit.bind-advantage } \mathcal{A} = \text{discrete-log.} \text{advantage } (\text{dis-log-}\mathcal{A} \ \mathcal{A})$   
 $\langle \text{proof} \rangle$

**end**

**locale** *pedersen-asymp* =  
**fixes**  $\mathcal{G} :: \text{nat} \Rightarrow \text{'grp cyclic-group}$   
**assumes**  $\text{pedersen: } \bigwedge \eta. \text{pedersen } (\mathcal{G} \ \eta)$   
**begin**

**sublocale** *pedersen*  $\mathcal{G} \ \eta$  **for**  $\eta$   $\langle \text{proof} \rangle$

**theorem** *pedersen-correct-asymp*:  
**shows**  $\text{ped-commit.correct } n$   
 $\langle \text{proof} \rangle$

**theorem** *pedersen-perfect-hiding-asymp*:



**shows** *ped-commit.perfect-hiding-ind-cpa*  $n$  ( $\mathcal{A}$   $n$ )  
     $\langle$ *proof* $\rangle$

**theorem** *pedersen-bind-asym*:

**shows** *negligible* ( $\lambda$   $n$ . *ped-commit.bind-advantage*  $n$  ( $\mathcal{A}$   $n$ ))  
     $\longleftrightarrow$  *negligible* ( $\lambda$   $n$ . *discrete-log.advantage*  $n$  (*dis-log- $\mathcal{A}$*   $n$  ( $\mathcal{A}$   $n$ )))  
     $\langle$ *proof* $\rangle$

**end**

**end**

### 1.3 Rivest Commitment Scheme

The Rivest commitment scheme was first introduced in [10]. We note however the original scheme did not allow for perfect hiding. This was pointed out by Blundo and Masucci in [3] who alightly ammended the commitment scheme so that is provided perfect hiding.

The Rivest commitment scheme uses a trusted initialiser to provide correlated randomness to the two parties before an execution of the protocol. In our framework we set these as keys that held by the respective parties.

**theory** *Rivest imports*

*Commitment-Schemes*  
    *HOL-Number-Theory.Cong*  
    *CryptHOL.CryptHOL*  
    *Cyclic-Group-Ext*  
    *Discrete-Log*  
    *Number-Theory-Aux*  
    *Uniform-Sampling*

**begin**

**locale** *rivest* =

**fixes**  $q :: \text{nat}$   
    **assumes** *prime-q*: *prime*  $q$

**begin**

**lemma** *q-gt-0* [*simp*]:  $q > 0$

$\langle$ *proof* $\rangle$

**type-synonym** *ck* =  $\text{nat} \times \text{nat}$

**type-synonym** *vk* =  $\text{nat} \times \text{nat}$

**type-synonym** *plain* =  $\text{nat}$

**type-synonym** *commit* =  $\text{nat}$

**type-synonym** *opening* =  $\text{nat} \times \text{nat}$

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

**where**

*key-gen* = *do* {

```

a :: nat ← sample-uniform q;
b :: nat ← sample-uniform q;
x1 :: nat ← sample-uniform q;
let y1 = (a * x1 + b) mod q;
return-spmf ((a,b), (x1,y1))}

```

**definition** *commit* :: *ck* ⇒ *plain* ⇒ (*commit* × *opening*) *spmf*

**where**

```

commit ck m = do {
  let (a,b) = ck;
  return-spmf ((m + a) mod q, (a,b))}

```

**fun** *verify* :: *vk* ⇒ *plain* ⇒ *commit* ⇒ *opening* ⇒ *bool*

**where**

```

verify (x1,y1) m c (a,b) = (((c = (m + a) mod q) ∧ (y1 = (a * x1 + b) mod
q))

```

**definition** *valid-msg* :: *plain* ⇒ *bool*

**where** *valid-msg* *msg* ≡ *msg* ∈ {..*q*}

**sublocale** *rivest-commit*: *abstract-commitment key-gen commit verify valid-msg*  
⟨*proof*⟩

**lemma** *abstract-correct*: *rivest-commit.correct*  
⟨*proof*⟩

**lemma** *rivest-hiding*: (*spmf* (*rivest-commit.hiding-game-ind-cpa* *A*) *True* − 1/2 = 0)

**including** *monad-normalisation*

⟨*proof*⟩

**lemma** *rivest-perfect-hiding*: *rivest-commit.perfect-hiding-ind-cpa* *A*  
⟨*proof*⟩

**lemma** *samp-uni-break'*:

**assumes** *fst-cond*: *m* ≠ *m'* ∧ *valid-msg* *m* ∧ *valid-msg* *m'*

**and** *c*: *c* = (*m* + *a*) mod *q* ∧ *y1* = (*a* \* *x1* + *b*) mod *q*

**and** *c'*: *c* = (*m'* + *a'*) mod *q* ∧ *y1* = (*a'* \* *x1* + *b'*) mod *q*

**and** *x1*: *x1* < *q*

**shows** *x1* = (if (*a* mod *q* > *a'* mod *q*) then nat ((int *b'* − int *b*) \* (inverse (nat ((int *a* mod *q* − int *a'* mod *q*) mod *qq*) mod *q*) else

nat ((int *b* − int *b'*) \* (inverse (nat ((int *a'* mod *q* − int *a* mod *q*) mod *q*

*q*) mod *q*))

⟨*proof*⟩

**lemma** *samp-uni-spmf-mod-q*:

**shows** *spmf* (*sample-uniform* *q*) (*x* mod *q*) = 1/*q*

⟨*proof*⟩

```

lemma spmf-samp-uni-eq-return-bool-mod:
  shows spmf (do {
    x1 ← sample-uniform q;
    return-spmf (int x1 = y mod q)} True = 1 / q
  <proof>

lemma bind-game-le-inv-q:
  shows spmf (rivest-commit.bind-game A) True ≤ 1 / q
  <proof>
  including monad-normalisation
  <proof>

lemma rivest-bind:
  shows rivest-commit.bind-advantage A ≤ 1 / q
  <proof>

end

locale rivest-asymp =
  fixes q :: nat ⇒ nat
  assumes rivest:  $\bigwedge \eta. \text{rivest } (q \ \eta)$ 
begin

  sublocale rivest q  $\eta$  for  $\eta$  <proof>

theorem rivest-correct:
  shows rivest-commit.correct n
  <proof>

theorem rivest-perfect-hiding-asym:
  assumes lossless-A: rivest-commit.lossless (A n)
  shows rivest-commit.perfect-hiding-ind-cpa n (A n)
  <proof>

theorem rivest-binding-asym:
  assumes negligible ( $\lambda n. 1 / (q \ n)$ )
  shows negligible ( $\lambda n. \text{rivest-commit.bind-advantage } n \ (\mathcal{A} \ n)$ )
  <proof>

end

end

```

## 2 $\Sigma$ -Protocols

$\Sigma$ -protocols were first introduced as an abstract notion by Cramer [9]. We point the reader to [7] for a good introduction to the primitive as well as

informal proofs of many of the constructions we formalise in this work. In particular the construction of commitment schemes from  $\Sigma$ -protocols and the construction of compound AND and OR statements.

In this section we define  $\Sigma$ -protocols then provide a general proof that they can be used to construct commitment schemes. Defining security for  $\Sigma$ -protocols uses a mixture of the game-based and simulation-based paradigms. The honest verifier zero knowledge property is considered using simulation-based proof, thus we follow the simulation-based formalisation of [1] and [4].

## 2.1 Defining $\Sigma$ -protocols

**theory** *Sigma-Protocols* **imports**

*CryptHOL.CryptHOL*

*Commitment-Schemes*

**begin**

**type-synonym** ('*msg'*, '*challenge'*', '*response'*') *conv-tuple* = ('*msg'*'  $\times$  '*challenge'*'  $\times$  '*response'*')  
 $\times$  '*response'*'

**type-synonym** ('*msg'*', '*response'*') *sim-out* = ('*msg'*'  $\times$  '*response'*')  
 $\times$  '*response'*'

**type-synonym** ('*pub-input'*', '*msg'*', '*challenge'*', '*response'*', '*witness'*') *prover-adversary*

= '*pub-input'*'  $\Rightarrow$  ('*msg'*', '*challenge'*', '*response'*') *conv-tuple*  
 $\Rightarrow$  ('*msg'*', '*challenge'*', '*response'*') *conv-tuple*  $\Rightarrow$  '*witness'*' *spmf*

**locale**  $\Sigma$ -*protocols-base* =

**fixes** *init* :: '*pub-input*'  $\Rightarrow$  '*witness*'  $\Rightarrow$  ('*rand*'  $\times$  '*msg*') *spmf* — initial message in  $\Sigma$ -protocol

**and** *response* :: '*rand*'  $\Rightarrow$  '*witness*'  $\Rightarrow$  '*challenge*'  $\Rightarrow$  '*response*' *spmf*

**and** *check* :: '*pub-input*'  $\Rightarrow$  '*msg*'  $\Rightarrow$  '*challenge*'  $\Rightarrow$  '*response*'  $\Rightarrow$  *bool*

**and** *Rel* :: ('*pub-input*'  $\times$  '*witness*') *set* — The relation the  $\Sigma$  protocol is considered over

**and** *S-raw* :: '*pub-input*'  $\Rightarrow$  '*challenge*'  $\Rightarrow$  ('*msg*', '*response*') *sim-out* *spmf* — Simulator for the HVZK property

**and** *Ass* :: ('*pub-input*', '*msg*', '*challenge*', '*response*', '*witness*') *prover-adversary* — Special soundness adversary

**and** *challenge-space* :: '*challenge*' *set* — The set of valid challenges

**and** *valid-pub* :: '*pub-input*' *set*

**assumes** *domain-subset-valid-pub*: *Domain Rel*  $\subseteq$  *valid-pub*

**begin**

**lemma** **assumes**  $x \in \text{Domain } \textit{Rel}$  **shows**  $\exists w. (x, w) \in \textit{Rel}$   
*<proof>*

The language defined by the relation is the set of all public inputs such that there exists a witness that satisfies the relation.

**definition**  $L \equiv \{x. \exists w. (x, w) \in Rel\}$

The first property of  $\Sigma$ -protocols we consider is completeness, we define a probabilistic programme that runs the components of the protocol and outputs the boolean defined by the check algorithm.

**definition** *completeness-game*  $:: 'pub-input \Rightarrow 'witness \Rightarrow 'challenge \Rightarrow bool \text{ spmf}$   
**where** *completeness-game*  $h w e = do \{$   
 $(r, a) \leftarrow init h w;$   
 $z \leftarrow response r w e;$   
 $return-spmf (check h a e z)\}$

We define completeness as the probability that the completeness-game returns true for all challenges assuming the relation holds on  $h$  and  $w$ .

**definition** *completeness*  $\equiv (\forall h w e. (h, w) \in Rel \longrightarrow e \in challenge-space \longrightarrow spmf (completeness-game h w e) True = 1)$

Second we consider the honest verifier zero knowledge property (HVZK). To reason about this we construct the real view of the  $\Sigma$ -protocol given a challenge  $e$  as input.

**definition**  $R :: 'pub-input \Rightarrow 'witness \Rightarrow 'challenge \Rightarrow ('msg, 'challenge, 'response) \text{ conv-tuple spmf}$   
**where**  $R h w e = do \{$   
 $(r, a) \leftarrow init h w;$   
 $z \leftarrow response r w e;$   
 $return-spmf (a, e, z)\}$

**definition**  $S \text{ where } S h e = map-spmf (\lambda (a, z). (a, e, z)) (S\text{-raw } h e)$

**lemma** *lossless-S-raw-imp-lossless-S*:  $lossless-spmf (S\text{-raw } h e) \longrightarrow lossless-spmf (S h e)$   
 $\langle proof \rangle$

The HVZK property requires that the simulator's output distribution is equal to the real views output distribution.

**definition**  $HVZK \equiv (\forall e \in challenge-space.$   
 $(\forall (h, w) \in Rel. R h w e = S h e)$   
 $\wedge (\forall h \in valid\text{-pub}. \forall (a, z) \in set-spmf (S\text{-raw } h e). check h a e z))$

The final property to consider is that of special soundness. This says that given two valid transcripts such that the challenges are not equal there exists an adversary  $\mathcal{A}ss$  that can output the witness.

