

# $\Sigma$ -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 . . . . .           | 21        |
| 1.3      | Rivest Commitment Scheme . . . . .             | 28        |
| <b>2</b> | <b><math>\Sigma</math>-Protocols</b>           | <b>36</b> |
| 2.1      | Defining $\Sigma$ -protocols . . . . .         | 36        |
| 2.2      | Commitments from $\Sigma$ -protocols . . . . . | 39        |
| 2.3      | Schnorr $\Sigma$ -protocol . . . . .           | 44        |
| 2.4      | Chaum-Pedersen $\Sigma$ -protocol . . . . .    | 53        |
| 2.5      | Okamoto $\Sigma$ -protocol . . . . .           | 65        |
| 2.6      | $\Sigma$ -AND statements . . . . .             | 84        |
| 2.7      | $\Sigma$ -OR statements . . . . .              | 89        |

# 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*  $\mathcal{A} \longleftrightarrow$

$((\forall pk. \text{lossless-spmf } (fst \mathcal{A} pk)) \wedge$

$(\forall \text{commit } \sigma. \text{lossless-spmf } (snd \mathcal{A} \text{commit } \sigma)))$

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**  $\llbracket \text{lossless-spmf } \text{key-gen}; \text{lossless-spmf } \text{TI};$

$\bigwedge pk \ m. \text{valid-msg } m \implies \text{lossless-spmf } (\text{commit } pk \ m) \rrbracket$

$\implies \text{valid-msg } m \implies \text{lossless-spmf } (\text{correct-game } m)$

**by** (*simp* *add*: *lossless-def* *correct-game-def* *split-def* *Let-def*)

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

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*:

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

$\bigwedge pk \ \text{plain}. \text{valid-msg } \text{plain} \implies \text{lossless-spmf } (\text{commit } pk \ \text{plain}) \rrbracket$

$\implies \text{lossless-spmf } (\text{hiding-game-ind-cpa } \mathcal{A})$

**by** (*auto* *simp* *add*: *lossless-def* *hiding-game-ind-cpa-def* *split-def* *Let-def*)

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*  $\mathcal{A} \equiv |spmf (hiding-game-ind-cpa \mathcal{A}) True - 1/2|$

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

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*  $\mathcal{A} = TRY do \{$   
 $(ck, vk) \leftarrow key-gen;$   
 $(c, m, d, m', d') \leftarrow \mathcal{A} ck;$   
 $- :: unit \leftarrow assert-spmf (m \neq m' \wedge valid-msg m \wedge valid-msg m');$   
 $let b = verify vk m c d;$   
 $let b' = verify vk m' c d';$   
 $return-spmf (b \wedge b')\} ELSE return-spmf False$

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

**proof** –

**have** *?lhs =*  $TRY do \{$   
 $(ck, vk) \leftarrow key-gen;$   
 $TRY do \{$   
 $(c, m, d, m', d') \leftarrow \mathcal{A} ck;$   
 $TRY do \{$   
 $- :: unit \leftarrow assert-spmf (m \neq m' \wedge valid-msg m \wedge valid-msg m');$   
 $TRY return-spmf (verify vk m c d \wedge verify vk m' c d') ELSE return-spmf$   
 $False$   
 $\} ELSE return-spmf False$

```

    } ELSE return-spmf False
  } ELSE return-spmf False
  unfolding split-def bind-game-def
  by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
  also have ... = TRY do {
    (ck, vk) ← key-gen;
    TRY do {
      (c, m, d, m', d') ← A ck;
      TRY do {
        - :: unit ← assert-spmf (m ≠ m' ∧ valid-msg m ∧ valid-msg m');
        TRY do {
          - :: unit ← assert-spmf (verify vk m c d ∧ verify vk m' c d');
          return-spmf True
        } ELSE return-spmf False
      } ELSE return-spmf False
    } ELSE return-spmf False
  } ELSE return-spmf False
  by(auto simp add: try-bind-assert-spmf try-spmf-return-spmf1 intro!: try-spmf-cong
bind-spmf-cong)
  also have ... = ?rhs
  unfolding split-def Let-def
  by(fold try-bind-spmf-lossless2[OF lossless-return-spmf]) simp
  finally show ?thesis .
qed

```

**lemma** *lossless-binding-game*:  $\text{lossless-spmf } (\text{bind-game } \mathcal{A})$   
 by (*simp add: bind-game-def*)

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

end

end

**theory** *Cyclic-Group-Ext* **imports**

*CryptHOL.CryptHOL*

*HOL-Number-Theory.Cong*

**begin**

**context** *cyclic-group* **begin**

**lemma** *generator-pow-order*:  $\mathbf{g} [\ ] \text{ order } G = \mathbf{1}$

**proof**(*cases order G > 0*)

**case** *True*

**hence** *fin*: *finite* (*carrier G*) **by**(*simp add: order-gt-0-iff-finite*)

**then have** [*symmetric*]:  $(\lambda x. x \otimes \mathbf{g}) \text{ 'carrier } G = \text{carrier } G$

**by**(*rule endo-inj-surj*)(*auto simp add: inj-on-multc*)

**then have** *carrier G* =  $(\lambda n. \mathbf{g} [\ ] \text{ Suc } n) \text{ ' } \{..<\text{order } G\}$  **using** *fin*

**by**(*simp add: carrier-conv-generator image-image*)

**then obtain**  $n$  **where**  $n: \mathbf{1} = \mathbf{g} [\wedge] \text{Suc } n \ n < \text{order } G$  **by** *auto*  
**have**  $n = \text{order } G - 1$  **using**  $n$  *inj-onD[OF inj-on-generator, of 0 Suc n]* **by**  
*fastforce*  
**with** *True n* **show** *?thesis* **by** *auto*  
**qed** *simp*

**lemma** *generator-pow-mult-order*:  $\mathbf{g} [\wedge] (\text{order } G * \text{order } G) = \mathbf{1}$   
**using** *generator-pow-order*  
**by** (*metis generator-closed nat-pow-one nat-pow-pow*)

**lemma** *pow-generator-mod*:  $\mathbf{g} [\wedge] (k \text{ mod } \text{order } G) = \mathbf{g} [\wedge] k$   
**proof**(*cases order G > 0*)  
**case** *True*  
**obtain**  $n$  **where**  $n: k = n * \text{order } G + k \text{ mod } \text{order } G$  **by** (*metis div-mult-mod-eq*)  
**have**  $\mathbf{g} [\wedge] k = (\mathbf{g} [\wedge] \text{order } G) [\wedge] n \otimes \mathbf{g} [\wedge] (k \text{ mod } \text{order } G)$   
**by**(*subst n*)(*simp add: nat-pow-mult nat-pow-pow mult-ac*)  
**then show** *?thesis* **by**(*simp add: generator-pow-order*)  
**qed** *simp*

**lemma** *pow-carrier-mod*:  
**assumes**  $g \in \text{carrier } G$   
**shows**  $g [\wedge] (k \text{ mod } \text{order } G) = g [\wedge] k$   
**using** *assms pow-generator-mod*  
**by** (*metis generatorE generator-closed mod-mult-right-eq nat-pow-pow*)

**lemma** *pow-generator-mod-int*:  $\mathbf{g} [\wedge] ((k::\text{int}) \text{ mod } \text{order } G) = \mathbf{g} [\wedge] k$   
**proof**(*cases order G > 0*)  
**case** *True*  
**obtain**  $n :: \text{int}$  **where**  $n: k = n * \text{order } G + k \text{ mod } \text{order } G$   
**by** (*metis div-mult-mod-eq*)  
**have**  $\mathbf{g} [\wedge] k = (\mathbf{g} [\wedge] \text{order } G) [\wedge] n \otimes \mathbf{g} [\wedge] (k \text{ mod } \text{order } G)$   
**apply**(*subst n*)**apply**(*simp add: int-pow-mult int-pow-pow mult-ac*)  
**by** (*metis generator-closed int-pow-int int-pow-pow mult.commute*)  
**then show** *?thesis* **by**(*simp add: generator-pow-order*)  
**qed** *simp*

**lemma** *pow-generator-eq-iff-cong*:  
*finite (carrier G)  $\implies \mathbf{g} [\wedge] x = \mathbf{g} [\wedge] y \iff [x = y] (\text{mod } \text{order } G)$*   
**apply**(*subst (1 2) pow-generator-mod[symmetric]*)  
**by**(*auto simp add: cong-def order-gt-0-iff-finite intro: inj-onD[OF inj-on-generator]*)

**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** –  
**obtain**  $x :: \text{nat}$  **where**  $x: h = \mathbf{g} [\wedge] x$   
**using** *assms generatorE* **by** *blast*  
**hence** *?lhs =  $\mathbf{g} [\wedge] (e * (1 + x))$*

by (*simp add: nat-pow-mult mult.commute nat-pow-pow*)  
 also have ... = (g [⌈] (1 + x)) [⌈] e  
 by (*metis generator-closed mult.commute nat-pow-pow*)  
 ultimately show ?thesis  
 by (*metis x One-nat-def generator-closed l-one monoid.nat-pow-Suc monoid-axioms nat-pow-0 nat-pow-mult*)  
 qed

lemma *neg-power-inverse*:

assumes  $g \in \text{carrier } G$   
 and  $x < \text{order } G$   
 shows  $g \lceil (\text{order } G - (x :: \text{nat})) = \text{inv } (g \lceil x)$   
 proof -  
 have  $\text{inv } (g \lceil x) = g \lceil (- \text{int } x)$   
 by (*simp add: int-pow-int int-pow-neg assms*)  
 moreover have  $g \lceil (\text{order } G - (x :: \text{nat})) = g \lceil (- \text{int } x)$   
 proof -  
 have  $g \lceil ((\text{order } G - (x :: \text{nat})) \bmod (\text{order } G)) = g \lceil ((- \text{int } x) \bmod (\text{order } G))$   
 proof -  
 have  $(\text{order } G - (x :: \text{nat})) \bmod (\text{order } G) = (- \text{int } x) \bmod (\text{order } G)$   
 using *assms(2) zmod-zminus1-eq-if* by *auto*  
 thus ?thesis  
 by (*metis int-pow-int*)  
 qed  
 thus ?thesis  
 proof -  
 have  $f1: \forall a. a \lceil \text{int } 0 = \mathbf{1}$   
 by *simp*  
 have  $f2: \forall n \text{ na}. ((\text{na} :: \text{nat}) + n) \bmod \text{na} = n \bmod \text{na}$   
 by *simp*  
 have  $f3: \forall a \text{ aa}. \text{aa} \otimes a \lceil \text{int } 0 = \text{aa} \vee \text{aa} \notin \text{carrier } G$   
 by *force*  
 have  $f4: \forall i a \text{ aa}. a \lceil \text{int } 0 \otimes \text{aa} \lceil i = \text{aa} \lceil (\text{int } 0 + i) \vee \text{aa} \notin \text{carrier } G$   
 by *force*  
 have  $\forall n a. a \lceil \text{int } (n * 0) = a \lceil (\text{int } 0 + \text{int } 0) \vee a \notin \text{carrier } G$   
 by *simp*  
 then have  $f5: \forall a \text{ aa}. \text{aa} \lceil \text{int } (\text{order } G) = a \lceil \text{int } 0 \vee \text{aa} \notin \text{carrier } G$   
 using  $f4$   $f3$   $f2$   $f1$  by (*metis int-pow-closed int-pow-int mod-mult-self2 pow-carrier-mod*)  
 have  $\forall n \text{ na}. \text{int } (n - \text{na}) = - \text{int } \text{na} + \text{int } n \vee \neg \text{na} \leq n$   
 by *auto*  
 then show ?thesis  
 using  $f5$   $f3$  by (*metis assms(1) assms(2) int-pow-closed int-pow-int int-pow-mult less-imp-le-nat*)  
 qed  
 qed  
 ultimately show ?thesis by *simp*

qed

**lemma** *int-nat-pow*: **assumes**  $a \geq 0$  **shows**  $(\mathbf{g} \ [\uparrow] \ (\text{int} \ (a \ :: \ \text{nat}))) \ [\uparrow] \ (b \ :: \ \text{int}) = \mathbf{g} \ [\uparrow] \ (a * b)$   
**using** *assms*  
**proof** (*cases*  $a > 0$ )  
  **case** *True*  
    **show** *?thesis*  
    **using** *int-pow-pow* **by** *blast*  
  **next case** *False*  
    **have**  $(\mathbf{g} \ [\uparrow] \ (\text{int} \ (a \ :: \ \text{nat}))) \ [\uparrow] \ (b \ :: \ \text{int}) = \mathbf{1}$  **using** *False* **by** *simp*  
    **also have**  $\mathbf{g} \ [\uparrow] \ (a * b) = \mathbf{1}$  **using** *False* **by** *simp*  
    **ultimately show** *?thesis* **by** *simp*  
qed

**lemma** *pow-gen-mod-mult*:  
  **shows**  $(\mathbf{g} \ [\uparrow] \ (a \ :: \ \text{nat}) \otimes \mathbf{g} \ [\uparrow] \ (b \ :: \ \text{nat})) \ [\uparrow] \ ((c \ :: \ \text{int}) * \text{int} \ (d \ :: \ \text{nat})) = (\mathbf{g} \ [\uparrow] \ a \otimes \mathbf{g} \ [\uparrow] \ b) \ [\uparrow] \ ((c * \text{int} \ d) \ \text{mod} \ (\text{order} \ G))$   
**proof** –  
  **have**  $(\mathbf{g} \ [\uparrow] \ (a \ :: \ \text{nat}) \otimes \mathbf{g} \ [\uparrow] \ (b \ :: \ \text{nat})) \in \text{carrier} \ G$  **by** *simp*  
  **then obtain**  $n \ :: \ \text{nat}$  **where**  $n: \mathbf{g} \ [\uparrow] \ n = (\mathbf{g} \ [\uparrow] \ (a \ :: \ \text{nat}) \otimes \mathbf{g} \ [\uparrow] \ (b \ :: \ \text{nat}))$   
    **by** (*simp add: monoid.nat-pow-mult*)  
  **also obtain**  $r$  **where**  $r: r = c * \text{int} \ d$  **by** *simp*  
  **have**  $1: (\mathbf{g} \ [\uparrow] \ (a \ :: \ \text{nat}) \otimes \mathbf{g} \ [\uparrow] \ (b \ :: \ \text{nat})) \ [\uparrow] \ ((c \ :: \ \text{int}) * \text{int} \ (d \ :: \ \text{nat})) = (\mathbf{g} \ [\uparrow] \ n) \ [\uparrow] \ r$  **using**  $n \ r$  **by** *simp*  
  **also have**  $2: \dots = (\mathbf{g} \ [\uparrow] \ n) \ [\uparrow] \ (r \ \text{mod} \ (\text{order} \ G))$  **using** *pow-generator-mod-int* *pow-generator-mod*  
  **by** (*metis int-nat-pow int-pow-int mod-mult-right-eq zero-le*)  
  **also have**  $3: \dots = (\mathbf{g} \ [\uparrow] \ a \otimes \mathbf{g} \ [\uparrow] \ b) \ [\uparrow] \ ((c * \text{int} \ d) \ \text{mod} \ (\text{order} \ G))$  **using**  $r \ n$  **by** *simp*  
  **ultimately show** *?thesis* **using**  $1 \ 2 \ 3$  **by** *simp*  
qed

**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***)**  
**proof** –  
  **obtain**  $n \ :: \ \text{nat}$  **where**  $n: a = \mathbf{g} \ [\uparrow] \ n$  **using** *generatorE assms* **by** *auto*  
  **also obtain**  $k \ :: \ \text{nat}$  **where**  $k: b = \mathbf{g} \ [\uparrow] \ k$  **using** *generatorE assms* **by** *auto*  
  **ultimately have** *?lhs*  $= \mathbf{g} \ [\uparrow] \ n \otimes \mathbf{g} \ [\uparrow] \ k$  **by** *simp*  
  **then have**  $\dots = \mathbf{g} \ [\uparrow] \ (n + k)$  **by** (*simp add: nat-pow-mult*)  
  **then have**  $\dots = \mathbf{g} \ [\uparrow] \ (k + n)$  **by** (*simp add: add.commute*)  
  **then show** *?thesis* **by** (*simp add: nat-pow-mult n k*)  
qed