**definition** *special-soundness*  $\equiv (\forall h e e' a z z'. h \in valid\text{-pub} \longrightarrow e \in challenge-space \longrightarrow e' \in challenge-space \longrightarrow e \neq e'$   
 $\longrightarrow check h a e z \longrightarrow check h a e' z' \longrightarrow (lossless-spmf (\mathcal{A}ss h (a, e, z)$   
 $(a, e', z')) \wedge$   
 $(\forall w' \in set-spmf (\mathcal{A}ss h (a, e, z) (a, e', z')). (h, w') \in Rel))$

**lemma** *special-soundness-alt*:

*special-soundness*  $\longleftrightarrow$   
 $(\forall h a e z e' z'. e \in \text{challenge-space} \longrightarrow e' \in \text{challenge-space} \longrightarrow h \in \text{valid-pub}$   
 $\longrightarrow e \neq e' \longrightarrow \text{check } h a e z \wedge \text{check } h a e' z'$   
 $\longrightarrow \text{bind-spmf } (\text{Ass } h (a, e, z) (a, e', z')) (\lambda w'. \text{return-spmf } ((h, w') \in$   
 $\text{Rel})) = \text{return-spmf True}$   
 $\langle \text{proof} \rangle$

**definition**  $\Sigma$ -*protocol*  $\equiv \text{completeness} \wedge \text{special-soundness} \wedge \text{HVZK}$

General lemmas

**lemma** *lossless-complete-game*:

**assumes** *lossless-init*:  $\forall h w. \text{lossless-spmf } (\text{init } h w)$   
**and** *lossless-response*:  $\forall r w e. \text{lossless-spmf } (\text{response } r w e)$   
**shows** *lossless-spmf* (*completeness-game*  $h w e$ )  
 $\langle \text{proof} \rangle$

**lemma** *complete-game-return-true*:

**assumes**  $(h, w) \in \text{Rel}$   
**and** *completeness*  
**and** *lossless-init*:  $\forall h w. \text{lossless-spmf } (\text{init } h w)$   
**and** *lossless-response*:  $\forall r w e. \text{lossless-spmf } (\text{response } r w e)$   
**and**  $e \in \text{challenge-space}$   
**shows** *completeness-game*  $h w e = \text{return-spmf True}$   
 $\langle \text{proof} \rangle$

**lemma** *HVZK-unfold1*:

**assumes**  $\Sigma$ -*protocol*  
**shows**  $\forall h w e. (h, w) \in \text{Rel} \longrightarrow e \in \text{challenge-space} \longrightarrow R h w e = S h e$   
 $\langle \text{proof} \rangle$

**lemma** *HVZK-unfold2*:

**assumes**  $\Sigma$ -*protocol*  
**shows**  $\forall h e \text{out}. e \in \text{challenge-space} \longrightarrow h \in \text{valid-pub} \longrightarrow \text{out} \in \text{set-spmf}$   
 $(S\text{-raw } h e) \longrightarrow \text{check } h (\text{fst out}) e (\text{snd out})$   
 $\langle \text{proof} \rangle$

**lemma** *HVZK-unfold2-alt*:

**assumes**  $\Sigma$ -*protocol*  
**shows**  $\forall h a e z. e \in \text{challenge-space} \longrightarrow h \in \text{valid-pub} \longrightarrow (a, z) \in \text{set-spmf}$   
 $(S\text{-raw } h e) \longrightarrow \text{check } h a e z$   
 $\langle \text{proof} \rangle$

**end**

## 2.2 Commitments from $\Sigma$ -protocols

In this section we provide a general proof that  $\Sigma$ -protocols can be used to construct commitment schemes. We follow the construction given by Damgard in [7].

**locale**  $\Sigma$ -protocols-to-commitments =  $\Sigma$ -protocols-base *init response check Rel S-raw Ass challenge-space valid-pub*

```

for init :: 'pub-input  $\Rightarrow$  'witness  $\Rightarrow$  ('rand  $\times$  'msg) spmf
  and response :: 'rand  $\Rightarrow$  'witness  $\Rightarrow$  'challenge  $\Rightarrow$  'response spmf
  and check :: 'pub-input  $\Rightarrow$  'msg  $\Rightarrow$  'challenge  $\Rightarrow$  'response  $\Rightarrow$  bool
  and Rel :: ('pub-input  $\times$  'witness) set
  and S-raw :: 'pub-input  $\Rightarrow$  'challenge  $\Rightarrow$  ('msg, 'response) sim-out spmf
  and Ass :: ('pub-input, 'msg, 'challenge, 'response, 'witness) prover-adversary
  and challenge-space :: 'challenge set
  and valid-pub :: 'pub-input set
  and G :: ('pub-input  $\times$  'witness) spmf — generates pairs that satisfy the relation
  +
assumes  $\Sigma$ -prot:  $\Sigma$ -protocol — assume we have a  $\Sigma$ -protocol
  and set-spmf-G-rel [simp]:  $(h,w) \in \text{set-spmf } G \implies (h,w) \in \text{Rel}$  — the generator
has the desired property
  and lossless-G: lossless-spmf G
  and lossless-init: lossless-spmf (init h w)
  and lossless-response: lossless-spmf (response r w e)
begin

```

**lemma** *set-spmf-G-domain-rel* [*simp*]:  $(h,w) \in \text{set-spmf } G \implies h \in \text{Domain Rel}$   
*<proof>*

**lemma** *set-spmf-G-L* [*simp*]:  $(h,w) \in \text{set-spmf } G \implies h \in L$   
*<proof>*

We define the advantage associated with the hard relation, this is used in the proof of the binding property where we reduce the binding advantage to the relation advantage.

**definition** *rel-game* :: ('pub-input  $\Rightarrow$  'witness spmf)  $\Rightarrow$  bool spmf  
**where** *rel-game*  $\mathcal{A} = \text{TRY do}$  {  
 (*h,w*)  $\leftarrow$  *G*;  
*w'*  $\leftarrow$   $\mathcal{A}$  *h*;  
*return-spmf*  $((h,w') \in \text{Rel})$  *ELSE return-spmf False*

**definition** *rel-advantage* :: ('pub-input  $\Rightarrow$  'witness spmf)  $\Rightarrow$  real  
**where** *rel-advantage*  $\mathcal{A} \equiv \text{spmf } (\text{rel-game } \mathcal{A}) \text{ True}$

We now define the algorithms that define the commitment scheme constructed from a  $\Sigma$ -protocol.

**definition** *key-gen* :: ('pub-input  $\times$  ('pub-input  $\times$  'witness)) spmf  
**where**  
*key-gen* = *do* {

$(x,w) \leftarrow G;$   
 $\text{return-spmf } (x, (x,w))\}$

**definition**  $\text{commit} :: 'pub\text{-input} \Rightarrow 'challenge \Rightarrow ('msg \times 'response) \text{ spmf}$   
**where**  
 $\text{commit } x \ e = \text{do } \{$   
 $(a,e,z) \leftarrow S \ x \ e;$   
 $\text{return-spmf } (a, z)\}$

**definition**  $\text{verify} :: ('pub\text{-input} \times 'witness) \Rightarrow 'challenge \Rightarrow 'msg \Rightarrow 'response \Rightarrow$   
 $\text{bool}$   
**where**  $\text{verify } x \ e \ a \ z = (\text{check } (\text{fst } x) \ a \ e \ z)$

We allow the adversary to output any message, so this means the type constraint is enough

**definition**  $\text{valid-msg } m = (m \in \text{challenge-space})$

Showing the construction of a commitment scheme from a  $\Sigma$ -protocol is a valid commitment scheme is trivial.

**sublocale**  $\text{abstract-com: abstract-commitment key-gen commit verify valid-msg } \langle \text{proof} \rangle$

**Correctness lemma**  $\text{commit-correct:}$   
**shows**  $\text{abstract-com.correct}$   
**including**  $\text{monad-normalisation}$   
 $\langle \text{proof} \rangle$

**The hiding property** We first show we have perfect hiding with respect to the hiding game that allows the adversary to choose the messages that are committed to, this is akin to the ind-cpa game for encryption schemes.

**lemma**  $\text{perfect-hiding:}$   
**shows**  $\text{abstract-com.perfect-hiding-ind-cpa } \mathcal{A}$   
**including**  $\text{monad-normalisation}$   
 $\langle \text{proof} \rangle$

We reduce the security of the binding property to the relation advantage. To do this we first construct an adversary that interacts with the relation game. This adversary succeeds if the binding adversary succeeds.

**definition**  $\text{adversary} :: ('pub\text{-input} \Rightarrow ('msg \times 'challenge \times 'response \times 'challenge$   
 $\times 'response) \text{ spmf}) \Rightarrow 'pub\text{-input} \Rightarrow 'witness \text{ spmf}$   
**where**  $\text{adversary } \mathcal{A} \ x = \text{do } \{$   
 $(c, e, ez, e', ez') \leftarrow \mathcal{A} \ x;$   
 $\text{Ass } x \ (c,e,ez) \ (c,e',ez')\}$

**lemma**  $\text{bind-advantage:}$   
**shows**  $\text{abstract-com.bind-advantage } \mathcal{A} \leq \text{rel-advantage } (\text{adversary } \mathcal{A})$   
 $\langle \text{proof} \rangle$



**end**

**end**

### 2.3 Schnorr $\Sigma$ -protocol

In this section we show the Schnorr protocol [11] is a  $\Sigma$ -protocol and then use it to construct a commitment scheme. The security statements for the resulting commitment scheme come for free from our general proof of the construction.

**theory** *Schnorr-Sigma-Commit* **imports**

*Commitment-Schemes*

*Sigma-Protocols*

*Cyclic-Group-Ext*

*Discrete-Log*

*Number-Theory-Aux*

*Uniform-Sampling*

*HOL-Number-Theory.Cong*

**begin**

**locale** *schnorr-base* =

**fixes**  $\mathcal{G} :: 'grp$  *cyclic-group* (**structure**)

**assumes** *prime-order*: *prime* (*order*  $\mathcal{G}$ )

**begin**

**lemma** *order-gt-0* [*simp*]: *order*  $\mathcal{G} > 0$

*<proof>*

The types for the  $\Sigma$ -protocol.

**type-synonym** *witness* = *nat*

**type-synonym** *rand* = *nat*

**type-synonym** *'grp' msg* = *'grp'*

**type-synonym** *response* = *nat*

**type-synonym** *challenge* = *nat*

**type-synonym** *'grp' pub-in* = *'grp'*

**definition** *R-DL* :: (*'grp pub-in*  $\times$  *witness*) *set*

**where** *R-DL* =  $\{(h, w). h = \mathbf{g} [\wedge] w\}$

**definition** *init* :: *'grp pub-in*  $\Rightarrow$  *witness*  $\Rightarrow$  (*rand*  $\times$  *'grp msg*) *spmf*

**where** *init* *h w* = *do* {

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

*return-spmf* (*r*,  $\mathbf{g} [\wedge] r$ )}

**lemma** *lossless-init*: *lossless-spmf* (*init* *h w*)

*<proof>*

**definition** *response* *r w c* = *return-spmf* ( $(w * c + r) \bmod (\text{order } \mathcal{G})$ )

**lemma** *lossless-response*: *lossless-spmf* (response  $r$   $w$   $c$ )  
 ⟨proof⟩

**definition**  $G :: ('grp$  pub-in  $\times$  witness) *spmf*  
 where  $G = do$  {  
    $w \leftarrow sample$ -uniform (order  $\mathcal{G}$ );  
   return-spmf ( $\mathbf{g} [\uparrow] w, w$ )}

**lemma** *lossless-G*: *lossless-spmf*  $G$   
 ⟨proof⟩

**definition** *challenge-space* =  $\{..< order \mathcal{G}\}$

**definition** *check* ::  $'grp$  pub-in  $\Rightarrow 'grp$  msg  $\Rightarrow$  challenge  $\Rightarrow$  response  $\Rightarrow$  bool  
 where *check*  $h$   $a$   $e$   $z = (a \otimes (h [\uparrow] e) = \mathbf{g} [\uparrow] z \wedge a \in carrier \mathcal{G})$

**definition**  $S2 :: 'grp \Rightarrow$  challenge  $\Rightarrow ('grp$  msg, response) *sim-out* *spmf*  
 where  $S2$   $h$   $e = do$  {  
    $c \leftarrow sample$ -uniform (order  $\mathcal{G}$ );  
   let  $a = \mathbf{g} [\uparrow] c \otimes (inv (h [\uparrow] e))$ ;  
   return-spmf ( $a, c$ )}

**definition** *ss-adversary* ::  $'grp \Rightarrow ('grp$  msg, challenge, response) *conv-tuple*  $\Rightarrow$   
 ( $'grp$  msg, challenge, response) *conv-tuple*  $\Rightarrow$  nat *spmf*  
 where *ss-adversary*  $x$   $c1$   $c2 = do$  {  
   let ( $a, e, z$ ) =  $c1$ ;  
   let ( $a', e', z'$ ) =  $c2$ ;  
   return-spmf (if ( $e > e'$ ) then  
     (nat ((int  $z - int z'$ ) \* inverse (( $e - e'$ )) (order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ))  
   else  
     (nat ((int  $z' - int z$ ) \* inverse (( $e' - e$ )) (order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ )))}

**definition** *valid-pub* = carrier  $\mathcal{G}$

We now use the Schnorr  $\Sigma$ -protocol use Schnorr to construct a commitment scheme.