**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***)**



**proof** –

**obtain**  $n :: nat$  **where**  $n: a = g \ [^] \ n$  **using** *generatorE assms* **by** *auto*  
**obtain**  $k :: nat$  **where**  $k: b = g \ [^] \ k$  **using** *generatorE assms* **by** *auto*  
**obtain**  $j :: nat$  **where**  $j: c = g \ [^] \ j$  **using** *generatorE assms* **by** *auto*  
**have**  $?lhs = (g \ [^] \ n \otimes g \ [^] \ k) \otimes g \ [^] \ j$  **using**  $n \ k \ j$  **by** *simp*  
**then have**  $\dots = g \ [^] \ (n + (k + j))$  **by**(*simp add: nat-pow-mult add.assoc*)  
**then show**  $?thesis$  **by**(*simp add: nat-pow-mult n k j*)

**qed**

**lemma** *l-cancel-inv*:

**assumes**  $h \in carrier \ G$   
**shows**  $(g \ [^] \ (a :: nat) \otimes inv \ (g \ [^] \ a)) \otimes h = h$   
(**is**  $?lhs = ?rhs$ )

**proof** –

**have**  $?lhs = (g \ [^] \ int \ a \otimes inv \ (g \ [^] \ int \ a)) \otimes h$  **by** *simp*  
**then have**  $\dots = (g \ [^] \ int \ a \otimes (g \ [^] \ (- \ a))) \otimes h$  **using** *int-pow-neg[symmetric]*  
**by** *simp*  
**then have**  $\dots = g \ [^] \ (int \ a - a) \otimes h$  **by**(*simp add: int-pow-mult*)  
**then have**  $\dots = g \ [^] \ ((0 :: int)) \otimes h$  **by** *simp*  
**then show**  $?thesis$  **by** (*simp add: assms*)

**qed**

**lemma** *inverse-split*:

**assumes**  $a \in carrier \ G$  **and**  $b \in carrier \ G$   
**shows**  $inv \ (a \otimes b) = inv \ a \otimes inv \ b$   
**by** (*simp add: assms comm-group.inv-mult cyclic-group-commute group-comm-groupI*)

**lemma** *inverse-pow-pow*:

**assumes**  $a \in carrier \ G$   
**shows**  $inv \ (a \ [^] \ (r :: nat)) = (inv \ a) \ [^] \ r$

**proof** –

**have**  $a \ [^] \ r \in carrier \ G$   
**using** *assms* **by** *blast*  
**then show**  $?thesis$   
**by** (*simp add: assms nat-pow-inv*)

**qed**

**lemma** *l-neq-1-exp-neq-0*:

**assumes**  $l \in carrier \ G$   
**and**  $l \neq 1$   
**and**  $l = g \ [^] \ (t :: nat)$   
**shows**  $t \neq 0$

**proof**(*rule ccontr*)

**assume**  $\neg \ (t \neq 0)$   
**hence**  $t = 0$  **by** *simp*  
**hence**  $g \ [^] \ t = 1$  **by** *simp*  
**then show** *False* **using** *assms* **by** *simp*

**qed**

```

lemma order-gt-1-gen-not-1:
  assumes order  $G > 1$ 
  shows  $\mathbf{g} \neq \mathbf{1}$ 
proof(rule ccontr)
  assume  $\neg \mathbf{g} \neq \mathbf{1}$ 
  hence  $\mathbf{g} = \mathbf{1}$  by simp
  hence g-pow-eq-1:  $\mathbf{g} [\wedge] n = \mathbf{1}$  for  $n :: \text{nat}$  by simp
  hence range  $(\lambda n :: \text{nat}. \mathbf{g} [\wedge] n) = \{\mathbf{1}\}$  by auto
  hence carrier  $G \subseteq \{\mathbf{1}\}$  using generator by auto
  hence order  $G < 1$ 
    by (metis inj-onD inj-on-generator lessThan-iff g-pow-eq-1 assms less-one
neq0-conv)
  with assms show False by simp
qed

lemma power-swap:  $((\mathbf{g} [\wedge] (\alpha 0 :: \text{nat})) [\wedge] (r :: \text{nat})) = ((\mathbf{g} [\wedge] r) [\wedge] \alpha 0)$ 
(is ?lhs = ?rhs)
proof -
  have ?lhs =  $\mathbf{g} [\wedge] (\alpha 0 * r)$  using nat-pow-pow mult.commute by auto
  hence ... =  $\mathbf{g} [\wedge] (r * \alpha 0)$  by(metis mult.commute)
  thus ?thesis using nat-pow-pow by auto
qed

lemma gen-power-0:
  fixes  $r :: \text{nat}$ 
  assumes  $\mathbf{g} [\wedge] r = \mathbf{1}$ 
    and  $r < \text{order } G$ 
  shows  $r = 0$ 
  using assms inj-onD inj-on-generator by fastforce

lemma group-eq-pow-eq-mod:
  fixes  $a b :: \text{nat}$ 
  assumes  $\mathbf{g} [\wedge] a = \mathbf{g} [\wedge] b$ 
    and order  $G > 0$ 
  shows  $[a = b] \pmod{\text{order } G}$ 
proof(cases  $a > b$ )
  case True
  have  $\mathbf{g} [\wedge] a \otimes \text{inv } (\mathbf{g} [\wedge] b) = \mathbf{1}$ 
    using assms by simp
  hence  $\mathbf{g} [\wedge] (a - b) = \mathbf{1}$ 
    by (smt True add-Suc-right assms diff-add-inverse generator-closed group.l-cancel-one'
group-l-invI l-inv-ex less-imp-Suc-add nat-pow-closed nat-pow-mult)
  hence  $\mathbf{g} [\wedge] ((a - b) \pmod{\text{order } G}) = \mathbf{1}$  using pow-generator-mod by auto
  thus ?thesis using gen-power-0
    using assms(1) assms(2) order-gt-0-iff-finite pow-generator-eq-iff-cong by blast
  next
  case False
  have  $\mathbf{g} [\wedge] a \otimes \text{inv } (\mathbf{g} [\wedge] b) = \mathbf{1}$ 
    using assms by simp

```

```

hence  $\mathbf{g} [\uparrow] (b - a) = \mathbf{1}$ 
by (metis (no-types, lifting) False Group.group.axioms(1) Units-eq add-diff-inverse-nat
assms(1) generator-closed group-l-invI l-inv-ex l-neq-1-exp-neq-0 monoid.Units-l-cancel
nat-pow-closed nat-pow-mult r-one)
hence  $\mathbf{g} [\uparrow] ((b - a) \bmod (\text{order } G)) = \mathbf{1}$  using pow-generator-mod by simp
thus ?thesis using gen-power-0
using assms(1) assms(2) order-gt-0-iff-finite pow-generator-eq-iff-cong by blast
qed

```

**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]: 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*  $\mathcal{A} = \text{TRY do}$  {

$x \leftarrow \text{sample-uniform (order } \mathcal{G})$ ;

$\text{let } h = \mathbf{g} [\uparrow] x$ ;

$x' \leftarrow \mathcal{A} h$ ;

$\text{return-spmf } ([x = x'] \bmod \text{order } \mathcal{G})$ } *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*  $\mathcal{A}$ )

**by**(*auto simp add: dis-log-def*)

**end**

**locale** *dis-log-alt* =

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

**and**  $x :: \text{nat}$

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

**begin**

**sublocale** *dis-log*: *dis-log*  $\mathcal{G}$   
**unfolding** *dis-log-def* **by** *simp*

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

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

**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*  $\mathcal{A} = \text{TRY do}$  {  
   $w \leftarrow \text{sample-uniform (order } \mathcal{G})$ ;  
   $\text{let } (h, w) = ((\mathbf{g} [\hat{\cdot}] w, g' [\hat{\cdot}] w), w)$ ;  
   $w' \leftarrow \mathcal{A} h$ ;  
   $\text{return-spmf } ([w = w'] \text{ (mod (order } \mathcal{G}))}] \text{ ELSE return-spmf False}$

**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*  $\mathcal{A} g = \text{do}$  {  
   $\mathcal{A} (g, g [\hat{\cdot}] x)$ }

**end**

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

**lemma** *dis-log-adv3*:  
  **shows** *advantage3*  $\mathcal{A} = \text{dis-log.} \text{advantage (adversary3 } \mathcal{A})$   
  **unfolding** *dis-log-alt.advantage3-def*  
  **by**(*simp add: advantage3-def dis-log.advantage-def adversary3-def dis-log.dis-log-def dis-log3-def Let-def g'-def power-swap*)

**lemma** *dis-log-adv2*:

**shows** *advantage2*  $\mathcal{A} = \text{dis-log.}\text{advantage} (\text{adversary2 } \mathcal{A})$   
**unfolding** *dis-log-alt.advantage2-def*  
**by**(*simp add: advantage2-def dis-log2-def dis-log.advantage-def dis-log.dis-log-def*  
*adversary2-def split-def*)

**end**

**end**

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

*HOL-Number-Theory.Cong*

*HOL-Number-Theory.Residues*

**begin**

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

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

**shows**  $[x * \text{inverse } x\ q = 1] \pmod{q}$

**proof** –

**have** 2:  $\text{fst } (\text{bezw } x\ q) * x + \text{snd } (\text{bezw } x\ q) * \text{int } q = 1$

**using** *bezw-aux assms int-minus*

**by** (*metis Num.of-nat-simps(2)*)

**hence** 3:  $(\text{fst } (\text{bezw } x\ q) * x + \text{snd } (\text{bezw } x\ q) * \text{int } q) \pmod{q} = 1 \pmod{q}$

**by** (*metis assms bezw-aux of-nat-mod*)

**hence** 4:  $(\text{fst } (\text{bezw } x\ q) * x) \pmod{q} = 1 \pmod{q}$

**by** *simp*

**hence** 5:  $[(\text{fst } (\text{bezw } x\ q)) * x = 1] \pmod{q}$

**using** 2 3 *cong-def* **by** *force*

**then show** *?thesis* **by**(*simp add: mult.commute*)

**qed**

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

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

**and** *prime*  $y$

**and**  $x > 2$

**and**  $y > 2$

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

**by** (*metis assms One-nat-def Suc-diff-1 nat-neq-iff numeral-2-eq-2 prime-gt-0-nat*  
*prime-product*)

**lemma** *ex-inverse:*

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

**and** *prime*  $P$

**and** *prime*  $Q$

**and**  $P \neq Q$

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

**proof** –

**have** *coprime*  $e (P-1)$

**using** *assms(1)* **by** *simp*

**then obtain**  $d$  **where**  $d: [e*d = 1] \pmod{(P-1)}$

**using** *cong-solve-coprime-nat* **by** *auto*  
**then show** *?thesis* **by** (*metis cong-0-1-nat cong-1 mult-0-right zero-neq-one*)  
**qed**

**lemma** *ex-k1-k2*:  
**assumes** *coprime: coprime (e :: nat) ((P-1)\*(Q-1))*  
**and**  $[e*d = 1] \pmod{P-1}$   
**shows**  $\exists k1\ k2. e*d + k1*(P-1) = 1 + k2*(P-1)$   
**by** (*metis assms(2) cong-iff-lin-nat*)  
**lemma**  $a > b \implies \text{int } a - \text{int } b = \text{int } (a - b)$   
**by** *simp*

**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**  $[e*d = 1] \pmod{P-1}$   
**shows**  $\exists k. e*d = 1 + k*(P-1)$

**proof** –  
**have**  $e > 0$   
**using** *assms(1) assms(2) prime-gt-0-nat* **by** *fastforce*  
**then have**  $e*d \geq 1$  **using** *assms* **by** *simp*  
**then obtain**  $k$  **where**  $k: e*d = 1 + k*(P-1)$   
**using** *assms(6) cong-to-1'-nat* **by** *auto*  
**then show** *?thesis*  
**by** *simp*

**qed**

**lemma** *fermat-little-theorem*:  
**assumes** *prime (P :: nat)*  
**shows**  $[x^P = x] \pmod{P}$   
**proof**(*cases P dvd x*)  
**case** *True*  
**hence**  $x \pmod{P} = 0$  **by** *simp*  
**moreover have**  $x^P \pmod{P} = 0$   
**by** (*simp add: True assms prime-dvd-power-nat-iff prime-gt-0-nat*)  
**ultimately show** *?thesis*  
**by** (*simp add: cong-def*)

**next**

**case** *False*  
**hence**  $[x^{P-1} = 1] \pmod{P}$  **using** *fermat-theorem assms* **by** *blast*  
**then show** *?thesis*  
**by** (*metis Suc-diff-1 assms cong-scalar-left nat-mult-1-right not-gr-zero not-prime-0 power-Suc*)

**qed**

**lemma** *prime-field*:

```

assumes prime (q::nat)
  and a < q
  and a ≠ 0
shows coprime a q
by (meson assms coprime-commute dvd-imp-le linorder-not-le neq0-conv prime-imp-coprime)

end
theory Uniform-Sampling imports
  CryptHOL.CryptHOL
  HOL-Number-Theory.Cong
begin

definition sample-uniform-units :: nat ⇒ nat spmf
  where sample-uniform-units q = spmf-of-set ( $\{..q\} - \{0\}$ )

lemma set-spmf-sample-uniform-units [simp]:
  set-spmf (sample-uniform-units q) =  $\{..q\} - \{0\}$ 
  by(simp add: sample-uniform-units-def)

lemma lossless-sample-uniform-units:
  assumes (p::nat) > 1
  shows lossless-spmf (sample-uniform-units p)
  unfolding sample-uniform-units-def
  using assms by auto

lemma weight-sample-uniform-units:
  assumes (p::nat) > 1
  shows weight-spmf (sample-uniform-units p) = 1
  using assms lossless-sample-uniform-units
  by (simp add: lossless-weight-spmfD)

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)
proof –
  have rhs: ?rhs = spmf-of-set ( $(\{..q\} - \{0\})$ )
  by(auto simp add: sample-uniform-units-def)
  also have map-spmf( $\lambda s. f s$ ) (spmf-of-set ( $\{..q\} - \{0\}$ )) = spmf-of-set ( $(\lambda s. f$ 
s) ‘ ( $\{..q\} - \{0\}$ ))
  by(simp add: inj-on)
  also have f ‘ ( $\{..q\} - \{0\}$ ) = ( $\{..q\} - \{0\}$ )
  apply(rule endo-inj-surj) by(simp, simp add: sur, simp add: inj-on)
  ultimately show ?thesis using rhs by simp
qed

```

**lemma** *one-time-pad*:  
**assumes** *inj-on*:  $\text{inj-on } f \{..<q\}$   
**and** *sur*:  $f \text{ ' } \{..<q\} = \{..<q\}$   
**shows**  $\text{map-spmf } f \text{ (sample-uniform } q) = \text{(sample-uniform } q)$   
**(is** *?lhs = ?rhs*)  
**proof** –  
**have** *rhs*:  $?rhs = \text{spmof-of-set } \{..<q\}$   
**by**(*auto simp add: sample-uniform-def*)  
**also have**  $\text{map-spmf}(\lambda s. f s) \text{ (spmof-of-set } \{..<q\}) = \text{spmof-of-set } ((\lambda s. f s) \text{ ' } \{..<q\})$   
**by**(*simp add: inj-on*)  
**also have**  $f \text{ ' } \{..<q\} = \{..<q\}$   
**apply**(*rule endo-inj-surj*) **by**(*simp, simp add: sur, simp add: inj-on*)  
**ultimately show** *?thesis* **using** *rhs* **by** *simp*  
**qed**

**lemma** *plus-inj-eq*:  
**assumes** *x*:  $x < q$   
**and** *x'*:  $x' < q$   
**and** *map*:  $((y :: \text{nat}) + x) \bmod q = (y + x') \bmod q$   
**shows**  $x = x'$   
**proof** –  
**have**  $((y :: \text{nat}) + x) \bmod q = (y + x') \bmod q \implies x \bmod q = x' \bmod q$   
**proof** –  
**have**  $((y :: \text{nat}) + x) \bmod q = (y + x') \bmod q \implies [((y :: \text{nat}) + x) = (y + x')] \text{ (mod } q)$   
**by**(*simp add: cong-def*)  
**moreover have**  $[((y :: \text{nat}) + x) = (y + x')] \text{ (mod } q) \implies [x = x'] \text{ (mod } q)$   
**by** (*simp add: cong-add-lcancel-nat*)  
**moreover have**  $[x = x'] \text{ (mod } q) \implies x \bmod q = x' \bmod q$   
**by**(*simp add: cong-def*)  
**ultimately show** *?thesis* **by**(*simp add: map*)  
**qed**  
**moreover have**  $x \bmod q = x' \bmod q \implies x = x'$   
**by**(*simp add: x x'*)  
**ultimately show** *?thesis* **by**(*simp add: map*)  
**qed**

**lemma** *inj-uni-samp-plus*:  $\text{inj-on } (\lambda(b :: \text{nat}). (y + b) \bmod q) \{..<q\}$   
**by**(*simp add: inj-on-def*)(*auto simp only: plus-inj-eq*)