**type-synonym**  $'grp'$  *ck* =  $'grp'$   
**type-synonym**  $'grp'$  *vk* =  $'grp'$   $\times$  nat  
**type-synonym** *plain* = nat  
**type-synonym**  $'grp'$  *commit* =  $'grp'$   
**type-synonym** *opening* = nat

The adversary we use in the discrete log game to reduce the binding property to the discrete log assumption.

**definition** *dis-log-A* :: ( $'grp$  *ck*, *plain*,  $'grp$  *commit*, *opening*) *bind-adversary*  $\Rightarrow$   
 $'grp$  *ck*  $\Rightarrow$  nat *spmf*  
 where *dis-log-A*  $\mathcal{A}$   $h = do$  {

```

(c, e, z, e', z') ←  $\mathcal{A}$  h;
- :: unit ← assert-spmf (e > e' ∧ ¬ [e = e'] (mod order  $\mathcal{G}$ ) ∧ (gcd (e - e') (order
 $\mathcal{G}$ ) = 1) ∧ c ∈ carrier  $\mathcal{G}$ );
- :: unit ← assert-spmf (((c ⊗ h [∧] e) =  $\mathbf{g}$  [∧] z) ∧ (c ⊗ h [∧] e') =  $\mathbf{g}$  [∧] z');
return-spmf (nat ((int z - int z') * inverse ((e - e') (order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ )))}

```

**sublocale** *discrete-log*: *dis-log*  $\mathcal{G}$   
 ⟨proof⟩

**end**

**locale** *schnorr-sigma-protocol* = *schnorr-base* + *cyclic-group*  $\mathcal{G}$   
**begin**

**sublocale** *Schnorr- $\Sigma$* :  *$\Sigma$ -protocols-base* *init* *response* *check* *R-DL* *S2* *ss-adversary*  
*challenge-space* *valid-pub*  
 ⟨proof⟩

The Schnorr  $\Sigma$ -protocol is complete.

**lemma** *completeness*: *Schnorr- $\Sigma$ .completeness*  
 ⟨proof⟩

The next two lemmas help us rewrite terms in the proof of honest verifier zero knowledge.

**lemma** *zr-rewrite*:  
**assumes**  $z = (x * c + r) \text{ mod } (\text{order } \mathcal{G})$   
**and**  $r < \text{order } \mathcal{G}$   
**shows**  $(z + (\text{order } \mathcal{G}) * x * c - x * c) \text{ mod } (\text{order } \mathcal{G}) = r$   
 ⟨proof⟩

**lemma** *h-sub-rewrite*:  
**assumes**  $h = \mathbf{g} [\wedge] x$   
**and**  $z < \text{order } \mathcal{G}$   
**shows**  $\mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * x * c - x * c)) = \mathbf{g} [\wedge] z \otimes \text{inv } (h [\wedge] c)$   
 (**is** ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *hvzk-R-rewrite-grp*:  
**fixes**  $x \ c \ r :: \text{nat}$   
**assumes**  $r < \text{order } \mathcal{G}$   
**shows**  $\mathbf{g} [\wedge] (((x * c + \text{order } \mathcal{G} - r) \text{ mod } \text{order } \mathcal{G} + \text{order } \mathcal{G} * x * c - x * c) \text{ mod } \text{order } \mathcal{G}) = \text{inv } \mathbf{g} [\wedge] r$   
 (**is** ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *hv-zk*:  
**assumes**  $(h, x) \in \text{R-DL}$   
**shows** *Schnorr- $\Sigma$ .R*  $h \ x \ c = \text{Schnorr- $\Sigma$ .S}$   $h \ c$   
**including** *monad-normalisation*

*<proof>*

We can now prove that honest verifier zero knowledge holds for the Schnorr  $\Sigma$ -protocol.

**lemma** *honest-verifier-ZK*:  
**shows** *Schnorr- $\Sigma$ .HVZK*  
*<proof>*

It is left to prove the special soundness property. First we prove a lemma we use to rewrite a term in the special soundness proof and then prove the property itself.

**lemma** *ss-rewrite*:  
**assumes**  $e' < e$   
**and**  $e < \text{order } \mathcal{G}$   
**and**  $a\text{-mem}: a \in \text{carrier } \mathcal{G}$   
**and**  $h\text{-mem}: h \in \text{carrier } \mathcal{G}$   
**and**  $a: a \otimes h [\cdot] e = \mathbf{g} [\cdot] z$   
**and**  $a': a \otimes h [\cdot] e' = \mathbf{g} [\cdot] z'$   
**shows**  $h = \mathbf{g} [\cdot] ((\text{int } z - \text{int } z') * \text{inverse } ((e - e') (\text{order } \mathcal{G}) \text{ mod int } (\text{order } \mathcal{G})))$   
*<proof>*

The special soundness property for the Schnorr  $\Sigma$ -protocol.

**lemma** *special-soundness*:  
**shows** *Schnorr- $\Sigma$ .special-soundness*  
*<proof>*

We are now able to prove that the Schnorr  $\Sigma$ -protocol is a  $\Sigma$ -protocol, the proof comes from the properties of completeness, HVZK and special soundness we have previously proven.

**theorem** *sigma-protocol*:  
**shows** *Schnorr- $\Sigma$ . $\Sigma$ -protocol*  
*<proof>*

Having proven the  $\Sigma$ -protocol property is satisfied we can show the commitment scheme we construct from the Schnorr  $\Sigma$ -protocol has the desired properties. This result comes with very little proof effort as we can instantiate our general proof.

**sublocale** *Schnorr- $\Sigma$ -commit:  $\Sigma$ -protocols-to-commitments init response check R-DL S2 ss-adversary challenge-space valid-pub G*  
*<proof>*

**lemma** *Schnorr- $\Sigma$ -commit.abstract-com.correct*  
*<proof>*

**lemma** *Schnorr- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa  $\mathcal{A}$*   
*<proof>*

**lemma** *rel-adv-eq-dis-log-adv*:

*Schnorr- $\Sigma$ -commit.rel-advantage*  $\mathcal{A} = \text{discrete-log.}advantage \mathcal{A}$   
*<proof>*

**lemma** *bind-advantage-bound-dis-log*:

*Schnorr- $\Sigma$ -commit.abstract-com.bind-advantage*  $\mathcal{A} \leq \text{discrete-log.}advantage (\text{Schnorr-}\Sigma\text{-commit.adversary } \mathcal{A})$   
*<proof>*

**end**

**locale** *schnorr-asymp* =

**fixes**  $\mathcal{G} :: \text{nat} \Rightarrow \text{'grp cyclic-group}$

**assumes** *schnorr*:  $\bigwedge \eta. \text{schnorr-sigma-protocol } (\mathcal{G} \eta)$

**begin**

**sublocale** *schnorr-sigma-protocol*  $\mathcal{G} \eta$  **for**  $\eta$

*<proof>*

The  $\Sigma$ -protocol statement comes easily in the asymptotic setting.

**theorem** *sigma-protocol*:

**shows** *Schnorr- $\Sigma$ . $\Sigma$ -protocol*  $n$

*<proof>*

We now show the statements of security for the commitment scheme in the asymptotic setting, the main difference is that we are able to show the binding advantage is negligible in the security parameter.

**lemma** *asymp-correct*: *Schnorr- $\Sigma$ -commit.abstract-com.correct*  $n$

*<proof>*

**lemma** *asymp-perfect-hiding*: *Schnorr- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa*  
 $n (\mathcal{A} n)$

*<proof>*

**lemma** *asymp-computational-binding*:

**assumes** *negligible*  $(\lambda n. \text{discrete-log.}advantage n (\text{Schnorr-}\Sigma\text{-commit.adversary } n (\mathcal{A} n)))$

**shows** *negligible*  $(\lambda n. \text{Schnorr-}\Sigma\text{-commit.abstract-com.bind-advantage } n (\mathcal{A} n))$

*<proof>*

**end**

**end**

## 2.4 Chaum-Pedersen $\Sigma$ -protocol

The Chaum-Pedersen  $\Sigma$ -protocol [6] considers a relation of equality of discrete logs.

```

theory Chaum-Pedersen-Sigma-Commit imports
  Commitment-Schemes
  Sigma-Protocols
  Cyclic-Group-Ext
  Discrete-Log
  Number-Theory-Aux
  Uniform-Sampling
begin

locale chaum-ped- $\Sigma$ -base =
  fixes  $\mathcal{G} :: 'grp$  cyclic-group (structure)
    and  $x :: nat$ 
  assumes prime-order: prime (order  $\mathcal{G}$ )
begin

definition  $g' = \mathbf{g} [\wedge] x$ 

lemma or-gt-1: order  $\mathcal{G} > 1$ 
   $\langle proof \rangle$ 

lemma or-gt-0 [simp]: order  $\mathcal{G} > 0$ 
   $\langle proof \rangle$ 

type-synonym witness = nat
type-synonym rand = nat
type-synonym 'grp' msg = 'grp'  $\times$  'grp'
type-synonym response = nat
type-synonym challenge = nat
type-synonym 'grp' pub-in = 'grp'  $\times$  'grp'

definition  $G = do \{$ 
   $w \leftarrow sample-uniform (order \mathcal{G});$ 
   $return-spmf ((\mathbf{g} [\wedge] w, g' [\wedge] w), w)$ 
lemma lossless-G: lossless-spmf  $G$ 
   $\langle proof \rangle$ 

definition challenge-space =  $\{.. < order \mathcal{G}\}$ 

definition  $init :: 'grp$  pub-in  $\Rightarrow$  witness  $\Rightarrow$  (rand  $\times$  'grp msg) spmf
  where  $init h w = do \{$ 
     $let (h, h') = h;$ 
     $r \leftarrow sample-uniform (order \mathcal{G});$ 
     $return-spmf (r, \mathbf{g} [\wedge] r, g' [\wedge] r)$ 
lemma lossless-init: lossless-spmf ( $init h w$ )
   $\langle proof \rangle$ 

definition  $response r w e = return-spmf ((w * e + r) mod (order \mathcal{G}))$ 

```

**lemma** *lossless-response*: *lossless-spmf* (response  $r$   $w$   $e$ )

*<proof>*

**definition** *check* :: *'grp pub-in*  $\Rightarrow$  *'grp msg*  $\Rightarrow$  *challenge*  $\Rightarrow$  *response*  $\Rightarrow$  *bool*

**where** *check*  $h$   $a$   $e$   $z = (fst\ a \otimes (fst\ h\ [\uparrow]\ e) = \mathbf{g}\ [\uparrow]\ z \wedge snd\ a \otimes (snd\ h\ [\uparrow]\ e) = g' [\uparrow]\ z \wedge fst\ a \in carrier\ \mathcal{G} \wedge snd\ a \in carrier\ \mathcal{G})$

**definition** *R* :: (*'grp pub-in*  $\times$  *witness*) *set*

**where**  $R = \{(h, w). (fst\ h = \mathbf{g}\ [\uparrow]\ w \wedge snd\ h = g' [\uparrow]\ w)\}$

**definition** *S2* :: *'grp pub-in*  $\Rightarrow$  *challenge*  $\Rightarrow$  (*'grp msg*, *response*) *sim-out* *spmf*

**where**  $S2\ H\ c = do\ \{$   
*let*  $(h, h') = H;$   
 $z \leftarrow (sample\ uniform\ (order\ \mathcal{G}));$   
*let*  $a = \mathbf{g}\ [\uparrow]\ z \otimes inv\ (h\ [\uparrow]\ c);$   
*let*  $a' = g' [\uparrow]\ z \otimes inv\ (h' [\uparrow]\ c);$   
*return-spmf*  $((a, a'), z)\}$

**definition** *ss-adversary* :: *'grp pub-in*  $\Rightarrow$  (*'grp msg*, *challenge*, *response*) *conv-tuple*  
 $\Rightarrow$  (*'grp msg*, *challenge*, *response*) *conv-tuple*  $\Rightarrow$  *nat* *spmf*