**lemma** *surj-uni-samp-plus*:  
**assumes** *inj*:  $\text{inj-on } (\lambda(b :: \text{nat}). (y + b) \bmod q) \{..<q\}$   
**shows**  $(\lambda(b :: \text{nat}). (y + b) \bmod q) \text{ ' } \{..<q\} = \{..<q\}$   
**apply**(*rule endo-inj-surj*) **using** *inj* **by** *auto*

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



**shows**  $\text{map-spmf } (\lambda b. (y + b) \text{ mod } q) (\text{sample-uniform } q) = \text{sample-uniform } q$   
**using** *inj-uni-samp-plus surj-uni-samp-plus one-time-pad* **by** *simp*

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

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

**proof** –

**have**  $x*y \text{ mod } q = x*y' \text{ mod } q \implies y \text{ mod } q = y' \text{ mod } q$

**proof** –

**have**  $x*y \text{ mod } q = x*y' \text{ mod } q \implies [x*y = x*y'] (\text{mod } q)$

**by**(*simp add: cong-def*)

**moreover have**  $[x*y = x*y'] (\text{mod } q) = [y = y'] (\text{mod } q)$

**by**(*simp add: cong-mult-lcancel-nat coprime*)

**moreover have**  $[y = y'] (\text{mod } q) \implies y \text{ mod } q = y' \text{ mod } q$

**by**(*simp add: cong-def*)

**ultimately show** *?thesis* **by**(*simp add: map*)

**qed**

**moreover have**  $y \text{ mod } q = y' \text{ mod } q \implies y = y'$

**by**(*simp add: y y'*)

**ultimately show** *?thesis* **by**(*simp add: map*)

**qed**

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

**assumes** *coprime*:  $\text{coprime } x (q::\text{nat})$   
**shows** *inj-on*  $(\lambda b. x*b \text{ mod } q) \{..<q\}$   
**apply**(*auto simp add: inj-on-def*)  
**using** *coprime* **by**(*simp only: mult-inj-eq*)

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

**assumes** *coprime*:  $\text{coprime } x (q::\text{nat})$   
**and** *inj*: *inj-on*  $(\lambda b. x*b \text{ mod } q) \{..<q\}$   
**shows**  $(\lambda b. x*b \text{ mod } q) ' \{..<q\} = \{..<q\}$   
**apply**(*rule endo-inj-surj*) **using** *coprime inj* **by** *auto*

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

**assumes** *coprime*:  $\text{coprime } x q$   
**shows**  $\text{map-spmf } (\lambda b. x*b \text{ mod } q) (\text{sample-uniform } q) = \text{sample-uniform } q$   
**using** *inj-on-mult surj-on-mult one-time-pad coprime* **by** *simp*

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

**assumes** *coprime*:  $\text{coprime } x (q::\text{nat})$   
**shows** *inj-on*  $(\lambda b. x*b \text{ mod } q) (\{..<q\} - \{0\})$   
**apply**(*auto simp add: inj-on-def*)

using *coprime* by(*simp only: mult-inj-eq*)

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

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

**and** *inj: inj-on*  $(\lambda b. x*b \bmod q)$   $(\{..<q\} - \{0\})$

**shows**  $(\lambda b. x*b \bmod q)$  '  $(\{..<q\} - \{0\}) = (\{..<q\} - \{0\})$

**proof**(*rule endo-inj-surj*)

**show** *finite*  $(\{..<q\} - \{0\})$  **by** *auto*

**show**  $(\lambda b. x * b \bmod q)$  '  $(\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\}$

**proof**–

**obtain**  $nn :: nat \text{ set} \Rightarrow (nat \Rightarrow nat) \Rightarrow nat \text{ set} \Rightarrow nat$  **where**

$\forall x0 \ x1 \ x2. (\exists v3. v3 \in x2 \wedge x1 \ v3 \notin x0) = (nn \ x0 \ x1 \ x2 \in x2 \wedge x1 \ (nn \ x0 \ x1 \ x2) \notin x0)$

**by** *moura*

**hence**  $1: \forall N \ f \ Na. nn \ Na \ f \ N \in N \wedge f \ (nn \ Na \ f \ N) \notin Na \vee f \ ' \ N \subseteq Na$

**by** (*meson image-subsetI*)

**have**  $2: x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \notin \{..<q\} \vee x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in insert \ 0 \ \{..<q\}$

**by** *force*

**have**  $3: (x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in insert \ 0 \ \{..<q\} - \{0\}) = (x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in \{..<q\} - \{0\})$

**by** *simp*

**{ assume**  $x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q = x * 0 \bmod q$

**hence**  $(0 \leq q) = (0 = q) \vee (nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \notin \{..<q\} \vee nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in \{0\}) \vee nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \notin \{..<q\} - \{0\} \vee x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in \{..<q\} - \{0\}$

**by** (*metis antisym-conv1 insertCI lessThan-iff local.coprime mult-inj-eq*) }

**moreover**

**{ assume**  $0 \neq x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q$

**moreover**

**{ assume**  $x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in insert \ 0 \ \{..<q\} \wedge x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \notin \{0\}$

**hence**  $(\lambda n. x * n \bmod q)$  '  $(\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\}$

**using**  $3 \ 1$  **by** (*meson Diff-iff*) }

**ultimately have**  $(\lambda n. x * n \bmod q)$  '  $(\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\} \vee (0 \leq q) = (0 = q)$

**using**  $2$  **by** (*metis antisym-conv1 lessThan-iff mod-less-divisor singletonD*)

}

**ultimately have**  $(\lambda n. x * n \bmod q)$  '  $(\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\} \vee nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \notin \{..<q\} - \{0\} \vee x * nn \ (\{..<q\} - \{0\}) \ (\lambda n. x * n \bmod q) \ (\{..<q\} - \{0\}) \bmod q \in \{..<q\} - \{0\}$

**by** *force*

**thus**  $(\lambda n. x * n \bmod q)$  '  $(\{..<q\} - \{0\}) \subseteq \{..<q\} - \{0\}$

```

    using 1 by meson
  qed
  show inj-on ( $\lambda b. x * b \text{ mod } q$ ) ( $\{..<q\} - \{0\}$ )
    using inj by blast
  qed

lemma mult-one-time-pad':
  assumes coprime: coprime x q
  shows map-spmf ( $\lambda b. x*b \text{ mod } q$ ) (sample-uniform-units q) = sample-uniform-units
  q
  using inj-on-mult' surj-on-mult' one-time-pad' coprime by simp

lemma samp-uni-add-mult:
  assumes coprime: coprime x (q::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 -
    have  $(y + x * x') \text{ mod } q = (y + x * y') \text{ mod } q \implies x' \text{ mod } q = y' \text{ mod } q$ 
    proof -
      have  $(y + x * x') \text{ mod } q = (y + x * y') \text{ mod } q \implies [y + x*x' = y + x*y'] \text{ (mod } q)$ 
      using cong-def by blast
      moreover have  $[y + x*x' = y + x*y'] \text{ (mod } q) \implies [x' = y'] \text{ (mod } q)$ 
      by(simp add: cong-add-lcancel-nat)(simp add: coprime cong-mult-lcancel-nat)
      ultimately show ?thesis by(simp add: cong-def map)
    qed
    moreover have  $x' \text{ mod } q = y' \text{ mod } q \implies x' = y'$ 
    by(simp add: x' y')
    ultimately show ?thesis by(simp add: map)
  qed

lemma inj-on-add-mult:
  assumes coprime: coprime x (q::nat)
  shows inj-on ( $\lambda b. (y + x*b) \text{ mod } q$ )  $\{..<q\}$ 
  apply(auto simp add: inj-on-def)
  using coprime by(simp only: samp-uni-add-mult)

lemma surj-on-add-mult:
  assumes coprime: coprime x (q::nat)
  and inj: inj-on ( $\lambda b. (y + x*b) \text{ mod } q$ )  $\{..<q\}$ 
  shows  $(\lambda b. (y + x*b) \text{ mod } q) ' \{..<q\} = \{..<q\}$ 
  apply(rule endo-inj-surj) using coprime inj by auto

lemma add-mult-one-time-pad:
  assumes coprime: coprime x q

```

**shows**  $\text{map-spmf } (\lambda b. (y + x*b) \text{ mod } q) (\text{sample-uniform } q) = (\text{sample-uniform } q)$

**using** *inj-on-add-mult surj-on-add-mult one-time-pad coprime by simp*

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

**proof**(*unfold inj-on-def; auto*)

**fix**  $x :: \text{nat}$  **and**  $y' :: \text{nat}$

**assume**  $x: x < q$

**assume**  $y': y' < q$

**assume**  $\text{map}: (y + q - x) \text{ mod } q = (y + q - y') \text{ mod } q$

**have**  $\forall n \text{ na } p. \exists nb. \forall nc \text{ nd } pa. (\neg (nc :: \text{nat}) < nd \vee \neg pa (nc - nd) \vee pa \ 0) \wedge (\neg p (0 :: \text{nat}) \vee p (n - na) \vee na + nb = n)$

**by** (*metis (no-types) nat-diff-split*)

**hence**  $\neg y < y' - q \wedge \neg y < x - q$

**using**  $y' \ x$  **by** (*metis add.commute less-diff-conv not-add-less2*)

**hence**  $\exists n. (y' + n) \text{ mod } q = (n + x) \text{ mod } q$

**using**  $\text{map}$  **by** (*metis add.commute add-diff-inverse-nat less-diff-conv mod-add-left-eq*)

**thus**  $x = y'$

**by** (*metis plus-inj-eq x y' add.commute*)

**qed**

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

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

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

**apply**(*rule endo-inj-surj*) **using** *inj* **by** *auto*

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

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

**using** *inj-on-minus surj-on-minus one-time-pad by simp*

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

**proof**–

**have** *inj-on Not*  $\{True, False\}$

**by** *simp*

**moreover** **have** *Not* '  $\{True, False\} = \{True, False\}$

**by** *auto*

**ultimately** **show** *?thesis* **using** *one-time-pad*

**by** (*simp add: UNIV-bool*)

**qed**

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

(*is ?lhs = ?rhs*)

**proof**–

**have** *rhs*: *?rhs = spmf-of-set*  $\{True, False\}$

**by** (*simp add: UNIV-bool insert-commute*)

```

also have map-spmf( $\lambda b. y \oplus b$ ) (spmf-of-set {True, False}) = spmf-of-set(( $\lambda$ 
 $b. y \oplus b$ ) ‘ {True, False})
by (simp add: xor-def)
also have ( $\lambda b. xor\ y\ b$ ) ‘ {True, False} = {True, False}
using xor-def by auto
finally show ?thesis using rhs by (simp)
qed

```

**lemma** ped-inv-mapping:

```

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

```

**proof** –

```

have ineq:  $q * m - a * m > 0$ 
using assms gr0I by force
have ?lhs = map-spmf ( $\lambda d. (a * m + d) \bmod q$ ) (sample-uniform  $q$ )
using add.commute by metis
also have ... = sample-uniform  $q$ 
using samp-uni-plus-one-time-pad by simp
also have ... = map-spmf ( $\lambda d. ((q * m - a * m) + d) \bmod q$ ) (sample-uniform
 $q$ )
using ineq samp-uni-plus-one-time-pad by metis
ultimately show ?thesis
using add.commute ineq
by (simp add: Groups.add-ac(2))

```

**qed**

**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  $\mathcal{G} > 0$ 
  by (simp add: prime-gt-0-nat prime-order)

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  $\times$  'grp vk) spmf
where
  key-gen = do {
     $x :: nat \leftarrow$  sample-uniform (order  $\mathcal{G}$ );
    let  $h = \mathbf{g} [\wedge] x$ ;
    return-spmf (h, h)
  }

definition commit :: 'grp ck  $\Rightarrow$  plain  $\Rightarrow$  ('grp commit  $\times$  opening) spmf
where
  commit ck m = do {
     $d :: nat \leftarrow$  sample-uniform (order  $\mathcal{G}$ );
    let  $c = (\mathbf{g} [\wedge] d) \otimes (ck [\wedge] m)$ ;
    return-spmf (c,d)
  }

definition commit-inv :: 'grp ck  $\Rightarrow$  plain  $\Rightarrow$  ('grp commit  $\times$  opening) spmf
where
  commit-inv ck m = do {
     $d :: nat \leftarrow$  sample-uniform (order  $\mathcal{G}$ );
    let  $c = (\mathbf{g} [\wedge] d) \otimes (inv\ ck [\wedge] m)$ ;
    return-spmf (c,d)
  }

definition verify :: 'grp vk  $\Rightarrow$  plain  $\Rightarrow$  'grp commit  $\Rightarrow$  opening  $\Rightarrow$  bool
where
  verify v-key m c d = (c = ( $\mathbf{g} [\wedge] d \otimes v\text{-key} [\wedge] m$ ))

```

**definition** *valid-msg* :: *plain*  $\Rightarrow$  *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-msg* *m*  $\wedge$  *valid-msg* *m'*);

- :: *unit*  $\leftarrow$  *assert-spmf* (*c* =  $\mathbf{g} [\wedge] d \otimes h [\wedge] m \wedge c = \mathbf{g} [\wedge] d' \otimes h [\wedge] m'$ );

*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* .

**sublocale** *discrete-log*: *dis-log* -

**unfolding** *dis-log-def* **by** (*simp*)

**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  $\mathcal{G}$ )

**by** (*metis* *assms* *Groups.mult-ac*(2) *cong-scalar-right* *cong-sym-eq* *cong-trans* *more-arith-simps*(11) *more-arith-simps*(5))

**lemma** *dis-log-break*:

**fixes** *d* *d'* *m* *m'* :: *nat*

**assumes** *c*:  $\mathbf{g} [\wedge] d' \otimes (\mathbf{g} [\wedge] y) [\wedge] m' = \mathbf{g} [\wedge] d \otimes (\mathbf{g} [\wedge] y) [\wedge] m$

**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** -

**have** *mm'*:  $\neg [m = m']$  (mod order  $\mathcal{G}$ )

**using** *m m-ge-m'* *basic-trans-rules*(19) *cong-less-modulus-unique-nat* **by** *blast*

**hence** *gcd*: *int* (*gcd* ((*m* - *m'*) (order  $\mathcal{G}$ ))) = 1

**using** *assms*(3) *assms*(4) *prime-field* *prime-order* **by** *auto*

**have**  $\mathbf{g} [\wedge] (d + y * m) = \mathbf{g} [\wedge] (d' + y * m')$

**using** *c* **by** (*simp* *add*: *nat-pow-mult* *nat-pow-pow*)

**hence** [*d* + *y* \* *m* = *d'* + *y* \* *m'*] (mod order  $\mathcal{G}$ )

**by** (*simp* *add*: *pow-generator-eq-iff-cong* *finite-carrier*)

**hence** [*int* *d* + *int* *y* \* *int* *m* = *int* *d'* + *int* *y* \* *int* *m'*] (mod order  $\mathcal{G}$ )

**using** *cong-int-iff* **by** *force*

```

from cong-diff[OF this cong-refl, of int d + int y * int m']
have [int y * int (m - m') = int d' - int d] (mod order  $\mathcal{G}$ ) using m-ge-m'
  by(simp add: int-distrib of-nat-diff)
hence *: [int y * int (m - m') * (fst (bezw ((m - m') (order  $\mathcal{G}$ ))) = (int d' -
int d) * (fst (bezw ((m - m') (order  $\mathcal{G}$ ))))] (mod order  $\mathcal{G}$ )
  by (simp add: cong-scalar-right)
hence [int y * (int (m - m') * (fst (bezw ((m - m') (order  $\mathcal{G}$ )))) = (int d' -
int d) * (fst (bezw ((m - m') (order  $\mathcal{G}$ ))))] (mod order  $\mathcal{G}$ )
  by (simp add: more-arith-simps(11))
hence [int y * 1 = (int d' - int d) * (fst (bezw ((m - m') (order  $\mathcal{G}$ ))))] (mod
order  $\mathcal{G}$ )
  using inverse gcd
  by (smt Groups.mult-ac(2) Number-Theory-Aux.inverse Totient.of-nat-eq-1-iff
* cong-def int-ops(9) mod-mult-right-eq mod-one-cancel)
hence [int y = (int d' - int d) * (fst (bezw ((m - m') (order  $\mathcal{G}$ ))))] (mod order
 $\mathcal{G}$ ) by simp
hence y mod order  $\mathcal{G}$  = (int d' - int d) * (fst (bezw ((m - m') (order  $\mathcal{G}$ ))))
mod order  $\mathcal{G}$ 
  using cong-def zmod-int by auto
thus ?thesis using y-less-order by simp
qed

```