**where** *ss-adversary*  $x' c1 c2 = do\ \{$   
*let*  $((a, a'), e, z) = c1;$   
*let*  $((b, b'), e', z') = c2;$   
*return-spmf*  $(if\ (e\ mod\ order\ \mathcal{G} > e'\ mod\ order\ \mathcal{G})\ then\ (nat\ ((int\ z - int\ z') * (fst\ (bezw\ ((e\ mod\ order\ \mathcal{G} - e'\ mod\ order\ \mathcal{G})\ mod\ order\ \mathcal{G})\ (order\ \mathcal{G})))\ mod\ order\ \mathcal{G}))\ else\ (nat\ ((int\ z' - int\ z) * (fst\ (bezw\ ((e'\ mod\ order\ \mathcal{G} - e\ mod\ order\ \mathcal{G})\ mod\ order\ \mathcal{G})\ (order\ \mathcal{G})))\ mod\ order\ \mathcal{G})))\}$

**definition** *valid-pub* = *carrier*  $\mathcal{G} \times$  *carrier*  $\mathcal{G}$

**end**

**locale** *chaum-ped- $\Sigma$*  = *chaum-ped- $\Sigma$ -base* + *cyclic-group*  $\mathcal{G}$

**begin**

**lemma** *g'-in-carrier* [*simp*]:  $g' \in carrier\ \mathcal{G}$

*<proof>*

**sublocale** *chaum-ped-sigma*:  *$\Sigma$ -protocols-base* *init* *response* *check* *R* *S2* *ss-adversary*  
*challenge-space* *valid-pub*

*<proof>*

**lemma** *completeness*:

**shows** *chaum-ped-sigma.completeness*

*<proof>*

**lemma** *hvzk-xr'-rewrite*:

**assumes**  $r: r < \text{order } \mathcal{G}$   
**shows**  $((w*c + r) \bmod (\text{order } \mathcal{G}) \bmod (\text{order } \mathcal{G}) + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}) = r$   
**(is ?lhs = ?rhs)**  
 <proof>

**lemma hvzk-h-sub-rewrite:**  
**assumes**  $h = \mathbf{g} [\uparrow] w$   
**and**  $z: z < \text{order } \mathcal{G}$   
**shows**  $\mathbf{g} [\uparrow] ((z + (\text{order } \mathcal{G}) * w * c - w*c)) = \mathbf{g} [\uparrow] z \otimes \text{inv } (h [\uparrow] c)$   
**(is ?lhs = ?rhs)**  
 <proof>

**lemma hvzk-h-sub2-rewrite:**  
**assumes**  $h' = g' [\uparrow] w$   
**and**  $z: z < \text{order } \mathcal{G}$   
**shows**  $g' [\uparrow] ((z + (\text{order } \mathcal{G}) * w * c - w*c)) = g' [\uparrow] z \otimes \text{inv } (h' [\uparrow] c)$   
**(is ?lhs = ?rhs)**  
 <proof>

**lemma hv-zk2:**  
**assumes**  $(H, w) \in R$   
**shows**  $\text{chaum-ped-sigma.R } H w c = \text{chaum-ped-sigma.S } H c$   
**including monad-normalisation**  
 <proof>

**lemma HVZK:**  
**shows**  $\text{chaum-ped-sigma.HVZK}$   
 <proof>

**lemma ss-rewrite1:**  
**assumes**  $\text{fst } h \in \text{carrier } \mathcal{G}$   
**and**  $a \in \text{carrier } \mathcal{G}$   
**and**  $e: e < \text{order } \mathcal{G}$   
**and**  $a \otimes \text{fst } h [\uparrow] e = \mathbf{g} [\uparrow] z$   
**and**  $e': e' < e$   
**and**  $a \otimes \text{fst } h [\uparrow] e' = \mathbf{g} [\uparrow] z'$   
**shows**  $\text{fst } h = \mathbf{g} [\uparrow] ((\text{int } z - \text{int } z') * \text{inverse } (e - e') (\text{order } \mathcal{G}) \bmod \text{int } (\text{order } \mathcal{G}))$   
 <proof>

**lemma ss-rewrite2:**  
**assumes**  $\text{fst } h \in \text{carrier } \mathcal{G}$   
**and**  $\text{snd } h \in \text{carrier } \mathcal{G}$   
**and**  $a \in \text{carrier } \mathcal{G}$   
**and**  $b \in \text{carrier } \mathcal{G}$   
**and**  $e < \text{order } \mathcal{G}$   
**and**  $a \otimes \text{fst } h [\uparrow] e = \mathbf{g} [\uparrow] z$   
**and**  $b \otimes \text{snd } h [\uparrow] e = g' [\uparrow] z$



**and**  $e' < e$   
**and**  $a \otimes \text{fst } h \ [\frown] \ e' = \mathbf{g} \ [\frown] \ z'$   
**and**  $b \otimes \text{snd } h \ [\frown] \ e' = g' \ [\frown] \ z'$   
**shows**  $\text{snd } h = g' \ [\frown] \ ((\text{int } z - \text{int } z') * \text{inverse } (e - e') \ (\text{order } \mathcal{G}) \ \text{mod } \text{int } (\text{order } \mathcal{G}))$   
 $\langle \text{proof} \rangle$

**lemma** *ss-rewrite-snd-h*:

**assumes**  $e\text{-}e'\text{-mod}$ :  $e' \ \text{mod } \text{order } \mathcal{G} < e \ \text{mod } \text{order } \mathcal{G}$   
**and**  $h\text{-mem}$ :  $\text{snd } h \in \text{carrier } \mathcal{G}$   
**and**  $a\text{-mem}$ :  $\text{snd } a \in \text{carrier } \mathcal{G}$   
**and**  $a1$ :  $\text{snd } a \otimes \text{snd } h \ [\frown] \ e = g' \ [\frown] \ z$   
**and**  $a2$ :  $\text{snd } a \otimes \text{snd } h \ [\frown] \ e' = g' \ [\frown] \ z'$   
**shows**  $\text{snd } h = g' \ [\frown] \ ((\text{int } z - \text{int } z') * \text{fst } (\text{bezw } ((e \ \text{mod } \text{order } \mathcal{G} - e' \ \text{mod } \text{order } \mathcal{G}) \ \text{mod } \text{order } \mathcal{G}) \ (\text{order } \mathcal{G})) \ \text{mod } \text{int } (\text{order } \mathcal{G}))$   
 $\langle \text{proof} \rangle$

**lemma** *special-soundness*:

**shows**  $\text{chaum-ped-sigma}$ .*special-soundness*  
 $\langle \text{proof} \rangle$

**theorem**  $\Sigma$ -*protocol*:  $\text{chaum-ped-sigma}$ . $\Sigma$ -*protocol*

$\langle \text{proof} \rangle$

**sublocale**  $\text{chaum-ped-}\Sigma$ -*commit*:  $\Sigma$ -*protocols-to-commitments* *init* *response* *check*  
 $R$   $S2$  *ss-adversary* *challenge-space* *valid-pub*  $G$

$\langle \text{proof} \rangle$

**sublocale** *dis-log*: *dis-log*  $\mathcal{G}$

$\langle \text{proof} \rangle$

**sublocale** *dis-log-alt*: *dis-log-alt*  $\mathcal{G}$   $x$

$\langle \text{proof} \rangle$

**lemma** *reduction-to-dis-log*:

**shows**  $\text{chaum-ped-}\Sigma$ -*commit*.*rel-advantage*  $\mathcal{A} = \text{dis-log}$ .*advantage* ( $\text{dis-log-alt}$ .*adversary3*  $\mathcal{A}$ )  
 $\langle \text{proof} \rangle$

**lemma** *commitment-correct*:  $\text{chaum-ped-}\Sigma$ -*commit*.*abstract-com*.*correct*

$\langle \text{proof} \rangle$

**lemma**  $\text{chaum-ped-}\Sigma$ -*commit*.*abstract-com*.*perfect-hiding-ind-cpa*  $\mathcal{A}$

$\langle \text{proof} \rangle$

**lemma** *binding*:  $\text{chaum-ped-}\Sigma$ -*commit*.*abstract-com*.*bind-advantage*  $\mathcal{A} \leq \text{dis-log}$ .*advantage* ( $\text{dis-log-alt}$ .*adversary3* ( $(\text{chaum-ped-}\Sigma$ -*commit*.*adversary3*  $\mathcal{A}))$ )

$\langle \text{proof} \rangle$

**end**

```
locale chaum-ped-asymp =  
  fixes  $\mathcal{G} :: \text{nat} \Rightarrow \text{'grp cyclic-group}$   
    and  $x :: \text{nat}$   
  assumes  $cp\text{-}\Sigma: \bigwedge \eta. \text{chaum-ped-}\Sigma (\mathcal{G} \eta)$   
begin
```

```
sublocale chaum-ped-}\Sigma \mathcal{G} \eta for  $\eta$   
  \langle proof \rangle
```

The  $\Sigma$ -protocol statement comes easily in the asymptotic setting.

```
theorem sigma-protocol:  
  shows chaum-ped-sigma.}\Sigma-protocol n  
  \langle proof \rangle
```

We now show the statements of security for the commitment scheme in the asymptotic setting, the main difference is that we are able to show the binding advantage is negligible in the security parameter.

```
lemma asymp-correct: chaum-ped-}\Sigma-commit.abstract-com.correct n  
  \langle proof \rangle
```

```
lemma asymp-perfect-hiding: chaum-ped-}\Sigma-commit.abstract-com.perfect-hiding-ind-cpa  
 $n (\mathcal{A} n)$   
  \langle proof \rangle
```

```
lemma asymp-computational-binding:  
  assumes negligible  $(\lambda n. \text{dis-log.}advantage\ n (\text{dis-log-alt.adversary3}\ n ((\text{chaum-ped-}\Sigma\text{-commit.adversary}$   
 $n (\mathcal{A}\ n))))))$   
  shows negligible  $(\lambda n. \text{chaum-ped-}\Sigma\text{-commit.abstract-com.bind-}advantage\ n (\mathcal{A}$   
 $n))$   
  \langle proof \rangle
```

**end**

**end**

## 2.5 Okamoto $\Sigma$ -protocol

```
theory Okamoto-Sigma-Commit imports  
  Commitment-Schemes  
  Sigma-Protocols  
  Cyclic-Group-Ext  
  Discrete-Log  
  HOL.GCD  
  Number-Theory-Aux  
  Uniform-Sampling  
begin
```

**locale** *okamoto-base* =  
**fixes**  $\mathcal{G} :: \text{'grp cyclic-group (structure)}$   
**and**  $x :: \text{nat}$   
**assumes** *prime-order*:  $\text{prime (order } \mathcal{G})$   
**begin**

**definition**  $g' = \mathbf{g} [\wedge] x$

**lemma** *order-gt-1*:  $\text{order } \mathcal{G} > 1$   
 $\langle \text{proof} \rangle$

**lemma** *order-gt-0* [*simp*]:  $\text{order } \mathcal{G} > 0$   
 $\langle \text{proof} \rangle$

**definition** *response*  $r w e = \text{do} \{$   
 $\text{let } (r1, r2) = r;$   
 $\text{let } (x1, x2) = w;$   
 $\text{let } z1 = (e * x1 + r1) \text{ mod (order } \mathcal{G});$   
 $\text{let } z2 = (e * x2 + r2) \text{ mod (order } \mathcal{G});$   
 $\text{return-spmf } ((z1, z2))\}$

**lemma** *lossless-response*:  $\text{lossless-spmf (response } r w e)$   
 $\langle \text{proof} \rangle$

**type-synonym** *witness* =  $\text{nat} \times \text{nat}$   
**type-synonym** *rand* =  $\text{nat} \times \text{nat}$   
**type-synonym** *'grp' msg* =  $\text{'grp'}$   
**type-synonym** *response* =  $(\text{nat} \times \text{nat})$   
**type-synonym** *challenge* =  $\text{nat}$   
**type-synonym** *'grp' pub-in* =  $\text{'grp'}$

**definition** *init* ::  $\text{'grp pub-in} \Rightarrow \text{witness} \Rightarrow (\text{rand} \times \text{'grp msg}) \text{ spmf}$   
**where** *init*  $y w = \text{do} \{$   
 $\text{let } (x1, x2) = w;$   
 $r1 \leftarrow \text{sample-uniform (order } \mathcal{G});$   
 $r2 \leftarrow \text{sample-uniform (order } \mathcal{G});$   
 $\text{return-spmf } ((r1, r2), \mathbf{g} [\wedge] r1 \otimes g' [\wedge] r2)\}$