```

lemma dis-log-break':
  assumes y < order  $\mathcal{G}$ 
  and ¬ m' < m
  and m ≠ m'
  and m: m' < order  $\mathcal{G}$ 
  and g [∗] d ⊗ (g [∗] y) [∗] m = g [∗] d' ⊗ (g [∗] y) [∗] m'
  shows y = nat ((int d - int d') * fst (bezw ((m' - m)) (order  $\mathcal{G}$ ))) mod int
(order  $\mathcal{G}$ )
proof -
  have m' > m using assms
  using group-eq-pow-eq-mod nat-neq-iff order-gt-0 by blast
  thus ?thesis
  using dis-log-break[of d y m d' m'] assms cong-sym-eq assms by blast
qed

```

```

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

```

```

lemma correct:
  shows spmf (ped-commit.correct-game m) True = 1
  using finite-carrier order-gt-0-iff-finite
  apply(simp add: abstract-commitment.correct-game-def Let-def commit-def ver-
ify-def)
  by(simp add: key-gen-def Let-def bind-spmf-const cong: bind-spmf-cong-simp)

```

```

theorem abstract-correct:

```



**shows** *ped-commit.correct*  
**unfolding** *abstract-commitment.correct-def* **using** *correct* **by** *simp*

**lemma** *perfect-hiding*:

**shows** *spmf (ped-commit.hiding-game-ind-cpa A) True - 1/2 = 0*  
**including** *monad-normalisation*

**proof** –

**obtain** *A1 A2* **where** [*simp*]:  $\mathcal{A} = (\mathcal{A}1, \mathcal{A}2)$  **by**(*cases A*)

**note** [*simp*] = *Let-def split-def*

**have** *ped-commit.hiding-game-ind-cpa (A1, A2) = TRY do {*

*(ck, vk) ← key-gen;*

*((m0, m1), σ) ← A1 vk;*

*- :: unit ← assert-spmf (valid-msg m0 ∧ valid-msg m1);*

*b ← coin-spmf;*

*(c, d) ← commit ck (if b then m0 else m1);*

*b' ← A2 c σ;*

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

**by**(*simp add: abstract-commitment.hiding-game-ind-cpa-def*)

**also have** ... = *TRY do {*

*x :: nat ← sample-uniform (order G);*

*let h = g [∧] x;*

*((m0, m1), σ) ← A1 h;*

*- :: unit ← assert-spmf (valid-msg m0 ∧ valid-msg m1);*

*b ← coin-spmf;*

*d :: nat ← sample-uniform (order G);*

*let c = ((g [∧] d) ⊗ (h [∧] (if b then m0 else m1)));*

*b' ← A2 c σ;*

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

**by**(*simp add: commit-def key-gen-def*)

**also have** ... = *TRY do {*

*x :: nat ← sample-uniform (order G);*

*let h = (g [∧] x);*

*((m0, m1), σ) ← A1 h;*

*- :: unit ← assert-spmf (valid-msg m0 ∧ valid-msg m1);*

*b ← coin-spmf;*

*z ← map-spmf (λz. g [∧] z ⊗ (h [∧] (if b then m0 else m1))) (sample-uniform (order G));*

*guess :: bool ← A2 z σ;*

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

**by**(*simp add: bind-map-spmf o-def*)

**also have** ... = *TRY do {*

*x :: nat ← sample-uniform (order G);*

*let h = (g [∧] x);*

*((m0, m1), σ) ← A1 h;*

*- :: unit ← assert-spmf (valid-msg m0 ∧ valid-msg m1);*

*b ← coin-spmf;*

*z ← map-spmf (λz. g [∧] z) (sample-uniform (order G));*

*guess :: bool ← A2 z σ;*

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

```

  by(simp add: sample-uniform-one-time-pad)
also have ... = TRY do {
  x :: nat ← sample-uniform (order  $\mathcal{G}$ );
  let h = ( $\mathbf{g}$  [ $\uparrow$ ] x);
  ((m0, m1),  $\sigma$ ) ←  $\mathcal{A}1$  h;
  - :: unit ← assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  z ← map-spmf ( $\lambda z. \mathbf{g}$  [ $\uparrow$ ] z) (sample-uniform (order  $\mathcal{G}$ ));
  guess :: bool ←  $\mathcal{A}2$  z  $\sigma$ ;
  map-spmf((=) guess) coin-spmf} ELSE coin-spmf
  by(simp add: map-spmf-conv-bind-spmf)
also have ... = coin-spmf
  by(auto simp add: bind-spmf-const map-eq-const-coin-spmf try-bind-spmf-lossless2'
scale-bind-spmf weight-spmf-le-1 scale-scale-spmf)
  ultimately show ?thesis by(simp add: spmf-of-set)
qed

```

**theorem** *abstract-perfect-hiding*:

**shows** *ped-commit.perfect-hiding-ind-cpa*  $\mathcal{A}$

**proof** –

**have** *spmf (ped-commit.hiding-game-ind-cpa  $\mathcal{A}$ ) True* –  $1/2 = 0$

**using** *perfect-hiding by fastforce*

**thus** *?thesis*

**by** (*simp add: abstract-commitment.perfect-hiding-ind-cpa-def abstract-commitment.hiding-advantage-ind-cpa*)

qed

**lemma** *bind-output-cong*:

**assumes**  $x < \text{order } \mathcal{G}$

**shows**  $(x = \text{nat } ((\text{int } b - \text{int } ab) * \text{fst } (\text{bezw } (aa - ac) (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G}))$

$\longleftrightarrow [x = \text{nat } ((\text{int } b - \text{int } ab) * \text{fst } (\text{bezw } (aa - ac) (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G})] (\text{mod order } \mathcal{G})$

**using** *assms cong-less-modulus-unique-nat nat-less-iff by auto*

**lemma** *bind-game-eq-dis-log*:

**shows** *ped-commit.bind-game  $\mathcal{A} = \text{discrete-log.dis-log } (\text{dis-log-}\mathcal{A} \mathcal{A})$*

**proof** –

**note** [*simp*] = *Let-def split-def*

**have** *ped-commit.bind-game  $\mathcal{A} = \text{TRY do } \{$*

*(ck,vk) ← key-gen;*

*(c, m, d, m', d') ←  $\mathcal{A}$  ck;*

*- :: unit ← assert-spmf( $m \neq m' \wedge \text{valid-msg } m \wedge \text{valid-msg } m'$ );*

*let b = verify vk m c d;*

*let b' = verify vk m' c d';*

*- :: unit ← assert-spmf (b  $\wedge$  b');*

*return-spmf True} ELSE return-spmf False*

**by** (*simp add: abstract-commitment.bind-game-alt-def*)

**also have** ... = *TRY do {*

*x :: nat ← sample-uniform (Coset.order  $\mathcal{G}$ );*

*(c, m, d, m', d') ←  $\mathcal{A}$  ( $\mathbf{g}$  [ $\uparrow$ ] x);*

```

- :: unit ← assert-spmf (m ≠ m' ∧ valid-msg m ∧ valid-msg m');
- :: unit ← assert-spmf (c = g [∧] d ⊗ (g [∧] x) [∧] m ∧ c = g [∧] d' ⊗ (g [∧]
x) [∧] m');
  return-spmf True} ELSE return-spmf False
  by(simp add: verify-def key-gen-def)
also have ... = TRY do {
  x :: nat ← sample-uniform (order G);
  (c, m, d, m', d') ← A (g [∧] x);
  - :: unit ← assert-spmf (m ≠ m' ∧ valid-msg m ∧ valid-msg m');
  - :: unit ← assert-spmf (c = g [∧] d ⊗ (g [∧] x) [∧] m ∧ c = g [∧] d' ⊗ (g [∧]
x) [∧] m');
  return-spmf (x = (if (m > m') then (nat ((int d' - int d) * (fst (bezw ((m -
m') (order G)))) mod order G)) else
    (nat ((int d - int d') * (fst (bezw ((m' - m)) (order G)))) mod order
G))))} ELSE return-spmf False
  apply(intro try-spmf-cong bind-spmf-cong[OF refl]; clarsimp?)
  apply(auto simp add: valid-msg-def)
  apply(intro bind-spmf-cong[OF refl]; clarsimp?)+
  apply(simp add: dis-log-break)
  apply(intro bind-spmf-cong[OF refl]; clarsimp?)+
  by(simp add: dis-log-break')
ultimately show ?thesis
apply(simp add: discrete-log.dis-log-def dis-log-A-def cong: bind-spmf-cong-simp)
apply(intro try-spmf-cong bind-spmf-cong[OF refl]; clarsimp?)+
using bind-output-cong by auto
qed

```

```

theorem pedersen-bind: ped-commit.bind-advantage A = discrete-log.advantage
(dis-log-A A)
unfolding abstract-commitment.bind-advantage-def discrete-log.advantage-def
using bind-game-eq-dis-log by simp

```

**end**

```

locale pedersen-asymp =
  fixes G :: nat ⇒ 'grp cyclic-group
  assumes pedersen: ∧η. pedersen (G η)
begin

```

```

sublocale pedersen G η for η by(simp add: pedersen)

```

```

theorem pedersen-correct-asymp:
shows ped-commit.correct n
using abstract-correct by simp

```

```

theorem pedersen-perfect-hiding-asymp:
shows ped-commit.perfect-hiding-ind-cpa n (A n)
by (simp add: abstract-perfect-hiding)

```

```

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- $\mathcal{A}$   $n$  ( $\mathcal{A}$   $n$ )))
  by(simp add: pedersen-bind)

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  $q :: nat$ 
  assumes prime-q: prime  $q$ 
begin

lemma q-gt-0 [simp]:  $q > 0$ 
  by (simp add: prime-q prime-gt-0-nat)

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

definition key-gen ::  $(ck \times vk)$  spmf
  where
    key-gen = do {
       $a :: nat \leftarrow$  sample-uniform  $q$ ;
       $b :: nat \leftarrow$  sample-uniform  $q$ ;
       $x1 :: nat \leftarrow$  sample-uniform  $q$ ;

```

```

let y1 = (a * x1 + b) mod q;
return-spmf ((a,b), (x1,y1))}

```

**definition** *commit* :: *ck*  $\Rightarrow$  *plain*  $\Rightarrow$  (*commit*  $\times$  *opening*) *spmf*

**where**

```

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

```

**fun** *verify* :: *vk*  $\Rightarrow$  *plain*  $\Rightarrow$  *commit*  $\Rightarrow$  *opening*  $\Rightarrow$  *bool*

**where**

```

verify (x1,y1) m c (a,b) = (((c = (m + a) mod q))  $\wedge$  (y1 = (a * x1 + b) mod q))

```

**definition** *valid-msg* :: *plain*  $\Rightarrow$  *bool*

**where** *valid-msg* *msg*  $\equiv$  *msg*  $\in$   $\{..< q\}$

**sublocale** *rivest-commit*: *abstract-commitment key-gen commit verify valid-msg* .

**lemma** *abstract-correct*: *rivest-commit.correct*

**unfolding** *abstract-commitment.correct-def abstract-commitment.correct-game-def*

**by**(*simp add: key-gen-def commit-def bind-spmf-const lossless-weight-spmfD*)

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

**including** *monad-normalisation*

**proof**–

**note** [*simp*] = *Let-def split-def*

**obtain** *A1 A2* **where** [*simp*]: *A* = (*A1*, *A2*) **by**(*cases A*)

**have** *rivest-commit.hiding-game-ind-cpa* (*A1*, *A2*) = *TRY* **do** {

```

a :: nat  $\leftarrow$  sample-uniform q;
x1 :: nat  $\leftarrow$  sample-uniform q;
y1  $\leftarrow$  map-spmf ( $\lambda$  b. (a * x1 + b) mod q) (sample-uniform q);
((m0, m1),  $\sigma$ )  $\leftarrow$  A1 (x1,y1);
- :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
d  $\leftarrow$  coin-spmf;
let c = ((if d then m0 else m1) + a) mod q;
b'  $\leftarrow$  A2 c  $\sigma$ ;
return-spmf (b' = d)} ELSE coin-spmf

```

**unfolding** *abstract-commitment.hiding-game-ind-cpa-def*

**by**(*simp add: commit-def key-gen-def o-def bind-map-spmf*)

**also have** ... = *TRY* **do** {

```

a :: nat  $\leftarrow$  sample-uniform q;
x1 :: nat  $\leftarrow$  sample-uniform q;
y1  $\leftarrow$  sample-uniform q;
((m0, m1),  $\sigma$ )  $\leftarrow$  A1 (x1,y1);
- :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
d  $\leftarrow$  coin-spmf;
let c = ((if d then m0 else m1) + a) mod q;

```

```

    b' ←  $\mathcal{A}2$  c  $\sigma$ ;
    return-spmf (b' = d) } ELSE coin-spmf
  by(simp add: samp-uni-plus-one-time-pad)
also have ... = TRY do {
  x1 :: nat ← sample-uniform q;
  y1 ← sample-uniform q;
  ((m0, m1),  $\sigma$ ) ←  $\mathcal{A}1$  (x1, y1);
  - :: unit ← assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  d ← coin-spmf;
  c ← map-spmf ( $\lambda$  a. ((if d then m0 else m1) + a) mod q) (sample-uniform q);
  b' ←  $\mathcal{A}2$  c  $\sigma$ ;
  return-spmf (b' = d) } ELSE coin-spmf
  by(simp add: o-def bind-map-spmf)
also have ... = TRY do {
  x1 :: nat ← sample-uniform q;
  y1 ← sample-uniform q;
  ((m0, m1),  $\sigma$ ) ←  $\mathcal{A}1$  (x1, y1);
  - :: unit ← assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  d ← coin-spmf;
  c ← sample-uniform q;
  b' :: bool ←  $\mathcal{A}2$  c  $\sigma$ ;
  return-spmf (b' = d) } ELSE coin-spmf
  by(simp add: samp-uni-plus-one-time-pad)
also have ... = TRY do {
  x1 :: nat ← sample-uniform q;
  y1 ← sample-uniform q;
  ((m0, m1),  $\sigma$ ) ←  $\mathcal{A}1$  (x1, y1);
  - :: unit ← assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  c :: nat ← sample-uniform q;
  guess :: bool ←  $\mathcal{A}2$  c  $\sigma$ ;
  map-spmf( (= ) guess ) coin-spmf } ELSE coin-spmf
  by(simp add: map-spmf-conv-bind-spmf)
also have ... = coin-spmf
  by(simp add: map-eq-const-coin-spmf bind-spmf-const try-bind-spmf-lossless2'
    scale-bind-spmf weight-spmf-le-1 scale-scale-spmf)
ultimately show ?thesis
  by(simp add: spmf-of-set)
qed

```

**lemma** *rivest-perfect-hiding*: *rivest-commit.perfect-hiding-ind-cpa*  $\mathcal{A}$

**unfolding** *abstract-commitment.perfect-hiding-ind-cpa-def abstract-commitment.hiding-advantage-ind-cpa-de*

**by**(simp add: *rivest-hiding*)

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

**assumes** *fst-cond*:  $m \neq m' \wedge \text{valid-msg } m \wedge \text{valid-msg } m'$

**and**  $c: c = (m + a) \bmod q \wedge y1 = (a * x1 + b) \bmod q$

**and**  $c': c = (m' + a') \bmod q \wedge y1 = (a' * x1 + b') \bmod q$

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

**shows**  $x1 = (\text{if } (a \bmod q > a' \bmod q) \text{ then } \text{nat } ((\text{int } b' - \text{int } b) * (\text{inverse } (\text{nat}$

```

((int a mod q - int a' mod q) mod q) q) mod q) else
  nat ((int b - int b') * (inverse (nat ((int a' mod q - int a mod q) mod q))
q) mod q))
proof -
  have m: m < q ∧ m' < q using fst-cond valid-msg-def by simp
  have a-a': ¬ [a = a'] (mod q)
  proof -
    have [m + a = m' + a'] (mod q)
    using assms cong-def by blast
    thus ?thesis
    by (metis m fst-cond c c' add.commute cong-less-modulus-unique-nat cong-add-rcancel-nat
cong-mod-right)
  qed
  have cong-y1: [int a * int x1 + int b = int a' * int x1 + int b'] (mod q)
  by (metis c c' cong-def Num.of-nat-simps(4) Num.of-nat-simps(5) cong-int-iff)
  show ?thesis
  proof (cases a mod q > a' mod q)
    case True
      moreover have ⟨(int a mod q - int a' mod q) mod q ≠ 0⟩
      by (metis True comm-monoid-add-class.add-0 diff-add-cancel mod-add-left-eq
mod-diff-eq nat-mod-as-int order-less-irrefl)
      moreover have ((int a mod q - int a' mod q) mod q) < q by simp
      ultimately have ⟨coprime (nat ((int a mod q - int a' mod q) mod q)) q⟩
      using prime-field [of q ⟨nat ((int a mod int q - int a' mod int q) mod int q)⟩]
prime-q
      by (simp flip: of-nat-mod of-nat-diff)
      then have gcd: gcd (nat ((int a mod q - int a' mod q) mod q)) q = 1
      by simp
      hence [int a * int x1 - int a' * int x1 = int b' - int b] (mod q)
      by (smt cong-diff-iff-cong-0 cong-y1 cong-diff cong-diff)
      hence [int a mod q * int x1 - int a' mod q * int x1 = int b' - int b] (mod q)
      proof -
        have [int x1 * (int a mod int q - int a' mod int q) = int x1 * (int a - int
a')] (mod int q)
        by (meson cong-def cong-mult cong-refl mod-diff-eq)
        then show ?thesis
        by (metis (no-types, opaque-lifting) Groups.mult-ac(2) ⟨int a * int x1 - int
a' * int x1 = int b' - int b⟩ (mod int q)⟩ cong-def mod-diff-left-eq mod-diff-right-eq
mod-mult-right-eq)
      qed
      hence [int x1 * (int a mod q - int a' mod q) = int b' - int b] (mod q)
      by (metis int-distrib(3) mult.commute)
      hence [int x1 * (int a mod q - int a' mod q) mod q = int b' - int b] (mod q)
      using cong-def by simp
      hence [int x1 * nat ((int a mod q - int a' mod q) mod q) = int b' - int b] (mod
q)
      by (simp add: True cong-def mod-mult-right-eq)
      hence [int x1 * nat ((int a mod q - int a' mod q) mod q) * inverse (nat ((int
a mod q - int a' mod q) mod q)) q

```

$$= (int\ b' - int\ b) * inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))$$

$$q] (mod\ q)$$
**using** *cong-scalar-right* **by** *blast*
**hence**  $[int\ x1 * (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q) * inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))\ q]$ 

$$= (int\ b' - int\ b) * inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))$$

$$q] (mod\ q)$$
**by** (*simp* *add: more-arith-simps(11)*)
**hence**  $[int\ x1 * 1 = (int\ b' - int\ b) * inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))\ q] (mod\ q)$ 
**using** *inverse gcd*
**by** (*meson cong-scalar-left cong-sym-eq cong-trans*)
**hence**  $[int\ x1 = (int\ b' - int\ b) * inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))\ q] (mod\ q)$ 
**by** *simp*
**hence**  $int\ x1\ mod\ q = ((int\ b' - int\ b) * inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))\ q)\ mod\ q$ 
**using** *cong-def* **by** *fast*
**thus** *?thesis* **using** *x1 True* **by** *simp*
**next**
**case** *False*
**hence** *aa'*:  $a\ mod\ q < a'\ mod\ q$ 
**using** *a-a' cong-refl nat-neq-iff*
**by** (*simp* *add: cong-def*)
**moreover** **have**  $((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q) \neq 0$ 
**by** (*metis aa' comm-monoid-add-class.add-0 diff-add-cancel mod-add-left-eq mod-diff-eq nat-mod-as-int order-less-irrefl*)
**moreover** **have**  $((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q) < q$  **by** *simp*
**ultimately** **have**  $\langle coprime\ (nat\ ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q))\ q \rangle$ 
**using** *prime-field [of q <nat ((int a' mod int q - int a mod int q) mod int q)]*
*prime-q*
**by** (*simp* *flip: of-nat-mod of-nat-diff*)
**then** **have** *gcd: gcd (nat ((int a' mod q - int a mod q) mod q)) q = 1*
**by** *simp*
**have**  $[int\ b - int\ b' = int\ a' * int\ x1 - int\ a * int\ x1] (mod\ q)$ 
**by** (*smt cong-diff-iff-cong-0 cong-y1 cong-diff cong-diff*)
**hence**  $[int\ b - int\ b' = int\ x1 * (int\ a' - int\ a)] (mod\ q)$ 
**using** *int-distrib mult commute* **by** *metis*
**hence**  $[int\ b - int\ b' = int\ x1 * (int\ a'\ mod\ q - int\ a\ mod\ q)] (mod\ q)$ 
**by** (*metis (no-types, lifting) cong-def mod-diff-eq mod-mult-right-eq*)
**hence**  $[int\ b - int\ b' = int\ x1 * (int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q] (mod\ q)$ 
**using** *cong-def* **by** *simp*
**hence**  $[(int\ b - int\ b') * inverse\ (nat\ ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q))\ q$ 

$$= int\ x1 * (int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q * inverse\ (nat\ ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q))\ q] (mod\ q)$$
**using** *cong-scalar-right* **by** *blast*
**hence**  $[(int\ b - int\ b') * inverse\ (nat\ ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q))\ q$ 

$$= int\ x1 * ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q * inverse\ (nat\ ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q))\ q] (mod\ q)$$



**by** (*metis (mono-tags, lifting) cong-def mod-mult-left-eq mod-mult-right-eq more-arith-simps(11)*)  
**hence** \*: [*int x1 \* ((int a' mod q - int a mod q) mod q \* inverse (nat ((int a' mod q - int a mod q) mod q)) q*]  
= (*int b - int b'*) \* *inverse (nat ((int a' mod q - int a mod q) mod q))*  
*q*] (*mod q*)  
**using** *cong-sym-eq by auto*  
**hence** [*int x1 \* 1 = (int b - int b') \* inverse (nat ((int a' mod q - int a mod q) mod q)) q*] (*mod q*)  
**proof** -  
**have** [(*int a' mod int q - int a mod int q*) *mod int q* \* *Number-Theory-Aux.inverse (nat ((int a' mod int q - int a mod int q) mod int q)) q = 1*] (*mod int q*)  
**using** *inverse [of ⟨nat ((int a' mod int q - int a mod int q) mod int q)⟩ q, OF gcd]*  
**by** *simp*  
**then show** *?thesis*  
**by** (*meson \* cong-scalar-left cong-sym-eq cong-trans*)  
**qed**  
**hence** [*int x1 = (int b - int b') \* inverse (nat ((int a' mod q - int a mod q) mod q)) q*] (*mod q*)  
**by** *simp*  
**hence** *int x1 mod q = (int b - int b') \* (inverse (nat ((int a' mod q - int a mod q) mod q)) q) mod q*  
**using** *cong-def by auto*  
**thus** *?thesis using x1 aa' by simp*  
**qed**  
**qed**

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

**shows** *spmf (sample-uniform q) (x mod q) = 1/q*  
**proof** -  
**have** *indicator {..*q*} (x mod q) = 1*  
**using** *q-gt-0 by auto*  
**moreover have** *real (card {..*q*}) = q by simp*  
**ultimately show** *?thesis*  
**by**(*auto simp add: spmf-of-set sample-uniform-def*)  
**qed**

**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** -  
**have** *spmf (do {*  
*x1 ← sample-uniform q;*  
*return-spmf (x1 = y mod q)}) True = spmf (sample-uniform q ≫ (λ x1.*  
*return-spmf x1)) (y mod q)*  
**apply**(*simp only: spmf-bind*)

```

apply(rule Bochner-Integration.integral-cong[OF refl])+
proof -
  fix x :: nat
  have y mod q = x  $\longrightarrow$  indicator {True} (x = (y mod q)) = (indicator {(y mod q)} x)::real
  by simp
  then have indicator {True} (x = y mod q) = (indicator {y mod q} x)::real
  by fastforce
  then show spmf (return-spmf (x = y mod q)) True = spmf (return-spmf x)
  (y mod q)
  by (metis pmf-return spmf-of-pmf-return-pmf spmf-spmf-of-pmf)
qed
thus ?thesis using samp-uni-spmf-mod-q by simp
qed

```

**lemma** bind-game-le-inv-q:

**shows** spmf (rivest-commit.bind-game  $\mathcal{A}$ ) True  $\leq 1 / q$

**proof** -

**let** ?eq =  $\lambda a \ a' \ b \ b'. (=)$

(if (a mod q > a' mod q) then nat ((int b' - int b) \* (inverse (nat ((int a mod q - int a' mod q) mod q))) mod q)

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

**have** spmf (rivest-commit.bind-game  $\mathcal{A}$ ) True = spmf (do {

(ck,(x1,y1))  $\leftarrow$  key-gen;

(c, m, (a,b), m', (a',b'))  $\leftarrow$   $\mathcal{A}$  ck;

- :: unit  $\leftarrow$  assert-spmf (m  $\neq$  m'  $\wedge$  valid-msg m  $\wedge$  valid-msg m');

let b = verify (x1,y1) m c (a,b);

let b' = verify (x1,y1) m' c (a',b');

- :: unit  $\leftarrow$  assert-spmf (b  $\wedge$  b');

return-spmf True}) True

**by**(simp add: abstract-commitment.bind-game-alt-def split-def spmf-try-spmf del: verify.simps)

**also have** ... = spmf (do {

a' :: nat  $\leftarrow$  sample-uniform q;

b' :: nat  $\leftarrow$  sample-uniform q;

x1 :: nat  $\leftarrow$  sample-uniform q;

let y1 = (a' \* x1 + b') mod q;

(c, m, (a,b), m', (a',b'))  $\leftarrow$   $\mathcal{A}$  (a',b');

- :: unit  $\leftarrow$  assert-spmf (m  $\neq$  m'  $\wedge$  valid-msg m  $\wedge$  valid-msg m');

- :: unit  $\leftarrow$  assert-spmf (c = (m + a) mod q  $\wedge$  y1 = (a \* x1 + b) mod q  $\wedge$  c = (m' + a') mod q  $\wedge$  y1 = (a' \* x1 + b') mod q);

return-spmf True}) True

**by**(simp add: key-gen-def Let-def)

**also have** ... = spmf (do {

a'' :: nat  $\leftarrow$  sample-uniform q;

b'' :: nat  $\leftarrow$  sample-uniform q;

x1 :: nat  $\leftarrow$  sample-uniform q;

let y1 = (a'' \* x1 + b'') mod q;

```

(c, m, (a,b), m', (a',b')) ←  $\mathcal{A}$  (a'',b'');
- :: unit ← assert-spmf (m ≠ m' ∧ valid-msg m ∧ valid-msg m');
- :: unit ← assert-spmf (c = (m + a) mod q ∧ y1 = (a * x1 + b) mod q ∧ c
= (m' + a') mod q ∧ y1 = (a' * x1 + b') mod q);
return-spmf (?eq a a' b b' x1)) True
unfolding split-def Let-def
by(rule arg-cong2[where f=spmf, OF - refl] bind-spmf-cong[OF refl])+
(auto simp add: eq-commute samp-uni-break' Let-def split-def valid-msg-def
cong: bind-spmf-cong-simp)
also have ... ≤ spmf (do {
a'' :: nat ← sample-uniform q;
b'' :: nat ← sample-uniform q;
(c, m, (a,(b::nat)), m', (a',b')) ←  $\mathcal{A}$  (a'',b'');
map-spmf (?eq a a' b b') (sample-uniform q)}) True
including monad-normalisation
unfolding split-def Let-def assert-spmf-def
apply(simp add: map-spmf-conv-bind-spmf)
apply(rule ord-spmf-eq-leD ord-spmf-bind-refl)+
apply auto
done
also have ... ≤ 1/q
proof((rule spmf-bind-leI)+, clarify)
fix a a' b b'
define A where A = Collect (?eq a a' b b')
define x1 where x1 = The (?eq a a' b b')
note q-gt-0[simp del]
have A ⊆ {x1} by(auto simp add: A-def x1-def)
hence card (A ∩ {.. $q$ }) ≤ card {x1} by(intro card-mono) auto
also have ... = 1 by simp
finally have spmf (map-spmf (λx. x ∈ A) (sample-uniform q)) True ≤ 1 / q
using q-gt-0 unfolding sample-uniform-def
by(subst map-mem-spmf-of-set)(auto simp add: field-simps)
then show spmf (map-spmf (?eq a a' b b') (sample-uniform q)) True ≤ 1 / q
unfolding A-def mem-Collect-eq .
qed auto
finally show ?thesis .
qed

lemma rivest-bind:
shows rivest-commit.bind-advantage  $\mathcal{A} \leq 1 / q$ 
using bind-game-le-inv-q rivest-commit.bind-advantage-def by simp

end

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

```

**sublocale** *rivest*  $q \eta$  **for**  $\eta$  **by** (*simp add: rivest*)

**theorem** *rivest-correct*:

**shows** *rivest-commit.correct*  $n$   
**using** *abstract-correct* **by** *simp*

**theorem** *rivest-perfect-hiding-asy*:

**assumes** *lossless-A: rivest-commit.lossless* ( $\mathcal{A} \ n$ )  
**shows** *rivest-commit.perfect-hiding-ind-cpa*  $n$  ( $\mathcal{A} \ n$ )  
**by** (*simp add: lossless-A rivest-perfect-hiding*)

**theorem** *rivest-binding-asy*:

**assumes** *negligible* ( $\lambda n. 1 / (q \ n)$ )  
**shows** *negligible* ( $\lambda n. rivest-commit.bind-advantage \ n$  ( $\mathcal{A} \ n$ ))  
**using** *negligible-le rivest-bind assms rivest-commit.bind-advantage-def* **by** *auto*

**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' × 'challenge'*  
*× 'response'*)

**type-synonym** (*'msg', 'response'*) *sim-out* = (*'msg' × 'response'*)

```

type-synonym ('pub-input', 'msg', 'challenge', 'response', 'witness') prover-adversary
  = 'pub-input' ⇒ ('msg', 'challenge', 'response') conv-tuple
    ⇒ ('msg', 'challenge', 'response') conv-tuple ⇒ 'witness' spmf

locale  $\Sigma$ -protocols-base =
  fixes init :: 'pub-input' ⇒ 'witness' ⇒ ('rand × 'msg) spmf — initial message in
 $\Sigma$ -protocol
  and response :: 'rand' ⇒ 'witness' ⇒ 'challenge' ⇒ 'response' spmf
  and check :: 'pub-input' ⇒ 'msg' ⇒ 'challenge' ⇒ 'response' ⇒ bool
  and Rel :: ('pub-input × 'witness) set — The relation the  $\Sigma$  protocol is considered
over
  and S-raw :: 'pub-input' ⇒ 'challenge' ⇒ ('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  $x \in \text{Domain } Rel$  shows  $\exists w. (x, w) \in Rel$ 
using assms by auto

```

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' ⇒ 'witness' ⇒ 'challenge' ⇒ bool spmf
where completeness-game h w e = do {
  (r, a) ← init h w;
  z ← 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  $\equiv (\forall h w e. (h, w) \in Rel \longrightarrow e \in \text{challenge-space} \longrightarrow$ 
 $\text{spmf } (\text{completeness-game } h w e) \text{ True} = 1)$ 

```

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' ⇒ 'witness' ⇒ 'challenge' ⇒ ('msg, 'challenge, 'response)
conv-tuple spmf
where R h w e = do {

```

```

(r,a) ← init h w;
z ← response r w e;
return-spmf (a,e,z)

```

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

**lemma**  $\text{lossless-}S\text{-raw-imp-lossless-}S$ :  $\text{lossless-spmf } (S\text{-raw } h e) \longrightarrow \text{lossless-spmf } (S h e)$   
**by**( $\text{simp add: } S\text{-def}$ )

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

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

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

**lemma**  $\text{special-soundness-alt}$ :

```

special-soundness ↔
  (∀ h a e z e' z'. e ∈ challenge-space → e' ∈ challenge-space → h ∈ valid-pub
    → e ≠ e' → check h a e z ∧ check h a e' z'
    → bind-spmf (Ass h (a,e,z) (a,e',z')) (λ w'. return-spmf ((h,w') ∈
      Rel)) = return-spmf True)
apply( $\text{auto simp add: special-soundness-def map-spmf-conv-bind-spmf[symmetric]$ 
 $\text{map-pmf-eq-return-pmf-iff in-set-spmf lossless-iff-set-pmf-None}$ )
apply( $\text{metis Domain.DomainI in-set-spmf not-Some-eq}$ )
using  $\text{Domain.intros by blast +}$ 

```

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

General lemmas

**lemma**  $\text{lossless-complete-game}$ :