**lemma** *lossless-init*:  $\text{lossless-spmf (init } h w)$   
 $\langle \text{proof} \rangle$

**definition** *check* ::  $\text{'grp pub-in} \Rightarrow \text{'grp msg} \Rightarrow \text{challenge} \Rightarrow \text{response} \Rightarrow \text{bool}$   
**where** *check*  $h a e z = (\mathbf{g} [\wedge] (\text{fst } z) \otimes g' [\wedge] (\text{snd } z) = a \otimes (h [\wedge] e) \wedge a \in \text{carrier } \mathcal{G})$

**definition** *R* ::  $(\text{'grp pub-in} \times \text{witness}) \text{ set}$   
**where**  $R \equiv \{(h, w). (h = \mathbf{g} [\wedge] (\text{fst } w) \otimes g' [\wedge] (\text{snd } w))\}$

**definition** *G* ::  $(\text{'grp pub-in} \times \text{witness}) \text{ spmf}$

**where**  $G = do \{$   
 $w1 \leftarrow sample-uniform (order \mathcal{G});$   
 $w2 \leftarrow sample-uniform (order \mathcal{G});$   
 $return-spmf (g [\wedge] w1 \otimes g' [\wedge] w2 , (w1,w2))\}$

**definition**  $challenge-space = \{..< order \mathcal{G}\}$

**lemma**  $lossless-G: lossless-spmf G$   
 $\langle proof \rangle$

**definition**  $S2 :: 'grp pub-in \Rightarrow challenge \Rightarrow ('grp msg, response) sim-out spmf$   
**where**  $S2 h c = do \{$   
 $z1 \leftarrow sample-uniform (order \mathcal{G});$   
 $z2 \leftarrow sample-uniform (order \mathcal{G});$   
 $let a = (g [\wedge] z1 \otimes g' [\wedge] z2) \otimes (inv h [\wedge] c);$   
 $return-spmf (a, (z1,z2))\}$

**definition**  $R2 :: 'grp pub-in \Rightarrow witness \Rightarrow challenge \Rightarrow ('grp msg, challenge, re-$   
 $sponse) conv-tuple spmf$   
**where**  $R2 h w c = do \{$   
 $let (x1,x2) = w;$   
 $r1 \leftarrow sample-uniform (order \mathcal{G});$   
 $r2 \leftarrow sample-uniform (order \mathcal{G});$   
 $let z1 = (c * x1 + r1) mod (order \mathcal{G});$   
 $let z2 = (c * x2 + r2) mod (order \mathcal{G});$   
 $return-spmf (g [\wedge] r1 \otimes g' [\wedge] r2 ,c,(z1,z2))\}$

**definition**  $ss-adversary :: 'grp \Rightarrow ('grp msg, challenge, response) conv-tuple \Rightarrow$   
 $('grp msg, challenge, response) conv-tuple \Rightarrow (nat \times nat) spmf$   
**where**  $ss-adversary y c1 c2 = do \{$   
 $let (a, e, (z1,z2)) = c1;$   
 $let (a', e', (z1',z2')) = c2;$   
 $return-spmf (if (e > e') then (nat ((int z1 - int z1') * inverse (e - e') (order$   
 $\mathcal{G}) mod order \mathcal{G})) else$   
 $(nat ((int z1' - int z1) * inverse (e' - e) (order \mathcal{G}) mod order$   
 $\mathcal{G})),$   
 $if (e > e') then (nat ((int z2 - int z2') * inverse (e - e') (order$   
 $\mathcal{G}) mod order \mathcal{G})) else$   
 $(nat ((int z2' - int z2) * inverse (e' - e) (order \mathcal{G}) mod order$   
 $\mathcal{G}))))\}$

**definition**  $valid-pub = carrier \mathcal{G}$   
**end**

**locale**  $okamoto = okamoto-base + cyclic-group \mathcal{G}$   
**begin**

**lemma**  $g'-in-carrier [simp]: g' \in carrier \mathcal{G}$   
 $\langle proof \rangle$

**sublocale**  $\Sigma$ -protocols-base:  $\Sigma$ -protocols-base init response check R S2 ss-adversary challenge-space valid-pub  
 ⟨proof⟩

**lemma**  $\Sigma$ -protocols-base.R h w c = R2 h w c  
 ⟨proof⟩

**lemma** completeness:  
 shows  $\Sigma$ -protocols-base.completeness  
 ⟨proof⟩

**lemma** hvzk-z-r:  
 assumes  $r1: r1 < \text{order } \mathcal{G}$   
 shows  $r1 = ((r1 + c * (x1 :: \text{nat})) \bmod (\text{order } \mathcal{G}) + \text{order } \mathcal{G} * c * x1 - c * x1) \bmod (\text{order } \mathcal{G})$   
 ⟨proof⟩

**lemma** hvzk-z1-r1-tuple-rewrite:  
 assumes  $r1: r1 < \text{order } \mathcal{G}$   
 shows  $(\mathbf{g} [\wedge] r1 \otimes g' [\wedge] r2, c, (r1 + c * x1) \bmod \text{order } \mathcal{G}, (r2 + c * x2) \bmod \text{order } \mathcal{G}) =$   
 $(\mathbf{g} [\wedge] (((r1 + c * x1) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * c * x1 - c * x1) \bmod \text{order } \mathcal{G})$   
 $\otimes g' [\wedge] r2, c, (r1 + c * x1) \bmod \text{order } \mathcal{G}, (r2 + c * x2) \bmod \text{order } \mathcal{G})$   
 ⟨proof⟩

**lemma** hvzk-z2-r2-tuple-rewrite:  
 assumes  $xb < \text{order } \mathcal{G}$   
 shows  $(\mathbf{g} [\wedge] (((x' + xa * x1) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x1 - xa * x1) \bmod \text{order } \mathcal{G})$   
 $\otimes g' [\wedge] xb, xa, (x' + xa * x1) \bmod \text{order } \mathcal{G}, (xb + xa * x2) \bmod \text{order } \mathcal{G}) =$   
 $(\mathbf{g} [\wedge] (((x' + xa * x1) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x1 - xa * x1) \bmod \text{order } \mathcal{G})$   
 $\otimes g' [\wedge] (((xb + xa * x2) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x2 - xa * x2) \bmod \text{order } \mathcal{G}), xa, (x' + xa * x1) \bmod \text{order } \mathcal{G}, (xb + xa * x2) \bmod \text{order } \mathcal{G})$   
 ⟨proof⟩

**lemma** hvzk-sim-inverse-rewrite:  
 assumes  $h: h = \mathbf{g} [\wedge] (x1 :: \text{nat}) \otimes g' [\wedge] (x2 :: \text{nat})$   
 shows  $\mathbf{g} [\wedge] (((z1 :: \text{nat}) + \text{order } \mathcal{G} * c * x1 - c * x1) \bmod (\text{order } \mathcal{G}))$   
 $\otimes g' [\wedge] (((z2 :: \text{nat}) + \text{order } \mathcal{G} * c * x2 - c * x2) \bmod (\text{order } \mathcal{G}))$   
 $= (\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2) \otimes (\text{inv } h [\wedge] c)$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** hv-zk:

**assumes**  $h = \mathbf{g} [\uparrow] x1 \otimes g' [\uparrow] x2$   
**shows**  $\Sigma\text{-protocols-base.R } h (x1, x2) c = \Sigma\text{-protocols-base.S } h c$   
**including** *monad-normalisation*  
*<proof>*

**lemma** *HVZK*:  
**shows**  $\Sigma\text{-protocols-base.HVZK}$   
*<proof>*

**lemma** *ss-rewrite*:  
**assumes**  $h \in \text{carrier } \mathcal{G}$   
**and**  $a \in \text{carrier } \mathcal{G}$   
**and**  $e < \text{order } \mathcal{G}$   
**and**  $\mathbf{g} [\uparrow] z1 \otimes g' [\uparrow] z1' = a \otimes h [\uparrow] e$   
**and**  $e' < e$   
**and**  $\mathbf{g} [\uparrow] z2 \otimes g' [\uparrow] z2' = a \otimes h [\uparrow] e'$   
**shows**  $h = \mathbf{g} [\uparrow] ((\text{int } z1 - \text{int } z2) * \text{fst } (\text{bezw } (e - e') (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) \otimes g' [\uparrow] ((\text{int } z1' - \text{int } z2') * \text{fst } (\text{bezw } (e - e') (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G}))$   
*<proof>*

**lemma**  
**assumes** *h-mem*:  $h \in \text{carrier } \mathcal{G}$   
**and** *a-mem*:  $a \in \text{carrier } \mathcal{G}$   
**and**  $a: \mathbf{g} [\uparrow] \text{fst } z \otimes g' [\uparrow] \text{snd } z = a \otimes h [\uparrow] e$   
**and**  $a': \mathbf{g} [\uparrow] \text{fst } z' \otimes g' [\uparrow] \text{snd } z' = a \otimes h [\uparrow] e'$   
**and** *e-e'-mod*:  $e' \text{ mod order } \mathcal{G} < e \text{ mod order } \mathcal{G}$   
**shows**  $h = \mathbf{g} [\uparrow] ((\text{int } (\text{fst } z) - \text{int } (\text{fst } z')) * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}) (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G})) \otimes g' [\uparrow] ((\text{int } (\text{snd } z) - \text{int } (\text{snd } z')) * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}) (\text{order } \mathcal{G})) \text{ mod int } (\text{order } \mathcal{G}))$   
*<proof>*

**lemma** *special-soundness*:  
**shows**  $\Sigma\text{-protocols-base.special-soundness}$   
*<proof>*

**theorem**  $\Sigma\text{-protocol}$ :  
**shows**  $\Sigma\text{-protocols-base.}\Sigma\text{-protocol}$   
*<proof>*

**sublocale** *okamoto- $\Sigma$ -commit*:  $\Sigma\text{-protocols-to-commitments init response check R S2 ss-adversary challenge-space valid-pub } \mathcal{G}$   
*<proof>*

**sublocale** *dis-log*:  $\text{dis-log } \mathcal{G}$   
*<proof>*

**sublocale** *dis-log-alt*:  $\text{dis-log-alt } \mathcal{G} x$   
*<proof>*

**lemma** *reduction-to-dis-log*:  
**shows** *okamoto- $\Sigma$ -commit.rel-advantage*  $\mathcal{A} = \text{dis-log.}advantage (\text{dis-log-alt.adversary2 } \mathcal{A})$   
 $\langle proof \rangle$   
**including** *monad-normalisation*  
 $\langle proof \rangle$

**lemma** *commitment-correct: okamoto- $\Sigma$ -commit.abstract-com.correct*  
 $\langle proof \rangle$

**lemma** *okamoto- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa*  $\mathcal{A}$   
 $\langle proof \rangle$

**lemma** *binding*:  
**shows** *okamoto- $\Sigma$ -commit.abstract-com.bind-advantage*  $\mathcal{A}$   
 $\leq \text{dis-log.}advantage (\text{dis-log-alt.adversary2 } (\text{okamoto-}\Sigma\text{-commit.adversary } \mathcal{A}))$   
 $\langle proof \rangle$

**end**

**locale** *okamoto-asymp* =  
**fixes**  $\mathcal{G} :: \text{nat} \Rightarrow 'grp \text{ cyclic-group}$   
**and**  $x :: \text{nat}$   
**assumes** *okamoto*:  $\bigwedge \eta. \text{okamoto } (\mathcal{G} \ \eta)$   
**begin**

**sublocale** *okamoto*  $\mathcal{G} \ \eta$  **for**  $\eta$   
 $\langle proof \rangle$

The  $\Sigma$ -protocol statement comes easily in the asymptotic setting.

**theorem** *sigma-protocol*:  
**shows**  *$\Sigma$ -protocols-base. $\Sigma$ -protocol*  $n$   
 $\langle proof \rangle$

We now show the statements of security for the commitment scheme in the asymptotic setting, the main difference is that we are able to show the binding advantage is negligible in the security parameter.

**lemma** *asypm-correct: okamoto- $\Sigma$ -commit.abstract-com.correct*  $n$   
 $\langle proof \rangle$

**lemma** *asypm-perfect-hiding: okamoto- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa*  
 $n \ (\mathcal{A} \ n)$   
 $\langle proof \rangle$

**lemma** *asypm-computational-binding*:  
**assumes** *negligible*  $(\lambda n. \text{dis-log.}advantage \ n \ (\text{dis-log-alt.adversary2 } (\text{okamoto-}\Sigma\text{-commit.adversary } n \ (\mathcal{A} \ n))))$

```

shows negligible ( $\lambda n.$  okamoto- $\Sigma$ -commit.abstract-com.bind-advantage  $n$  ( $\mathcal{A}$   $n$ ))
  <proof>

end

end

theory Xor imports
  HOL-Algebra.Complete-Lattice
  CryptHOL.Misc-CryptHOL
begin

no-notation
  bot-class.bot ( $\perp$ ) and
  top-class.top ( $\top$ ) and
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65)

context bounded-lattice begin

lemma top-join [simp]:  $x \in \text{carrier } L \implies \top \sqcup x = \top$ 
  <proof>

lemma join-top [simp]:  $x \in \text{carrier } L \implies x \sqcup \top = \top$ 
  <proof>

lemma bot-join [simp]:  $x \in \text{carrier } L \implies \perp \sqcup x = x$ 
  <proof>

lemma join-bot [simp]:  $x \in \text{carrier } L \implies x \sqcup \perp = x$ 
  <proof>

lemma bot-meet [simp]:  $x \in \text{carrier } L \implies \perp \sqcap x = \perp$ 
  <proof>

lemma meet-bot [simp]:  $x \in \text{carrier } L \implies x \sqcap \perp = \perp$ 
  <proof>

lemma top-meet [simp]:  $x \in \text{carrier } L \implies \top \sqcap x = x$ 
  <proof>

lemma meet-top [simp]:  $x \in \text{carrier } L \implies x \sqcap \top = x$ 
  <proof>

lemma join-idem [simp]:  $x \in \text{carrier } L \implies x \sqcup x = x$ 
  <proof>

lemma meet-idem [simp]:  $x \in \text{carrier } L \implies x \sqcap x = x$ 
  <proof>

```



**lemma** *meet-leftcomm*:  $x \sqcap (y \sqcup z) = y \sqcap (x \sqcup z)$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 $z \in \text{carrier } L$

*<proof>*

**lemma** *join-leftcomm*:  $x \sqcup (y \sqcap z) = y \sqcup (x \sqcap z)$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$

*<proof>*

**lemmas** *meet-ac = meet-assoc meet-comm meet-leftcomm*

**lemmas** *join-ac = join-assoc join-comm join-leftcomm*

**end**

**record** *'a boolean-algebra* = *'a gorder* +  
*compl* :: *'a*  $\Rightarrow$  *'a* ( $-_1$  1000)

**definition** *xor* :: (*'a, 'b*) *boolean-algebra-scheme*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a* (**infixr**  $\oplus_1$  100)

**where**

$x \oplus y = (x \sqcup y) \sqcap (- (x \sqcap y))$  **for**  $L$  (**structure**)

**locale** *boolean-algebra* = *bounded-lattice*  $L$

**for**  $L$  (**structure**) +

**assumes** *compl-closed* [*intro, simp*]:  $x \in \text{carrier } L \Longrightarrow - x \in \text{carrier } L$

**and** *meet-compl-bot* [*simp*]:  $x \in \text{carrier } L \Longrightarrow - x \sqcap x = \perp$

**and** *join-compl-top* [*simp*]:  $x \in \text{carrier } L \Longrightarrow - x \sqcup x = \top$

**and** *join-meet-distrib1*:  $\llbracket x \in \text{carrier } L; y \in \text{carrier } L; z \in \text{carrier } L \rrbracket \Longrightarrow x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

**begin**

**lemma** *join-meet-distrib2*:  $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$

**if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$

*<proof>*

**lemma** *meet-join-distrib1*:  $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

**if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$

*<proof>*

**lemma** *meet-join-distrib2*:  $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$

**if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$

*<proof>*

**lemmas** *join-meet-distrib* = *join-meet-distrib1 join-meet-distrib2*

**lemmas** *meet-join-distrib* = *meet-join-distrib1 meet-join-distrib2*

**lemmas** *distrib* = *join-meet-distrib meet-join-distrib*

**lemma** *meet-compl2-bot* [*simp*]:  $x \in \text{carrier } L \Longrightarrow x \sqcap - x = \perp$

*<proof>*

**lemma** *join-compl2-top* [*simp*]:  $x \in \text{carrier } L \implies x \sqcup - x = \top$   
*<proof>*

**lemma** *compl-unique*:  
 **assumes**  $x \sqcap y = \perp$   
 **and**  $x \sqcup y = \top$   
 **and** [*simp*]:  $x \in \text{carrier } L \ y \in \text{carrier } L$   
 **shows**  $- x = y$   
*<proof>*

**lemma** *double-compl* [*simp*]:  $- (- x) = x$  **if** [*simp*]:  $x \in \text{carrier } L$   
*<proof>*

**lemma** *compl-eq-compl-iff* [*simp*]:  $- x = - y \iff x = y$  **if**  $x \in \text{carrier } L \ y \in \text{carrier } L$   
*<proof>*

**lemma** *compl-bot-eq* [*simp*]:  $- \perp = \top$   
*<proof>*

**lemma** *compl-top-eq* [*simp*]:  $- \top = \perp$   
*<proof>*

**lemma** *compl-inf* [*simp*]:  $- (x \sqcap y) = - x \sqcup - y$  **if** [*simp*]:  $x \in \text{carrier } L \ y \in \text{carrier } L$   
*<proof>*

**lemma** *compl-sup* [*simp*]:  $- (x \sqcup y) = - x \sqcap - y$  **if**  $x \in \text{carrier } L \ y \in \text{carrier } L$   
*<proof>*

**lemma** *compl-mono*:  
 **assumes**  $x \sqsubseteq y$   
 **and**  $x \in \text{carrier } L \ y \in \text{carrier } L$   
 **shows**  $- y \sqsubseteq - x$   
*<proof>*

**lemma** *compl-le-compl-iff* [*simp*]:  $- x \sqsubseteq - y \iff y \sqsubseteq x$  **if**  $x \in \text{carrier } L \ y \in \text{carrier } L$   
*<proof>*

**lemma** *compl-le-swap1*:  
 **assumes**  $y \sqsubseteq - x \ x \in \text{carrier } L \ y \in \text{carrier } L$   
 **shows**  $x \sqsubseteq - y$   
*<proof>*

**lemma** *compl-le-swap2*:  
 **assumes**  $- y \sqsubseteq x \ x \in \text{carrier } L \ y \in \text{carrier } L$

**shows**  $-x \sqsubseteq y$   
*<proof>*

**lemma** *join-compl-top-left1* [*simp*]:  $-x \sqcup (x \sqcup y) = \top$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
*<proof>*

**lemma** *join-compl-top-left2* [*simp*]:  $x \sqcup (-x \sqcup y) = \top$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
*<proof>*

**lemma** *meet-compl-bot-left1* [*simp*]:  $-x \sqcap (x \sqcap y) = \perp$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
*<proof>*

**lemma** *meet-compl-bot-left2* [*simp*]:  $x \sqcap (-x \sqcap y) = \perp$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
*<proof>*

**lemma** *meet-compl-bot-right* [*simp*]:  $x \sqcap (y \sqcap -x) = \perp$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
*<proof>*

**lemma** *xor-closed* [*intro, simp*]:  $\llbracket x \in \text{carrier } L; y \in \text{carrier } L \rrbracket \implies x \oplus y \in \text{carrier } L$   
*<proof>*

**lemma** *xor-comm*:  $\llbracket x \in \text{carrier } L; y \in \text{carrier } L \rrbracket \implies x \oplus y = y \oplus x$   
*<proof>*

**lemma** *xor-assoc*:  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$   
**if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$   
*<proof>*

**lemma** *xor-left-comm*:  $x \oplus (y \oplus z) = y \oplus (x \oplus z)$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$   
*<proof>*

**lemma** [*simp*]:  
**assumes**  $x \in \text{carrier } L$   
**shows** *xor-bot*:  $x \oplus \perp = x$   
**and** *bot-xor*:  $\perp \oplus x = x$   
**and** *xor-top*:  $x \oplus \top = -x$   
**and** *top-xor*:  $\top \oplus x = -x$   
*<proof>*

**lemma** *xor-inverse* [*simp*]:  $x \oplus x = \perp$  **if**  $x \in \text{carrier } L$   
*<proof>*

**lemma** *xor-left-inverse* [simp]:  $x \oplus x \oplus y = y$  if  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 ⟨proof⟩

**lemmas** *xor-ac = xor-assoc xor-comm xor-left-comm*

**lemma** *inj-on-xor*: *inj-on*  $((\oplus) x)$   $(\text{carrier } L)$  if  $x \in \text{carrier } L$   
 ⟨proof⟩

**lemma** *surj-xor*:  $(\oplus) x \in \text{carrier } L = \text{carrier } L$  if [simp]:  $x \in \text{carrier } L$   
 ⟨proof⟩

**lemma** *one-time-pad*: *map-spmf*  $((\oplus) x)$   $(\text{spmf-of-set } (\text{carrier } L)) = \text{spmf-of-set}$   
 $(\text{carrier } L)$   
 if  $x \in \text{carrier } L$   
 ⟨proof⟩

**end**

**end**

## 2.6 $\Sigma$ -AND statements

**theory** *Sigma-AND* imports

*Sigma-Protocols*

*Xor*

**begin**

**locale**  $\Sigma$ -AND-base =  $\Sigma 0$ :  $\Sigma$ -protocols-base *init0 response0 check0 Rel0 S0-raw*  
*Ass0 carrier L valid-pub0*  
 +  $\Sigma 1$ :  $\Sigma$ -protocols-base *init1 response1 check1 Rel1 S1-raw Ass1 carrier L valid-pub1*  
**for** *init1* ::  $'pub1 \Rightarrow 'witness1 \Rightarrow ('rand1 \times 'msg1)$  *spmf*  
**and** *response1* ::  $'rand1 \Rightarrow 'witness1 \Rightarrow 'bool \Rightarrow 'response1$  *spmf*  
**and** *check1* ::  $'pub1 \Rightarrow 'msg1 \Rightarrow 'bool \Rightarrow 'response1 \Rightarrow 'bool$   
**and** *Rel1* ::  $('pub1 \times 'witness1)$  *set*  
**and** *S1-raw* ::  $'pub1 \Rightarrow 'bool \Rightarrow ('msg1 \times 'response1)$  *spmf*  
**and** *Ass1* ::  $'pub1 \Rightarrow 'msg1 \times 'bool \times 'response1 \Rightarrow 'msg1 \times 'bool \times 'response1$   
 $\Rightarrow 'witness1$  *spmf*  
**and** *challenge-space1* ::  $'bool$  *set*  
**and** *valid-pub1* ::  $'pub1$  *set*  
**and** *init0* ::  $'pub0 \Rightarrow 'witness0 \Rightarrow ('rand0 \times 'msg0)$  *spmf*  
**and** *response0* ::  $'rand0 \Rightarrow 'witness0 \Rightarrow 'bool \Rightarrow 'response0$  *spmf*  
**and** *check0* ::  $'pub0 \Rightarrow 'msg0 \Rightarrow 'bool \Rightarrow 'response0 \Rightarrow 'bool$   
**and** *Rel0* ::  $('pub0 \times 'witness0)$  *set*  
**and** *S0-raw* ::  $'pub0 \Rightarrow 'bool \Rightarrow ('msg0 \times 'response0)$  *spmf*  
**and** *Ass0* ::  $'pub0 \Rightarrow 'msg0 \times 'bool \times 'response0 \Rightarrow 'msg0 \times 'bool \times 'response0$   
 $\Rightarrow 'witness0$  *spmf*  
**and** *challenge-space0* ::  $'bool$  *set*  
**and** *valid-pub0* ::  $'pub0$  *set*  
**and** *G* ::  $((pub0 \times pub1) \times (witness0 \times witness1))$  *spmf*

**and**  $L :: 'bool$  boolean-algebra (**structure**)  
 +  
**assumes**  $\Sigma$ -prot1:  $\Sigma 1.$  $\Sigma$ -protocol  
**and**  $\Sigma$ -prot0:  $\Sigma 0.$  $\Sigma$ -protocol  
**and**  $lossless$ -init:  $lossless$ -spmf (init0 h0 w0)  $lossless$ -spmf (init1 h1 w1)  
**and**  $lossless$ -response:  $lossless$ -spmf (response0 r0 w0 e0)  $lossless$ -spmf (response1  
 $r1 w1 e1$ )  
**and**  $lossless$ -S:  $lossless$ -spmf (S0 h0 e0)  $lossless$ -spmf (S1 h1 e1)  
**and**  $lossless$ -Ass:  $lossless$ -spmf (Ass0 x0 (a0,e,z0) (a0,e',z0'))  $lossless$ -spmf  
(Ass1 x1 (a1,e,z1) (a1,e',z1'))  
**and**  $lossless$ -G:  $lossless$ -spmf G  
**and**  $set$ -spmf-G [simp]:  $(h,w) \in set$ -spmf G  $\implies$  Rel h w  
**begin**