```

assumes  $\text{lossless-init: } \forall h w. \text{lossless-spmf } (\text{init } h w)$ 
and  $\text{lossless-response: } \forall r w e. \text{lossless-spmf } (\text{response } r w e)$ 
shows  $\text{lossless-spmf } (\text{completeness-game } h w e)$ 
by( $\text{simp add: completeness-game-def lossless-init lossless-response split-def}$ )

```

**lemma**  $\text{complete-game-return-true}$ :

```

assumes  $(h,w) \in Rel$ 
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  $h w e = \text{return-spmf } True$ 
proof –
have spmf  $(\text{completeness-game } h w e) True = \text{spmf } (\text{return-spmf } True) True$ 
using assms  $\Sigma\text{-protocol-def completeness-def}$  by fastforce
then have completeness-game  $h w e = \text{return-spmf } True$ 
by (metis (full-types) lossless-complete-game lossless-init lossless-response lossless-return-spmf spmf-False-conv-True spmf-eqI)
then show ?thesis
by (simp add: completeness-game-def)
qed

```

```

lemma HVZK-unfold1:
assumes  $\Sigma\text{-protocol}$ 
shows  $\forall h w e. (h,w) \in Rel \longrightarrow e \in \text{challenge-space} \longrightarrow R h w e = S h e$ 
using assms by (auto simp add:  $\Sigma\text{-protocol-def HVZK-def}$ )

```

```

lemma HVZK-unfold2:
assumes  $\Sigma\text{-protocol}$ 
shows  $\forall h e \text{out}. e \in \text{challenge-space} \longrightarrow h \in \text{valid-pub} \longrightarrow \text{out} \in \text{set-spmf}$ 
 $(S\text{-raw } h e) \longrightarrow \text{check } h (\text{fst out}) e (\text{snd out})$ 
using assms by (auto simp add:  $\Sigma\text{-protocol-def HVZK-def split-def}$ )

```

```

lemma HVZK-unfold2-alt:
assumes  $\Sigma\text{-protocol}$ 
shows  $\forall h a e z. e \in \text{challenge-space} \longrightarrow h \in \text{valid-pub} \longrightarrow (a,z) \in \text{set-spmf}$ 
 $(S\text{-raw } h e) \longrightarrow \text{check } h a e z$ 
using assms by (fastforce simp add:  $\Sigma\text{-protocol-def HVZK-def}$ )

```

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

```

```

for init ::  $'pub\text{-input} \Rightarrow 'witness \Rightarrow ('rand \times 'msg) \text{ spmf}$ 
and response ::  $'rand \Rightarrow 'witness \Rightarrow 'challenge \Rightarrow 'response \text{ spmf}$ 
and check ::  $'pub\text{-input} \Rightarrow 'msg \Rightarrow 'challenge \Rightarrow 'response \Rightarrow \text{bool}$ 
and Rel ::  $('pub\text{-input} \times 'witness) \text{ set}$ 
and S-raw ::  $'pub\text{-input} \Rightarrow 'challenge \Rightarrow ('msg, 'response) \text{ sim-out spmf}$ 
and Ass ::  $('pub\text{-input}, 'msg, 'challenge, 'response, 'witness) \text{ prover-adversary}$ 
and challenge-space ::  $'challenge \text{ set}$ 

```

**and** *valid-pub* :: 'pub-input set  
**and** *G* :: ('pub-input × 'witness) spmf — generates pairs that satisfy the relation  
+  
**assumes** *Σ-prot*: *Σ-protocol* — assume we have a *Σ*-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*]:  $(h,w) \in \text{set-spmf } G \implies h \in \text{Domain Rel}$   
**using** *set-spmf-G-rel* **by** *fast*

**lemma** *set-spmf-G-L* [*simp*]:  $(h,w) \in \text{set-spmf } G \implies h \in L$   
**by** (*metis mem-Collect-eq set-spmf-G-rel L-def*)

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 ⇒ 'witness spmf) ⇒ bool spmf  
**where** *rel-game* *A* = TRY do {  
(*h,w*) ← *G*;  
*w'* ← *A* *h*;  
return-spmf  $((h,w') \in \text{Rel})$ } ELSE return-spmf *False*

**definition** *rel-advantage* :: ('pub-input ⇒ 'witness spmf) ⇒ real  
**where** *rel-advantage* *A* ≡ spmf (*rel-game* *A*) *True*

We now define the algorithms that define the commitment scheme constructed from a *Σ*-protocol.

**definition** *key-gen* :: ('pub-input × ('pub-input × 'witness)) spmf  
**where**  
*key-gen* = do {  
(*x,w*) ← *G*;  
return-spmf (*x*, (*x,w*))}

**definition** *commit* :: 'pub-input ⇒ 'challenge ⇒ ('msg × 'response) spmf  
**where**  
*commit* *x e* = do {  
(*a,e,z*) ← *S* *x e*;  
return-spmf (*a*, *z*)}

**definition** *verify* :: ('pub-input × 'witness) ⇒ 'challenge ⇒ 'msg ⇒ 'response ⇒ bool  
**where** *verify* *x e a z* = (*check* (*fst* *x*) *a e z*)

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



**definition** *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** *abstract-com*: *abstract-commitment key-gen commit verify valid-msg* .

**Correctness lemma** *commit-correct*:

**shows** *abstract-com.correct*

**including** *monad-normalisation*

**proof** –

**have**  $\forall m \in \text{challenge-space}. \text{abstract-com.correct-game } m = \text{return-spmf True}$

**proof**

**fix**  $m$

**assume**  $m: m \in \text{challenge-space}$

**show** *abstract-com.correct-game*  $m = \text{return-spmf True}$

**proof** –

**have** *abstract-com.correct-game*  $m = \text{do}$  {

$(ck, (vk1, vk2)) \leftarrow \text{key-gen};$

$(a, e, z) \leftarrow S \text{ ck } m;$

$\text{return-spmf } (\text{check } vk1 \ a \ m \ z)$ }

**unfolding** *abstract-com.correct-game-def*

**by**(*simp add: commit-def verify-def split-def*)

**also have** ... =  $\text{do}$  {

$(x, w) \leftarrow G;$

$\text{let } (ck, (vk1, vk2)) = (x, (x, w));$

$(a, e, z) \leftarrow S \text{ ck } m;$

$\text{return-spmf } (\text{check } vk1 \ a \ m \ z)$ }

**by**(*simp add: key-gen-def split-def*)

**also have** ... =  $\text{do}$  {

$(x, w) \leftarrow G;$

$(a, e, z) \leftarrow S \ x \ m;$

$\text{return-spmf } (\text{check } x \ a \ m \ z)$ }

**by**(*simp add: Let-def*)

**also have** ... =  $\text{do}$  {

$(x, w) \leftarrow G;$

$(a, e, z) \leftarrow R \ x \ w \ m;$

$\text{return-spmf } (\text{check } x \ a \ m \ z)$ }

**using**  $\Sigma\text{-prot HVZK-unfold1 } m$

**by**(*intro bind-spmf-cong bind-spmf-cong[OF refl]; clarsimp?*)

**also have** ... =  $\text{do}$  {

$(x, w) \leftarrow G;$

$(r, a) \leftarrow \text{init } x \ w;$

$z \leftarrow \text{response } r \ w \ m;$

$\text{return-spmf } (\text{check } x \ a \ m \ z)$ }

**by**(*simp add: R-def split-def*)

**also have** ... =  $\text{do}$  {

$(x, w) \leftarrow G;$

$\text{return-spmf True}$ }

**apply**(*intro bind-spmf-cong bind-spmf-cong[OF refl]; clarsimp?*)

```

using complete-game-return-true lossless-init lossless-response  $\Sigma$ -prot  $\Sigma$ -protocol-def
by(simp add: split-def completeness-game-def  $\Sigma$ -protocols-base. $\Sigma$ -protocol-def
m cong: bind-spmf-cong-simp)
ultimately show abstract-com.correct-game m = return-spmf True
by(simp add: bind-spmf-const lossless-G lossless-weight-spmfD split-def)
qed
qed
thus ?thesis
using abstract-com.correct-def abstract-com.valid-msg-set-def valid-msg-def by
simp
qed

```

**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** perfect-hiding:

```

shows abstract-com.perfect-hiding-ind-cpa  $\mathcal{A}$ 
including monad-normalisation
proof –
obtain  $\mathcal{A}1$   $\mathcal{A}2$  where [simp]:  $\mathcal{A} = (\mathcal{A}1, \mathcal{A}2)$  by(cases  $\mathcal{A}$ )
have abstract-com.hiding-game-ind-cpa ( $\mathcal{A}1, \mathcal{A}2$ ) = TRY do {
  (x,w)  $\leftarrow$  G;
  ((m0, m1),  $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  (x,w);
  - :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  b  $\leftarrow$  coin-spmf;
  (a,e,z)  $\leftarrow$  S x (if b then m0 else m1);
  b'  $\leftarrow$   $\mathcal{A}2$  a  $\sigma$ ;
  return-spmf (b' = b)} ELSE coin-spmf
by(simp add: abstract-com.hiding-game-ind-cpa-def commit-def; simp add: key-gen-def
split-def)
also have ... = TRY do {
  (x,w)  $\leftarrow$  G;
  ((m0, m1),  $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  (x,w);
  - :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  b :: bool  $\leftarrow$  coin-spmf;
  (a,e,z)  $\leftarrow$  R x w (if b then m0 else m1);
  b' :: bool  $\leftarrow$   $\mathcal{A}2$  a  $\sigma$ ;
  return-spmf (b' = b)} ELSE coin-spmf
apply(intro try-spmf-cong bind-spmf-cong[OF refl]; clarsimp?)+
by(simp add:  $\Sigma$ -prot HVZK-unfold1 valid-msg-def)
also have ... = TRY do {
  (x,w)  $\leftarrow$  G;
  ((m0, m1),  $\sigma$ )  $\leftarrow$   $\mathcal{A}1$  (x,w);
  - :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  b  $\leftarrow$  coin-spmf;
  (r,a)  $\leftarrow$  init x w;
  z :: 'response  $\leftarrow$  response r w (if b then m0 else m1);
  guess :: bool  $\leftarrow$   $\mathcal{A}2$  a  $\sigma$ ;

```

```

    return-spmf(guess = b) } ELSE coin-spmf
  using  $\Sigma$ -protocols-base.R-def
  by(simp add: bind-map-spmf o-def R-def split-def)
also have ... = TRY do {
  (x,w)  $\leftarrow$  G;
  ((m0, m1),  $\sigma$ )  $\leftarrow$  A1 (x,w);
  - :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  b  $\leftarrow$  coin-spmf;
  (r,a)  $\leftarrow$  init x w;
  guess :: bool  $\leftarrow$  A2 a  $\sigma$ ;
  return-spmf(guess = b) } ELSE coin-spmf
  by(simp add: bind-spmf-const lossless-response lossless-weight-spmfD)
also have ... = TRY do {
  (x,w)  $\leftarrow$  G;
  ((m0, m1),  $\sigma$ )  $\leftarrow$  A1 (x,w);
  - :: unit  $\leftarrow$  assert-spmf (valid-msg m0  $\wedge$  valid-msg m1);
  (r,a)  $\leftarrow$  init x w;
  guess :: bool  $\leftarrow$  A2 a  $\sigma$ ;
  map-spmf( (=) guess) coin-spmf } ELSE coin-spmf
  apply(simp add: map-spmf-conv-bind-spmf)
  by(simp add: split-def)
also have ... = coin-spmf
  by(auto simp add: map-eq-const-coin-spmf try-bind-spmf-lossless2' Let-def split-def
bind-spmf-const scale-bind-spmf weight-spmf-le-1 scale-scale-spmf)
  ultimately have spmf (abstract-com.hiding-game-ind-cpa A) True = 1/2
  by(simp add: spmf-of-set)
  thus ?thesis
  by (simp add: abstract-com.perfect-hiding-ind-cpa-def abstract-com.hiding-advantage-ind-cpa-def)
qed

```

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

**lemma** *bind-advantage*:

**shows** abstract-com.bind-advantage A  $\leq$  rel-advantage (adversary A)

**proof** –

**have** abstract-com.bind-game A = TRY do {  
 (x,w)  $\leftarrow$  G;  
 (c, m, d, m', d')  $\leftarrow$  A x;  
 - :: unit  $\leftarrow$  assert-spmf (m  $\neq$  m'  $\wedge$  m  $\in$  challenge-space  $\wedge$  m'  $\in$  challenge-space);  
 let b = check x c m d;  
 let b' = check x c m' d';  
 - :: unit  $\leftarrow$  assert-spmf (b  $\wedge$  b');

```

w' ← Ass x (c,m, d) (c,m', d');
return-spmf ((x,w') ∈ Rel)} ELSE return-spmf False
  unfolding abstract-com.bind-game-alt-def
  apply(simp add: key-gen-def verify-def Let-def split-def valid-msg-def)
  apply(intro try-spmf-cong bind-spmf-cong[OF refl]; clarsimp?)+
  using special-soundness-def Σ-prot Σ-protocol-def special-soundness-alt spe-
cial-soundness-def set-spmf-G-rel set-spmf-G-domain-rel
  by (smt basic-trans-rules(31) bind-spmf-cong domain-subset-valid-pub)
  hence abstract-com.bind-advantage  $\mathcal{A} \leq \text{spm}f$  (TRY do {
(x,w) ← G;
(c, m, d, m', d') ←  $\mathcal{A}$  x;
w' ← Ass x (c,m, d) (c,m', d');
return-spmf ((x,w') ∈ Rel)} ELSE return-spmf False) True
  unfolding abstract-com.bind-advantage-def
  apply(simp add: spmf-try-spmf)
  apply(rule ord-spmf-eq-leD)
  apply(rule ord-spmf-bind-reflI;clarsimp)+
  by(simp add: assert-spmf-def)
thus ?thesis
  by(simp add: rel-game-def adversary-def split-def rel-advantage-def)
qed

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  $\mathcal{G} > 0$

using *prime-order prime-gt-0-nat* by *blast*

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

**lemma** *lossless-init*: *lossless-spmf* (*init* *h w*)  
**by**(*simp add: init-def*)

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

**lemma** *lossless-response*: *lossless-spmf* (*response* *r w c*)  
**by**(*simp add: response-def*)

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

**lemma** *lossless-G*: *lossless-spmf* *G*  
**by**(*simp add: G-def*)

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

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

**definition** *S2* :: *'grp*  $\Rightarrow$  *challenge*  $\Rightarrow$  (*'grp msg*, *response*) *sim-out* *spmf*  
**where** *S2* *h e* = *do* {  
*c*  $\leftarrow$  *sample-uniform* (*order* *G*);  
*let* *a* =  $\mathbf{g} [\wedge] c \otimes (\text{inv } (h [\wedge] e))$ ;  
*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 A h = do {
    (c, e, z, e', z')  $\leftarrow$  A h;
    - :: unit  $\leftarrow$  assert-spmf (e > e'  $\wedge$   $\neg$  [e = e'] (mod order  $\mathcal{G}$ )  $\wedge$  (gcd (e - e') (order
 $\mathcal{G}$ ) = 1)  $\wedge$  c  $\in$  carrier  $\mathcal{G}$ );
    - :: unit  $\leftarrow$  assert-spmf (((c  $\otimes$  h [ ] e) = g [ ] z)  $\wedge$  (c  $\otimes$  h [ ] e') = g [ ] z');
    return-spmf (nat ((int z - int z') * inverse ((e - e') (order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ )))}

```

```

sublocale discrete-log: dis-log  $\mathcal{G}$ 
unfolding dis-log-def by simp

```

**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
apply unfold-locales
by(simp add: R-DL-def valid-pub-def; blast)

```

The Schnorr  $\Sigma$ -protocol is complete.

**lemma** completeness: Schnorr- $\Sigma$ .completeness

**proof** –