**definition** challenge-space = carrier L

**definition** Rel-AND ::  $(('pub0 \times 'pub1) \times 'witness0 \times 'witness1)$  set  
**where** Rel-AND =  $\{((x0,x1), (w0,w1)). ((x0,w0) \in Rel0 \wedge (x1,w1) \in Rel1)\}$

**definition** init-AND ::  $('pub0 \times 'pub1) \Rightarrow ('witness0 \times 'witness1) \Rightarrow (('rand0 \times$   
 $'rand1) \times 'msg0 \times 'msg1)$  spmf  
**where** init-AND X W = do {  
 let  $(x0, x1) = X$ ;  
 let  $(w0,w1) = W$ ;  
 $(r0, a0) \leftarrow init0$  x0 w0;  
 $(r1, a1) \leftarrow init1$  x1 w1;  
 return-spmf  $((r0,r1), (a0,a1))$ }

**lemma**  $lossless$ -init-AND:  $lossless$ -spmf (init-AND X W)  
 ⟨proof⟩

**definition** response-AND ::  $('rand0 \times 'rand1) \Rightarrow ('witness0 \times 'witness1) \Rightarrow 'bool$   
 $\Rightarrow ('response0 \times 'response1)$  spmf  
**where** response-AND R W s = do {  
 let  $(r0,r1) = R$ ;  
 let  $(w0,w1) = W$ ;  
 $z0 \leftarrow response0$  r0 w0 s;  
 $z1 :: 'response1 \leftarrow response1$  r1 w1 s;  
 return-spmf  $(z0,z1)$ }

**lemma**  $lossless$ -response-AND:  $lossless$ -spmf (response-AND R W s)  
 ⟨proof⟩

**fun** check-AND ::  $('pub0 \times 'pub1) \Rightarrow ('msg0 \times 'msg1) \Rightarrow 'bool \Rightarrow ('response0 \times$   
 $'response1) \Rightarrow bool$   
**where** check-AND  $(x0,x1) (a0,a1) s (z0,z1) = (check0$  x0 a0 s z0  $\wedge$  check1 x1  
 $a1 s z1)$

**definition** S-AND ::  $'pub0 \times 'pub1 \Rightarrow 'bool \Rightarrow (('msg0 \times 'msg1) \times 'response0 \times$

```

'response1) spmf
  where S-AND X e = do {
    let (x0,x1) = X;
    (a0, z0) ← S0-raw x0 e;
    (a1, z1) ← S1-raw x1 e;
    return-spmf ((a0,a1),(z0,z1))}

fun Ass-AND :: 'pub0 × 'pub1 ⇒ ('msg0 × 'msg1) × 'bool × 'response0 ×
'response1 ⇒ ('msg0 × 'msg1) × 'bool × 'response0 × 'response1 ⇒ ('witness0
× 'witness1) spmf
  where Ass-AND (x0,x1) ((a0,a1), e, (z0,z1)) ((a0',a1'), e', (z0',z1')) = do {
    w0 :: 'witness0 ← Ass0 x0 (a0,e,z0) (a0',e',z0');
    w1 ← Ass1 x1 (a1,e,z1) (a1',e',z1');
    return-spmf (w0,w1)}

definition valid-pub-AND = {(x0,x1). x0 ∈ valid-pub0 ∧ x1 ∈ valid-pub1}

sublocale Σ-AND: Σ-protocols-base init-AND response-AND check-AND Rel-AND
S-AND Ass-AND challenge-space valid-pub-AND
  ⟨proof⟩

end

locale Σ-AND = Σ-AND-base +
  assumes set-spmf-G-L: ((x0, x1), w0, w1) ∈ set-spmf G ⇒ ((x0, x1), (w0,w1))
  ∈ Rel-AND
begin

lemma hvzk:
  assumes Rel-AND: ((x0,x1), (w0,w1)) ∈ Rel-AND
  and e ∈ challenge-space
  shows Σ-AND.R (x0,x1) (w0,w1) e = Σ-AND.S (x0,x1) e
  including monad-normalisation
  ⟨proof⟩

lemma HVZK: Σ-AND.HVZK
  ⟨proof⟩

lemma correct:
  assumes Rel-AND: ((x0,x1), (w0,w1)) ∈ Rel-AND
  and e ∈ challenge-space
  shows Σ-AND.completeness-game (x0,x1) (w0,w1) e = return-spmf True
  including monad-normalisation
  ⟨proof⟩

lemma completeness: Σ-AND.completeness
  ⟨proof⟩

lemma ss:

```

**assumes**  $e\text{-neg-}e'$ :  $s \neq s'$   
**and**  $\text{valid-pub}$ :  $(x0,x1) \in \text{valid-pub-AND}$   
**and**  $\text{challenge-space}$ :  $s \in \text{challenge-space}$   $s' \in \text{challenge-space}$   
**and**  $\text{check-AND}$   $(x0,x1)$   $(a0,a1)$   $s$   $(z0,z1)$   
**and**  $\text{check-AND}$   $(x0,x1)$   $(a0,a1)$   $s'$   $(z0',z1')$   
**shows**  $\text{lossless-spmf}$   $(\text{Ass-AND}$   $(x0,x1)$   $((a0,a1), s, (z0,z1))$   $((a0,a1), s', (z0',z1'))$ )  
 $\wedge (\forall w' \in \text{set-spmf}$   $(\text{Ass-AND}$   $(x0,x1)$   $((a0,a1), s, (z0,z1))$   $((a0,a1), s', (z0',z1'))$ ).  $((x0,x1), w') \in \text{Rel-AND}$ )  
 $\langle \text{proof} \rangle$

**lemma**  $\text{special-soundness}$ :  
**shows**  $\Sigma\text{-AND}$ . $\text{special-soundness}$   
 $\langle \text{proof} \rangle$

**theorem**  $\Sigma\text{-protocol}$ :  
**shows**  $\Sigma\text{-AND}$ . $\Sigma\text{-protocol}$   
 $\langle \text{proof} \rangle$

**sublocale**  $\text{AND-}\Sigma\text{-commit}$ :  $\Sigma\text{-protocols-to-commitments}$   $\text{init-AND}$   $\text{response-AND}$   $\text{check-AND}$   $\text{Rel-AND}$   $\text{S-AND}$   $\text{Ass-AND}$   $\text{challenge-space}$   $\text{valid-pub-AND}$   $G$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{AND-}\Sigma\text{-commit}$ . $\text{abstract-com}$ . $\text{correct}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{AND-}\Sigma\text{-commit}$ . $\text{abstract-com}$ . $\text{perfect-hiding-ind-cpa}$   $\mathcal{A}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{bind-advantage-bound-dis-log}$ :  
**shows**  $\text{AND-}\Sigma\text{-commit}$ . $\text{abstract-com}$ . $\text{bind-advantage}$   $\mathcal{A} \leq \text{AND-}\Sigma\text{-commit}$ . $\text{rel-advantage}$   $(\text{AND-}\Sigma\text{-commit}$ . $\text{adversary}$   $\mathcal{A})$   
 $\langle \text{proof} \rangle$

**end**

**end**

## 2.7 $\Sigma\text{-OR}$ statements

**theory**  $\Sigma\text{-OR}$  **imports**  
 $\Sigma\text{-Protocols}$   
 $\text{Xor}$   
**begin**

**locale**  $\Sigma\text{-OR-base}$  =  $\Sigma0$ :  $\Sigma\text{-protocols-base}$   $\text{init0}$   $\text{response0}$   $\text{check0}$   $\text{Rel0}$   $\text{S0-raw}$   $\text{Ass0}$   
 $\text{carrier}$   $L$   $\text{valid-pub0}$   
 $+ \Sigma1$ :  $\Sigma\text{-protocols-base}$   $\text{init1}$   $\text{response1}$   $\text{check1}$   $\text{Rel1}$   $\text{S1-raw}$   $\text{Ass1}$   $\text{carrier}$   $L$   $\text{valid-pub1}$   
**for**  $\text{init1} :: 'pub1 \Rightarrow 'witness1 \Rightarrow ('rand1 \times 'msg1)$   $\text{spmf}$

```

and response1 :: 'rand1 ⇒ 'witness1 ⇒ 'bool ⇒ 'response1 spmf
and check1 :: 'pub1 ⇒ 'msg1 ⇒ 'bool ⇒ 'response1 ⇒ bool
and Rel1 :: ('pub1 × 'witness1) set
and S1-raw :: 'pub1 ⇒ 'bool ⇒ ('msg1 × 'response1) spmf
and Ass1 :: 'pub1 ⇒ 'msg1 × 'bool × 'response1 ⇒ 'msg1 × 'bool × 'response1
⇒ 'witness1 spmf
and challenge-space1 :: 'bool set
and valid-pub1 :: 'pub1 set
and init0 :: 'pub0 ⇒ 'witness0 ⇒ ('rand0 × 'msg0) spmf
and response0 :: 'rand0 ⇒ 'witness0 ⇒ 'bool ⇒ 'response0 spmf
and check0 :: 'pub0 ⇒ 'msg0 ⇒ 'bool ⇒ 'response0 ⇒ bool
and Rel0 :: ('pub0 × 'witness0) set
and S0-raw :: 'pub0 ⇒ 'bool ⇒ ('msg0 × 'response0) spmf
and Ass0 :: 'pub0 ⇒ 'msg0 × 'bool × 'response0 ⇒ 'msg0 × 'bool × 'response0
⇒ 'witness0 spmf
and challenge-space0 :: 'bool set
and valid-pub0 :: 'pub0 set
and G :: (('pub0 × 'pub1) × ('witness0 + 'witness1)) spmf
and L :: 'bool boolean-algebra (structure)
+
assumes Σ-prot1: Σ1.Σ-protocol
and Σ-prot0: Σ0.Σ-protocol
and lossless-init: lossless-spmf (init0 h0 w0) lossless-spmf (init1 h1 w1)
and lossless-response: lossless-spmf (response0 r0 w0 e0) lossless-spmf (response1
r1 w1 e1)
and lossless-S: lossless-spmf (S0 h0 e0) lossless-spmf (S1 h1 e1)
and finite-L: finite (carrier L)
and carrier-L-not-empty: carrier L ≠ {}
and lossless-G: lossless-spmf G
begin

inductive-set Rel-OR :: (('pub0 × 'pub1) × ('witness0 + 'witness1)) set where
  Rel-OR-I0: ((x0, x1), Inl w0) ∈ Rel-OR if (x0, w0) ∈ Rel0 ∧ x1 ∈ valid-pub1
| Rel-OR-I1: ((x0, x1), Inr w1) ∈ Rel-OR if (x1, w1) ∈ Rel1 ∧ x0 ∈ valid-pub0

inductive-simps Rel-OR-simps [simp]:
  ((x0, x1), Inl w0) ∈ Rel-OR
  ((x0, x1), Inr w1) ∈ Rel-OR

lemma Domain-Rel-cases:
assumes (x0,x1) ∈ Domain Rel-OR
shows (∃ w0. (x0,w0) ∈ Rel0 ∧ x1 ∈ valid-pub1) ∨ (∃ w1. (x1,w1) ∈ Rel1 ∧
x0 ∈ valid-pub0)
  ⟨proof⟩

lemma set-spmf-lists-sample [simp]: set-spmf (spmf-of-set (carrier L)) = (carrier
L)
  ⟨proof⟩

```



**definition** *challenge-space* = carrier *L*

**fun** *init-OR* :: ('pub0 × 'pub1) ⇒ ('witness0 + 'witness1) ⇒ (((('rand0 × 'bool × 'response1 + 'rand1 × 'bool × 'response0)) × 'msg0 × 'msg1)) spmf  
**where** *init-OR* (*x0,x1*) (*Inl w0*) = do {  
 (*r0,a0*) ← *init0* *x0 w0*;  
*e1* ← *spmf-of-set* (*carrier L*);  
(*a1, e'1, z1*) ←  $\Sigma 1.S$  *x1 e1*;  
*return-spmf* (*Inl* (*r0, e1, z1*), *a0, a1*)} |  
*init-OR* (*x0, x1*) (*Inr w1*) = do {  
(*r1, a1*) ← *init1* *x1 w1*;  
*e0* ← *spmf-of-set* (*carrier L*);  
(*a0, e'0, z0*) ←  $\Sigma 0.S$  *x0 e0*;  
*return-spmf* (*Inr* (*r1, e0, z0*), *a0, a1*))}

**lemma** *lossless- $\Sigma$ -S*: *lossless-spmf* ( $\Sigma 1.S$  *x1 e1*) *lossless-spmf* ( $\Sigma 0.S$  *x0 e0*)  
<proof>