```

have g [ ] y  $\otimes$  (g [ ] w') [ ] e = g [ ] (y + w' * e) for y e w' :: nat
using nat-pow-pow nat-pow-mult by simp
then show ?thesis
unfolding Schnorr- $\Sigma$ .completeness-game-def Schnorr- $\Sigma$ .completeness-def

```

**by**(*auto simp add: init-def response-def check-def pow-generator-mod R-DL-def add.commute bind-spmf-const*)

**qed**

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

**lemma** *zr-rewrite:*

**assumes**  $z: z = (x*c + r) \text{ mod } (\text{order } \mathcal{G})$

**and**  $r: r < \text{order } \mathcal{G}$

**shows**  $(z + (\text{order } \mathcal{G}) * x*c - x*c) \text{ mod } (\text{order } \mathcal{G}) = r$

**proof**(*cases x = 0*)

**case** *True*

**then show** *?thesis* **using** *assms* **by** *simp*

**next**

**case** *x-neq-0: False*

**then show** *?thesis*

**proof**(*cases c = 0*)

**case** *True*

**then show** *?thesis*

**by** (*simp add: assms*)

**next**

**case** *False*

**have** *cong*:  $[z + (\text{order } \mathcal{G}) * x*c = x*c + r] \text{ (mod } (\text{order } \mathcal{G}))$

**by** (*simp add: cong-def mult.assoc z*)

**hence**  $[z + (\text{order } \mathcal{G}) * x*c - x*c = r] \text{ (mod } (\text{order } \mathcal{G}))$

**proof**–

**have**  $z + (\text{order } \mathcal{G}) * x*c > x*c$

**by** (*metis One-nat-def mult-less-cancel2 n-less-m-mult-n neq0-conv prime-gt-1-nat prime-order trans-less-add2 x-neq-0 False*)

**then show** *?thesis*

**by** (*metis cong add-diff-inverse-nat cong-add-lcancel-nat less-imp-le linorder-not-le*)

**qed**

**then show** *?thesis*

**by**(*simp add: cong-def r*)

**qed**

**qed**

**lemma** *h-sub-rewrite:*

**assumes**  $h = \mathbf{g} [\ ] x$

**and**  $z: z < \text{order } \mathcal{G}$

**shows**  $\mathbf{g} [\ ] ((z + (\text{order } \mathcal{G}) * x*c - x*c)) = \mathbf{g} [\ ] z \otimes \text{inv } (h [\ ] c)$

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

**proof**(*cases x = 0*)

**case** *True*

**then show** *?thesis* **using** *assms* **by** *simp*

**next**

**case** *x-neq-0: False*

**then show** *?thesis*

```

proof–
  have  $(z + \text{order } \mathcal{G} * x * c - x * c) = (z + (\text{order } \mathcal{G} * x * c - x * c))$ 
    using  $z$  by (simp add: less-imp-le-nat mult-le-mono)
  then have  $lhs: ?lhs = \mathbf{g} [\wedge] z \otimes \mathbf{g} [\wedge] ((\text{order } \mathcal{G}) * x * c - x * c)$ 
    by(simp add: nat-pow-mult)
  have  $\mathbf{g} [\wedge] ((\text{order } \mathcal{G}) * x * c - x * c) = \text{inv } (h [\wedge] c)$ 
  proof(cases c = 0)
    case True
      then show ?thesis by simp
    next
      case False
      hence  $\text{bound}: ((\text{order } \mathcal{G}) * x * c - x * c) > 0$ 
        using assms x-neq-0 prime-gt-1-nat prime-order by auto
      then have  $\mathbf{g} [\wedge] ((\text{order } \mathcal{G}) * x * c - x * c) = \mathbf{g} [\wedge] \text{int } ((\text{order } \mathcal{G}) * x * c - x * c)$ 
        by (simp add: int-pow-int)
      also have  $\dots = \mathbf{g} [\wedge] \text{int } ((\text{order } \mathcal{G}) * x * c) \otimes \text{inv } (\mathbf{g} [\wedge] (x * c))$ 
        by (metis bound generator-closed int-ops(6) int-pow-int of-nat-eq-0-iff
of-nat-less-0-iff of-nat-less-iff int-pow-diff)
      also have  $\dots = \mathbf{g} [\wedge] ((\text{order } \mathcal{G}) * x * c) \otimes \text{inv } (\mathbf{g} [\wedge] (x * c))$ 
        by (metis int-pow-int)
      also have  $\dots = \mathbf{g} [\wedge] ((\text{order } \mathcal{G}) * x * c) \otimes \text{inv } ((\mathbf{g} [\wedge] x) [\wedge] c)$ 
        by(simp add: nat-pow-pow)
      also have  $\dots = \mathbf{g} [\wedge] ((\text{order } \mathcal{G}) * x * c) \otimes \text{inv } (h [\wedge] c)$ 
        using assms by simp
      also have  $\dots = \mathbf{1} \otimes \text{inv } (h [\wedge] c)$ 
        using generator-pow-order
        by (metis generator-closed mult-is-0 nat-pow-0 nat-pow-pow)
      ultimately show ?thesis
        by (simp add: assms(1))
    qed
  then show ?thesis using lhs by simp
qed
qed

```

**lemma** *hvzk-R-rewrite-grp*:

```

  fixes  $x \ c \ r :: \text{nat}$ 
  assumes  $r < \text{order } \mathcal{G}$ 
  shows  $\mathbf{g} [\wedge] (((x * c + \text{order } \mathcal{G} - r) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * x * c - x * c) \bmod \text{order } \mathcal{G}) = \text{inv } \mathbf{g} [\wedge] r$ 
    (is ?lhs = ?rhs)

```

**proof**–

```

  have  $[(x * c + \text{order } \mathcal{G} - r) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * x * c - x * c = \text{order } \mathcal{G} - r] \ (\bmod \text{order } \mathcal{G})$ 
  proof–
    have  $[(x * c + \text{order } \mathcal{G} - r) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * x * c - x * c = x * c + \text{order } \mathcal{G} - r + \text{order } \mathcal{G} * x * c - x * c] \ (\bmod \text{order } \mathcal{G})$ 
    by (smt cong-def One-nat-def add-diff-inverse-nat cong-diff-nat less-imp-le-nat linorder-not-less mod-add-left-eq mult.assoc n-less-m-mult-n prime-gt-1-nat prime-order trans-less-add2 zero-less-diff)
  
```



**hence**  $[(x * c + \text{order } \mathcal{G} - r) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * x * c - x * c$   
 $= \text{order } \mathcal{G} - r + \text{order } \mathcal{G} * x * c] \pmod{\text{order } \mathcal{G}}$   
**using** *assms* **by** *auto*  
**thus** *?thesis*  
**by** (*simp add: cong-def mult.assoc*)  
**qed**  
**hence**  $\mathbf{g} [\wedge] ((x * c + \text{order } \mathcal{G} - r) \bmod \text{order } \mathcal{G} + \text{order } \mathcal{G} * x * c - x * c) =$   
 $\mathbf{g} [\wedge] (\text{order } \mathcal{G} - r)$   
**using** *finite-carrier pow-generator-eq-iff-cong* **by** *blast*  
**thus** *?thesis using neg-power-inverse*  
**by** (*simp add: assms inverse-pow-pow pow-generator-mod*)  
**qed**

**lemma** *hv-zk*:

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

**proof** –

**have**  $\text{Schnorr-}\Sigma.R \ h \ x \ c = \text{do } \{$   
 $r \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $\text{let } z = (x * c + r) \bmod (\text{order } \mathcal{G});$   
 $\text{let } a = \mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * x * c - x * c) \bmod (\text{order } \mathcal{G}));$   
 $\text{return-spmf } (a, c, z)\}$   
**apply** (*simp add: Let-def Schnorr-}\Sigma.R-def init-def response-def*)  
**using** *assms zr-rewrite R-DL-def*  
**by** (*simp cong: bind-spmf-cong-simp*)  
**also have**  $\dots = \text{do } \{$   
 $z \leftarrow \text{map-spmf } (\lambda r. (x * c + r) \bmod (\text{order } \mathcal{G})) (\text{sample-uniform } (\text{order } \mathcal{G}));$   
 $\text{let } a = \mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * x * c - x * c) \bmod (\text{order } \mathcal{G}));$   
 $\text{return-spmf } (a, c, z)\}$   
**by** (*simp add: bind-map-spmf o-def Let-def*)  
**also have**  $\dots = \text{do } \{$   
 $z \leftarrow (\text{sample-uniform } (\text{order } \mathcal{G}));$   
 $\text{let } a = \mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * x * c - x * c));$   
 $\text{return-spmf } (a, c, z)\}$   
**by** (*simp add: samp-uni-plus-one-time-pad pow-generator-mod*)  
**also have**  $\dots = \text{do } \{$   
 $z \leftarrow (\text{sample-uniform } (\text{order } \mathcal{G}));$   
 $\text{let } a = \mathbf{g} [\wedge] z \otimes \text{inv } (h [\wedge] c);$   
 $\text{return-spmf } (a, c, z)\}$   
**using** *h-sub-rewrite assms R-DL-def*  
**by** (*simp cong: bind-spmf-cong-simp*)  
**ultimately show** *?thesis*  
**by** (*simp add: Schnorr-}\Sigma.S-def S2-def map-spmf-conv-bind-spmf*)  
**qed**

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

**lemma** *honest-verifier-ZK*:

**shows** *Schnorr- $\Sigma$ .HVZK*  
**unfolding** *Schnorr- $\Sigma$ .HVZK-def*  
**by** (*auto simp add: hv-zk R-DL-def S2-def check-def valid-pub-def challenge-space-def cyclic-group-assoc*)

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** –  
**have**  $\text{gcd} (\text{gcd } (\text{nat } (\text{int } e - \text{int } e') \text{ mod } (\text{order } \mathcal{G})) (\text{order } \mathcal{G})) (\text{order } \mathcal{G}) = 1$   
**using** *prime-field*  
**by** (*metis Primes.prime-nat-def assms(1) assms(2) coprime-imp-gcd-eq-1 diff-is-0-eq less-imp-diff-less mod-less nat-minus-as-int not-less schnorr-base.prime-order schnorr-base-axioms*)  
**have**  $a = \mathbf{g} [\cdot] z \otimes \text{inv } (h [\cdot] e)$   
**using**  $a\text{-mem}$   
**by** (*simp add: h-mem group.inv-solve-right*)  
**moreover have**  $a = \mathbf{g} [\cdot] z' \otimes \text{inv } (h [\cdot] e')$   
**using**  $a'\text{-mem}$   
**by** (*simp add: h-mem group.inv-solve-right*)  
**ultimately have**  $\mathbf{g} [\cdot] z \otimes h [\cdot] e' = \mathbf{g} [\cdot] z' \otimes h [\cdot] e$   
**using**  $h\text{-mem}$   
**by** (*metis (no-types, lifting) a a' h-mem a-mem cyclic-group-assoc cyclic-group-commute nat-pow-closed*)  
**moreover obtain**  $t :: \text{nat}$  **where**  $t: h = \mathbf{g} [\cdot] t$   
**using**  $h\text{-mem generatorE}$  **by** *blast*  
**ultimately have**  $\mathbf{g} [\cdot] (z + t * e') = \mathbf{g} [\cdot] (z' + t * e)$   
**by** (*simp add: monoid.nat-pow-mult nat-pow-pow*)  
**hence**  $[z + t * e' = z' + t * e] (\text{mod } \text{order } \mathcal{G})$   
**using** *group-eq-pow-eq-mod order-gt-0* **by** *blast*  
**hence**  $[\text{int } z + \text{int } t * \text{int } e' = \text{int } z' + \text{int } t * \text{int } e] (\text{mod } \text{order } \mathcal{G})$   
**using** *cong-int-iff* **by** *force*  
**hence**  $[\text{int } z - \text{int } z' = \text{int } t * \text{int } e - \text{int } t * \text{int } e'] (\text{mod } \text{order } \mathcal{G})$   
**by** (*smt cong-iff-lin*)  
**hence**  $[\text{int } z - \text{int } z' = \text{int } t * (\text{int } e - \text{int } e')] (\text{mod } \text{order } \mathcal{G})$   
**by** (*simp add:  $\langle [\text{int } z - \text{int } z' = \text{int } t * \text{int } e - \text{int } t * \text{int } e'] (\text{mod int } (\text{order } \mathcal{G})) \rangle$  right-diff-distrib*)  
**hence**  $[\text{int } z - \text{int } z' = \text{int } t * (\text{int } e - \text{int } e')] (\text{mod } \text{order } \mathcal{G})$   
**by** (*meson cong-diff cong-mod-left cong-mult cong-refl cong-trans*)  
**hence**  $*: [\text{int } z - \text{int } z' = \text{int } t * (\text{int } e - \text{int } e')] (\text{mod } \text{order } \mathcal{G})$

```

using assms
by (simp add: int-ops(9) of-nat-diff)
hence [int z - int z' = int t * nat (int e - int e')] (mod order G)
using assms
by auto
hence **: [(int z - int z') * fst (bezw ((nat (int e - int e'))) (order G))]
          = int t * (nat (int e - int e')
          * fst (bezw ((nat (int e - int e'))) (order G))] (mod order G)
by (smt <[int z - int z' = int t * (int e - int e')] (mod int (order G))> assms(1))
assms(2)
cong-scalar-right int-nat-eq less-imp-of-nat-less mod-less more-arith-simps(11)
nat-less-iff of-nat-0-le-iff)
hence [(int z - int z') * fst (bezw ((nat (int e - int e'))) (order G)) = int t * 1]
(mod order G)
by (metis (no-types, opaque-lifting) gcd inverse assms(2) cong-scalar-left cong-trans
less-imp-diff-less mod-less mult.comm-neutral nat-minus-as-int)
hence [(int z - int z') * fst (bezw ((nat (int e - int e'))) (order G))]
      = t] (mod order G) by simp
hence [ ((int z - int z') * fst (bezw ((nat (int e - int e'))) (order G))) mod order
G
      = t] (mod order G)
using cong-mod-left by blast
hence **: [nat (((int z - int z') * fst (bezw ((nat (int e - int e'))) (order
G))) mod order G]
      = t] (mod order G)
by (metis cong-def mod-mod-trivial nat-int of-nat-mod)
hence g [ $\Uparrow$ ] (nat (((int z - int z') * fst (bezw ((nat (int e - int e'))) (order
G))) mod order G) = g [ $\Uparrow$ ] t
using cyclic-group.pow-generator-eq-iff-cong cyclic-group-axioms order-gt-0 or-
der-gt-0-iff-finite by blast
thus ?thesis using t
by (simp add: nat-minus-as-int)
qed

```

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

**lemma** *special-soundness*:

**shows** *Schnorr- $\Sigma$ .special-soundness*

**unfolding** *Schnorr- $\Sigma$ .special-soundness-def*

**by**(*auto simp add: valid-pub-def ss-rewrite challenge-space-def split-def ss-adversary-def*  
*check-def R-DL-def Let-def*)

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*

**by**(*simp add: Schnorr- $\Sigma$ . $\Sigma$ -protocol-def completeness honest-verifier-ZK special-soundness*)

Having proven the  $\Sigma$ -protocol property is satisfied we can show the com-

mitment 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
unfolding  $\Sigma$ -protocols-to-commitments-def  $\Sigma$ -protocols-to-commitments-axioms-def
apply(auto simp add:  $\Sigma$ -protocols-base-def)
apply(simp add: R-DL-def valid-pub-def)
apply(auto simp add: sigma-protocol lossless-G lossless-init lossless-response)
by(simp add: R-DL-def G-def)

```

```

lemma Schnorr- $\Sigma$ -commit.abstract-com.correct
by(fact Schnorr- $\Sigma$ -commit.commit-correct)

```

```

lemma Schnorr- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa A
by(fact Schnorr- $\Sigma$ -commit.perfect-hiding)

```

```

lemma rel-adv-eq-dis-log-adv:
Schnorr- $\Sigma$ -commit.rel-advantage A = discrete-log.advantage A
proof –
have Schnorr- $\Sigma$ -commit.rel-game A = discrete-log.dis-log A
unfolding Schnorr- $\Sigma$ -commit.rel-game-def discrete-log.dis-log-def
by(auto intro: try-spmf-cong bind-spmf-cong[OF refl]
simp add: G-def R-DL-def cong-less-modulus-unique-nat group-eq-pow-eq-mod
finite-carrier pow-generator-eq-iff-cong)
thus ?thesis
using Schnorr- $\Sigma$ -commit.rel-advantage-def discrete-log.advantage-def by simp
qed

```

```

lemma bind-advantage-bound-dis-log:
Schnorr- $\Sigma$ -commit.abstract-com.bind-advantage A  $\leq$  discrete-log.advantage (Schnorr- $\Sigma$ -commit.adversary
A)
using Schnorr- $\Sigma$ -commit.bind-advantage rel-adv-eq-dis-log-adv by simp

```

**end**

```

locale schnorr-asymp =
fixes G :: nat  $\Rightarrow$  'grp cyclic-group
assumes schnorr:  $\bigwedge \eta$ . schnorr-sigma-protocol (G  $\eta$ )
begin

```

```

sublocale schnorr-sigma-protocol G  $\eta$  for  $\eta$ 
by(simp add: schnorr)

```

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

```

theorem sigma-protocol:
shows Schnorr- $\Sigma$ . $\Sigma$ -protocol n
by(simp add: sigma-protocol)