**lemma** *lossless-init-OR*: *lossless-spmf* (*init-OR* (*x0,x1*) *w*)  
<proof>

**fun** *response-OR* :: (('rand0 × 'bool × 'response1 + 'rand1 × 'bool × 'response0))  
⇒ ('witness0 + 'witness1)  
⇒ 'bool ⇒ (('bool × 'response0) × ('bool × 'response1)) spmf  
**where** *response-OR* (*Inl* (*r0, e-1, z1*)) (*Inl w0*) *s* = do {  
*let* *e0* = *s* ⊕ *e-1*;  
*z0* ← *response0* *r0 w0 e0*;  
*return-spmf* ((*e0,z0*), (*e-1,z1*))} |  
*response-OR* (*Inr* (*r1, e-0, z0*)) (*Inr w1*) *s* = do {  
*let* *e1* = *s* ⊕ *e-0*;  
*z1* ← *response1* *r1 w1 e1*;  
*return-spmf* ((*e-0, z0*), (*e1, z1*))}

**definition** *check-OR* :: ('pub0 × 'pub1) ⇒ ('msg0 × 'msg1) ⇒ 'bool ⇒ (('bool × 'response0) × ('bool × 'response1)) ⇒ bool  
**where** *check-OR* *X A s Z*  
= (*s* = (*fst* (*fst Z*)) ⊕ (*fst* (*snd Z*)))  
∧ (*fst* (*fst Z*)) ∈ *challenge-space* ∧ (*fst* (*snd Z*)) ∈ *challenge-space*  
∧ *check0* (*fst X*) (*fst A*) (*fst* (*fst Z*)) (*snd* (*fst Z*)) ∧ *check1* (*snd X*) (*snd A*) (*fst* (*snd Z*)) (*snd* (*snd Z*)))

**lemma** *check-OR* (*x0,x1*) (*a0,a1*) *s* ((*e0,z0*), (*e1,z1*))  
= (*s* = *e0* ⊕ *e1*)  
∧ *e0* ∈ *challenge-space* ∧ *e1* ∈ *challenge-space*  
∧ *check0* *x0 a0 e0 z0* ∧ *check1* *x1 a1 e1 z1*)  
<proof>

**fun** *S-OR* **where** *S-OR* (*x0,x1*) *c* = do {  
*e1* ← *spmf-of-set* (*carrier L*);

$(a1, e1', z1) \leftarrow \Sigma 1.S x1 e1;$   
 $let e0 = c \oplus e1;$   
 $(a0, e0', z0) \leftarrow \Sigma 0.S x0 e0;$   
 $let z = ((e0', z0), (e1', z1));$   
 $return-spmf ((a0, a1), z)$

**definition**  $Ass-OR'$  ::  $'pub0 \times 'pub1 \Rightarrow ('msg0 \times 'msg1) \times 'bool \times ('bool \times 'response0) \times 'bool \times 'response1$   
 $\Rightarrow ('msg0 \times 'msg1) \times 'bool \times ('bool \times 'response0) \times 'bool \times 'response1 \Rightarrow ('witness0 + 'witness1) spmf$   
**where**  $Ass-OR' X C1 C2 = TRY do \{$   
 $- :: unit \leftarrow assert-spmf ((fst (fst (snd (snd C1)))) \neq (fst (fst (snd (snd C2))));$   
 $w0 :: 'witness0 \leftarrow Ass0 (fst X) (fst (fst C1), fst (fst (snd (snd C1))), snd (fst (snd (snd C1)))) (fst (fst C2), fst (fst (snd (snd C2))), snd (fst (snd (snd C2))));$   
 $return-spmf ((Inl w0) :: ('witness0 + 'witness1) spmf) \}$  *ELSE*  $do \{$   
 $w1 :: 'witness1 \leftarrow Ass1 (snd X) (snd (fst C1), fst (snd (snd (snd C1))), snd (snd (snd (snd C1)))) (snd (fst C2), fst (snd (snd (snd C2))), snd (snd (snd (snd C2))));$   
 $(return-spmf ((Inr w1) :: ('witness0 + 'witness1) spmf)) \}$

**definition**  $Ass-OR$  ::  $'pub0 \times 'pub1 \Rightarrow ('msg0 \times 'msg1) \times 'bool \times ('bool \times 'response0) \times 'bool \times 'response1$   
 $\Rightarrow ('msg0 \times 'msg1) \times 'bool \times ('bool \times 'response0) \times 'bool \times 'response1 \Rightarrow ('witness0 + 'witness1) spmf$   
**where**  $Ass-OR X C1 C2 = do \{$   
 $if ((fst (fst (snd (snd C1)))) \neq (fst (fst (snd (snd C2)))) then do$   
 $\{w0 :: 'witness0 \leftarrow Ass0 (fst X) (fst (fst C1), fst (fst (snd (snd C1))), snd (fst (snd (snd C1)))) (fst (fst C2), fst (fst (snd (snd C2))), snd (fst (snd (snd C2))));$   
 $return-spmf (Inl w0) \}$   
 $else$   
 $do \{w1 :: 'witness1 \leftarrow Ass1 (snd X) (snd (fst C1), fst (snd (snd (snd C1))), snd (snd (snd (snd C1)))) (snd (fst C2), fst (snd (snd (snd C2))), snd (snd (snd (snd C2))));$   
 $return-spmf (Inr w1) \}$

**lemma**  $Ass-OR-alt-def$ :  $Ass-OR (x0, x1) ((a0, a1), s, (e0, z0), e1, z1) ((a0, a1), s', (e0', z0'), e1', z1')$   
 $= do \{$   
 $if (e0 \neq e0') then do \{w0 :: 'witness0 \leftarrow Ass0 x0 (a0, e0, z0) (a0, e0', z0');$   
 $return-spmf (Inl w0) \}$   
 $else do \{w1 :: 'witness1 \leftarrow Ass1 x1 (a1, e1, z1) (a1, e1', z1'); return-spmf (Inr w1) \}$   
 $\langle proof \rangle$

**definition**  $valid-pub-OR = \{(x0, x1). x0 \in valid-pub0 \wedge x1 \in valid-pub1\}$

**sublocale**  $\Sigma-OR$ :  $\Sigma$ -protocols-base *init-OR* *response-OR* *check-OR* *Rel-OR* *S-OR*  
 $Ass-OR$  *challenge-space* *valid-pub-OR*  
 $\langle proof \rangle$

**end**

**locale**  $\Sigma$ -OR-proofs =  $\Sigma$ -OR-base + boolean-algebra L +  
**assumes**  $G$ -Rel-OR:  $((x0, x1), w) \in \text{set-spmf } G \implies ((x0, x1), w) \in \text{Rel-OR}$   
**and**  $\text{lossless-response-OR}$ :  $\text{lossless-spmf } (\text{response-OR } R \ W \ s)$   
**begin**

**lemma** HVZK1:  
**assumes**  $(x1, w1) \in \text{Rel1}$   
**shows**  $\forall c \in \text{challenge-space. } \Sigma\text{-OR.R } (x0, x1) \ (\text{Inr } w1) \ c = \Sigma\text{-OR.S } (x0, x1) \ c$   
**including**  $\text{monad-normalisation}$   
 $\langle \text{proof} \rangle$

**lemma** HVZK0:  
**assumes**  $(x0, w0) \in \text{Rel0}$   
**shows**  $\forall c \in \text{challenge-space. } \Sigma\text{-OR.R } (x0, x1) \ (\text{Inl } w0) \ c = \Sigma\text{-OR.S } (x0, x1) \ c$   
 $\langle \text{proof} \rangle$

**lemma** HVZK:  
**shows**  $\Sigma\text{-OR.HVZK}$   
 $\langle \text{proof} \rangle$

**lemma** **assumes**  $(x0, x1) \in \text{Domain Rel-OR}$   
**shows**  $(\exists w0. (x0, w0) \in \text{Rel0}) \vee (\exists w1. (x1, w1) \in \text{Rel1})$   
 $\langle \text{proof} \rangle$

**lemma**  $ss$ :  
**assumes**  $\text{valid-pub-OR}$ :  $(x0, x1) \in \text{valid-pub-OR}$   
**and**  $\text{check}$ :  $\text{check-OR } (x0, x1) \ (a0, a1) \ s \ ((e0, z0), (e1, z1))$   
**and**  $\text{check}'$ :  $\text{check-OR } (x0, x1) \ (a0, a1) \ s' \ ((e0', z0'), (e1', z1'))$   
**and**  $s \neq s'$   
**and**  $\text{challenge-space}$ :  $s \in \text{challenge-space } s' \in \text{challenge-space}$   
**shows**  $\text{lossless-spmf } (\mathcal{A}\text{ss-OR } (x0, x1) \ ((a0, a1), s, (e0, z0), e1, z1) \ ((a0, a1), s', (e0', z0'), e1', z1')) \wedge$   
 $(\forall w' \in \text{set-spmf } (\mathcal{A}\text{ss-OR } (x0, x1) \ ((a0, a1), s, (e0, z0), e1, z1) \ ((a0, a1), s', (e0', z0'), e1', z1'))). ((x0, x1), w') \in \text{Rel-OR})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{special-soundness}$ :  
**shows**  $\Sigma\text{-OR.special-soundness}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{correct0}$ :  
**assumes**  $e$ -in-carrier:  $e \in \text{carrier } L$   
**and**  $(x0, w0) \in \text{Rel0}$   
**and**  $\text{valid-pub}$ :  $x1 \in \text{valid-pub1}$   
**shows**  $\Sigma\text{-OR.completeness-game } (x0, x1) \ (\text{Inl } w0) \ e = \text{return-spmf True}$   
**(is ?lhs = ?rhs)**  
 $\langle \text{proof} \rangle$

**lemma** *correct1*:  
**assumes** *rel1*:  $(x1, w1) \in Rel1$   
**and** *valid-pub*:  $x0 \in valid-pub0$   
**and** *e-in-carrier*:  $e \in carrier L$   
**shows**  $\Sigma-OR.completeness-game (x0, x1) (Inr w1) e = return-spmf True$   
**(is ?lhs = ?rhs)**  
 $\langle proof \rangle$

**lemma** *completeness'*:  
**assumes** *Rel-OR-asm*:  $((x0, x1), w) \in Rel-OR$   
**shows**  $\forall e \in carrier L. spmf (\Sigma-OR.completeness-game (x0, x1) w e) True = 1$   
 $\langle proof \rangle$

**lemma** *completeness*: **shows**  $\Sigma-OR.completeness$   
 $\langle proof \rangle$

**lemma**  *$\Sigma$ -protocol*: **shows**  $\Sigma-OR.\Sigma-protocol$   
 $\langle proof \rangle$

**sublocale** *OR- $\Sigma$ -commit*:  $\Sigma-protocols-to-commitments init-OR response-OR check-OR$   
 $Rel-OR S-OR Ass-OR challenge-space valid-pub-OR G$   
 $\langle proof \rangle$

**lemma** *OR- $\Sigma$ -commit.abstract-com.correct*  
 $\langle proof \rangle$

**lemma** *OR- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa  $\mathcal{A}$*   
 $\langle proof \rangle$

**lemma** *bind-advantage-bound-dis-log*:  
**shows**  $OR-\Sigma-commit.abstract-com.bind-advantage \mathcal{A} \leq OR-\Sigma-commit.rel-advantage$   
 $(OR-\Sigma-commit.adversary \mathcal{A})$   
 $\langle proof \rangle$

**end**

**end**

## References

- [1] D. Aspinall and D. Butler. Multi-party computation. *Archive of Formal Proofs*, 2019, 2019.
- [2] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *IACR Cryptology ePrint Archive*, 2017:753, 2017.

- [3] C. Blundo, B. Masucci, D. R. Stinson, and R. Wei. Constructions and bounds for unconditionally secure non-interactive commitment schemes. *Des. Codes Cryptogr.*, 26(1-3):97–110, 2002.
- [4] D. Butler, D. Aspinall, and A. Gascón. How to simulate it in Isabelle: Towards formal proof for secure multi-party computation. In *ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2017.
- [5] D. Butler, D. Aspinall, and A. Gascón. On the formalisation of  $\Sigma$ -protocols and commitment schemes. In *POST*, volume 11426 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2019.
- [6] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.
- [7] I. Damgård. On  $\Sigma$ -protocols. *Lecture Notes, University of Aarhus, Department for Computer Science.*, 2002.
- [8] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.
- [9] R. Cramer. Modular design of secure, yet practical cryptographic protocols. *PhD thesis PhD Thesis, University of Amsterdam*, 1996.
- [10] R. Rivest. Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer. *Unpublished manuscript*, 1999.
- [11] C. Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.
- [12] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332, 2004.