```

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** *asyp-correct*: *Schnorr- $\Sigma$ -commit.abstract-com.correct* *n*  
**using** *Schnorr- $\Sigma$ -commit.commit-correct* **by** *simp*

**lemma** *asyp-perfect-hiding*: *Schnorr- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa* *n* (*A n*)  
**using** *Schnorr- $\Sigma$ -commit.perfect-hiding* **by** *blast*

**lemma** *asyp-computational-binding*:  
**assumes** *negligible* ( $\lambda n. discrete-log.advantage\ n\ (Schnorr-\Sigma-commit.adversary\ n\ (A\ n))$ )  
**shows** *negligible* ( $\lambda n. Schnorr-\Sigma-commit.abstract-com.bind-advantage\ n\ (A\ n)$ )  
**using** *Schnorr- $\Sigma$ -commit.bind-advantage* *assms* *Schnorr- $\Sigma$ -commit.abstract-com.bind-advantage-def* *negligible-le* *bind-advantage-bound-dis-log* **by** *auto*

**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**  $g' = g \ [\hat{\cdot}] \ x$

**lemma** *or-gt-1*: *order*  $\mathcal{G} > 1$   
**using** *prime-order*  
**using** *prime-gt-1-nat* **by** *blast*

**lemma** *or-gt-0* [*simp*]: *order*  $\mathcal{G} > 0$   
**using** *or-gt-1* **by** *simp*

**type-synonym** *witness* = *nat*  
**type-synonym** *rand* = *nat*  
**type-synonym** *'grp' msg* = *'grp' × 'grp'*  
**type-synonym** *response* = *nat*  
**type-synonym** *challenge* = *nat*  
**type-synonym** *'grp' pub-in* = *'grp' × 'grp'*

**definition** *G* = *do* {  
*w* ← *sample-uniform* (*order G*);  
*return-spmf* ((**g** [ $\uparrow$ ] *w*, *g'* [ $\uparrow$ ] *w*), *w*)

**lemma** *lossless-G*: *lossless-spmf G*  
**by**(*simp add: G-def*)

**definition** *challenge-space* = {..*order G*}

**definition** *init* :: *'grp pub-in* ⇒ *witness* ⇒ (*rand* × *'grp msg*) *spmf*  
**where** *init h w* = *do* {  
*let* (*h*, *h'*) = *h*;  
*r* ← *sample-uniform* (*order G*);  
*return-spmf* (*r*, **g** [ $\uparrow$ ] *r*, *g'* [ $\uparrow$ ] *r*)

**lemma** *lossless-init*: *lossless-spmf (init h w)*  
**by**(*simp add: init-def*)

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

**lemma** *lossless-response*: *lossless-spmf (response r w e)*  
**by**(*simp add: response-def*)

**definition** *check* :: *'grp pub-in* ⇒ *'grp msg* ⇒ *challenge* ⇒ *response* ⇒ *bool*  
**where** *check h a e z* = (*fst a* ⊗ (*fst h* [ $\uparrow$ ] *e*) = **g** [ $\uparrow$ ] *z* ∧ *snd a* ⊗ (*snd h* [ $\uparrow$ ] *e*)  
= *g'* [ $\uparrow$ ] *z* ∧ *fst a* ∈ *carrier G* ∧ *snd a* ∈ *carrier G*)

**definition** *R* :: (*'grp pub-in* × *witness*) *set*  
**where** *R* = {(*h*, *w*). (*fst h* = **g** [ $\uparrow$ ] *w* ∧ *snd h* = *g'* [ $\uparrow$ ] *w*)}

**definition** *S2* :: *'grp pub-in* ⇒ *challenge* ⇒ (*'grp msg*, *response*) *sim-out spmf*  
**where** *S2 H c* = *do* {  
*let* (*h*, *h'*) = *H*;  
*z* ← (*sample-uniform* (*order G*));  
*let a* = **g** [ $\uparrow$ ] *z* ⊗ *inv* (*h* [ $\uparrow$ ] *c*);  
*let a'* = *g'* [ $\uparrow$ ] *z* ⊗ *inv* (*h'* [ $\uparrow$ ] *c*);  
*return-spmf* ((*a*, *a'*), *z*)

**definition** *ss-adversary* :: *'grp pub-in* ⇒ (*'grp msg*, *challenge*, *response*) *conv-tuple*  
⇒ (*'grp msg*, *challenge*, *response*) *conv-tuple* ⇒ *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  $\mathcal{G}$  × carrier  $\mathcal{G}$

**end**

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

**lemma** *g'-in-carrier* [*simp*]:  $g' \in \text{carrier } \mathcal{G}$   
**by**(*simp add: g'-def*)

**sublocale** *chaum-ped-sigma*:  $\Sigma$ -protocols-base *init response check R S2 ss-adversary*  
*challenge-space valid-pub*  
**by** *unfold-locale (auto simp add: R-def valid-pub-def)*

**lemma** *completeness*:

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

**proof** –

**have**  $g' [\wedge] y \otimes (g' [\wedge] w') [\wedge] e = g' [\wedge] ((w' * e + y) \text{ mod order } \mathcal{G})$  **for**  $y \ e \ w'$   
**by** (*simp add: Groups.add-ac(2) pow-carrier-mod nat-pow-pow nat-pow-mult*)  
**moreover have**  $\mathbf{g} [\wedge] y \otimes (\mathbf{g} [\wedge] w') [\wedge] e = \mathbf{g} [\wedge] ((w' * e + y) \text{ mod order } \mathcal{G})$   
**for**  $y \ e \ w'$

**by** (*metis add.commute nat-pow-pow nat-pow-mult pow-generator-mod generator-closed mod-mult-right-eq*)

**ultimately show** *?thesis*

**unfolding** *chaum-ped-sigma.completeness-def chaum-ped-sigma.completeness-game-def*

**by**(*auto simp add: R-def challenge-space-def init-def check-def response-def split-def bind-spmf-const*)

**qed**

**lemma** *hzk-xr'-rewrite*:

**assumes**  $r: r < \text{order } \mathcal{G}$

**shows**  $((w*c + r) \text{ mod } (\text{order } \mathcal{G}) \text{ mod } (\text{order } \mathcal{G}) + (\text{order } \mathcal{G}) * w*c - w*c) \text{ mod } (\text{order } \mathcal{G}) = r$

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

**proof** –

**have** *?lhs* =  $(w*c + r + (\text{order } \mathcal{G}) * w*c - w*c) \text{ mod } (\text{order } \mathcal{G})$

**by** (*metis Nat.add-diff-assoc Num.of-nat-simps(1) One-nat-def add-less-same-cancel2 less-imp-le-nat*)

*mod-add-left-eq mult.assoc mult-0-right n-less-m-mult-n nat-neq-iff not-add-less2 of-nat-0-le-iff prime-gt-1-nat prime-order*)

**thus** *?thesis using r*

by (metis ab-semigroup-add-class.add-ac(1) ab-semigroup-mult-class.mult-ac(1)  
diff-add-inverse mod-if mod-mult-self2)

qed

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(cases  $w = 0$ )

case True

then show ?thesis using assms by simp

next

case  $w\text{-gt-}0$ : False

then show ?thesis

proof-

have  $(z + \text{order } \mathcal{G} * w * c - w * c) = (z + (\text{order } \mathcal{G} * w * c - w * c))$   
using  $z$  by (simp add: less-imp-le-nat mult-le-mono)

then have lhs: ?lhs =  $\mathbf{g} [\uparrow] z \otimes \mathbf{g} [\uparrow] ((\text{order } \mathcal{G}) * w * c - w * c)$

by (simp add: nat-pow-mult)

have  $\mathbf{g} [\uparrow] ((\text{order } \mathcal{G}) * w * c - w * c) = \text{inv} (h [\uparrow] c)$

proof(cases  $c = 0$ )

case True

then show ?thesis using lhs by simp

next

case False

hence \*:  $((\text{order } \mathcal{G}) * w * c - w * c) > 0$  using assms  $w\text{-gt-}0$

using gr0I mult-less-cancel2 n-less-m-mult-n numeral-nat(7) prime-gt-1-nat  
prime-order zero-less-diff by presburger

then have  $\mathbf{g} [\uparrow] ((\text{order } \mathcal{G}) * w * c - w * c) = \mathbf{g} [\uparrow] \text{int} ((\text{order } \mathcal{G}) * w * c - w * c)$   
by (simp add: int-pow-int)

also have ... =  $\mathbf{g} [\uparrow] \text{int} ((\text{order } \mathcal{G}) * w * c) \otimes \text{inv} (\mathbf{g} [\uparrow] (w * c))$

using int-pow-diff[of  $\mathbf{g}$  order  $\mathcal{G} * w * c w * c$ ] \* generator-closed int-ops(6)

int-pow-neg int-pow-neg-int by presburger

also have ... =  $\mathbf{g} [\uparrow] ((\text{order } \mathcal{G}) * w * c) \otimes \text{inv} (\mathbf{g} [\uparrow] (w * c))$   
by (metis int-pow-int)

also have ... =  $\mathbf{g} [\uparrow] ((\text{order } \mathcal{G}) * w * c) \otimes \text{inv} ((\mathbf{g} [\uparrow] w) [\uparrow] c)$   
by (simp add: nat-pow-pow)

also have ... =  $\mathbf{g} [\uparrow] ((\text{order } \mathcal{G}) * w * c) \otimes \text{inv} (h [\uparrow] c)$

using assms by simp

also have ... =  $\mathbf{1} \otimes \text{inv} (h [\uparrow] c)$

using generator-pow-order

by (metis generator-closed mult-is-0 nat-pow-0 nat-pow-pow)

ultimately show ?thesis

by (simp add: assms(1))

qed

then show ?thesis using lhs by simp

qed



qed

lemma *hvk-h-sub2-rewrite*:

assumes  $h' = g' [\wedge] w$

and  $z: z < \text{order } \mathcal{G}$

shows  $g' [\wedge] ((z + (\text{order } \mathcal{G}) * w * c - w * c)) = g' [\wedge] z \otimes \text{inv } (h' [\wedge] c)$   
(is ?lhs = ?rhs)

proof(cases  $w = 0$ )

case *True*

then show ?thesis

using *assms* by (simp add: *g'-def*)

next

case *w-gt-0: False*

then show ?thesis

proof-

have  $g' = \mathbf{g} [\wedge] x$  using *g'-def* by *simp*

have *g'-carrier*:  $g' \in \text{carrier } \mathcal{G}$  using *g'-def* by *simp*

have  $1: g' [\wedge] ((\text{order } \mathcal{G}) * w * c - w * c) = \text{inv } (h' [\wedge] c)$

proof(cases  $c = 0$ )

case *True*

then show ?thesis by *simp*

next

case *False*

hence \*:  $((\text{order } \mathcal{G}) * w * c - w * c) > 0$

using *assms mult-strict-mono w-gt-0 prime-gt-1-nat prime-order* by *auto*

then have  $g' [\wedge] ((\text{order } \mathcal{G}) * w * c - w * c) = g' [\wedge] (\text{int } (\text{order } \mathcal{G} * w * c) - \text{int } (w * c))$

by (*metis int-ops(6) int-pow-int of-nat-0-less-iff order.irrefl*)

also have  $\dots = g' [\wedge] ((\text{order } \mathcal{G}) * w * c) \otimes \text{inv } (g' [\wedge] (w * c))$

by (*metis g'-carrier int-pow-diff int-pow-int*)

also have  $\dots = g' [\wedge] ((\text{order } \mathcal{G}) * w * c) \otimes \text{inv } (h' [\wedge] c)$

by(*simp add: nat-pow-pow assms*)

also have  $\dots = \mathbf{1} \otimes \text{inv } (h' [\wedge] c)$

by (*metis g'-carrier nat-pow-one nat-pow-pow pow-order-eq-1*)

ultimately show ?thesis

by (*simp add: assms(1)*)

qed

have  $(z + \text{order } \mathcal{G} * w * c - w * c) = (z + (\text{order } \mathcal{G} * w * c - w * c))$

using *z* by (*simp add: less-imp-le-nat mult-le-mono*)

then have *lhs*:  $?lhs = g' [\wedge] z \otimes g' [\wedge] ((\text{order } \mathcal{G}) * w * c - w * c)$

by(*auto simp add: nat-pow-mult*)

then show ?thesis using *1* by *simp*

qed

qed

lemma *hv-zk2*:

assumes  $(H, w) \in R$

shows *chaum-ped-sigma.R H w c = chaum-ped-sigma.S H c*

including *monad-normalisation*

```

proof–
  have  $H: H = (\mathbf{g} [\wedge] (w::\text{nat}), g' [\wedge] w)$ 
    using assms R-def by(simp add: prod.expand)
  have  $g'\text{-carrier}: g' \in \text{carrier } \mathcal{G}$  using  $g'\text{-def}$  by simp
  have  $\text{chaum-ped-sigma.R } H \ w \ c = \text{do}$  {
     $\text{let } (h, h') = H;$ 
     $r \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$ 
     $\text{let } z = (w*c + r) \bmod (\text{order } \mathcal{G});$ 
     $\text{let } a = \mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{let } a' = g' [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{return-spmf } ((a, a'), c, z)$ 
    apply(simp add: chaum-ped-sigma.R-def Let-def response-def split-def init-def)
    using assms hvzk-xr'-rewrite
    by(simp cong: bind-spmf-cong-simp)
  }
  also have  $\dots = \text{do}$  {
     $\text{let } (h, h') = H;$ 
     $z \leftarrow \text{map-spmf } (\lambda r. (w*c + r) \bmod (\text{order } \mathcal{G})) (\text{sample-uniform } (\text{order } \mathcal{G}));$ 
     $\text{let } a = \mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{let } a' = g' [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{return-spmf } ((a, a'), c, z)$ 
    by(simp add: bind-map-spmf Let-def o-def)
  }
  also have  $\dots = \text{do}$  {
     $\text{let } (h, h') = H;$ 
     $z \leftarrow (\text{sample-uniform } (\text{order } \mathcal{G}));$ 
     $\text{let } a = \mathbf{g} [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{let } a' = g' [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{return-spmf } ((a, a'), c, z)$ 
    by(simp add: samp-uni-plus-one-time-pad)
  }
  also have  $\dots = \text{do}$  {
     $\text{let } (h, h') = H;$ 
     $z \leftarrow (\text{sample-uniform } (\text{order } \mathcal{G}));$ 
     $\text{let } a = \mathbf{g} [\wedge] z \otimes \text{inv } (h [\wedge] c);$ 
     $\text{let } a' = g' [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c) \bmod (\text{order } \mathcal{G}));$ 
     $\text{return-spmf } ((a, a'), c, z)$ 
    using hvzk-h-sub-rewrite assms
    apply(simp add: Let-def H)
    apply(intro bind-spmf-cong[OF refl]; clarsimp?)
    by (simp add: pow-generator-mod)
  }
  also have  $\dots = \text{do}$  {
     $\text{let } (h, h') = H;$ 
     $z \leftarrow (\text{sample-uniform } (\text{order } \mathcal{G}));$ 
     $\text{let } a = \mathbf{g} [\wedge] z \otimes \text{inv } (h [\wedge] c);$ 
     $\text{let } a' = g' [\wedge] ((z + (\text{order } \mathcal{G}) * w*c - w*c));$ 
     $\text{return-spmf } ((a, a'), c, z)$ 
    using  $g'\text{-carrier pow-carrier-mod}[of g']$  by simp
  }
  also have  $\dots = \text{do}$  {
     $\text{let } (h, h') = H;$ 
     $z \leftarrow (\text{sample-uniform } (\text{order } \mathcal{G}));$ 
     $\text{let } a = \mathbf{g} [\wedge] z \otimes \text{inv } (h [\wedge] c);$ 
  }

```

$let\ a' = g' [\ ] z \otimes inv\ (h' [\ ] c);$   
 $return\-spmf\ ((a, a'), c, z)$   
**using** *hvzk-h-sub2-rewrite* *assms H*  
**by**(*simp cong: bind-spmf-cong-simp*)  
**ultimately show** *?thesis*  
**unfolding** *chaum-ped-sigma.S-def chaum-ped-sigma.R-def*  
**by**(*simp add: init-def S2-def split-def Let-def  $\Sigma$ -protocols-base.S-def bind-map-spmf*  
*map-spmf-conv-bind-spmf*)  
**qed**

**lemma HVZK:**  
**shows** *chaum-ped-sigma.HVZK*  
**unfolding** *chaum-ped-sigma.HVZK-def*  
**by**(*auto simp add: hv-zk2 R-def valid-pub-def S2-def check-def cyclic-group-assoc*)

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

**proof** –

**have** *gcd: gcd (e - e') (order  $\mathcal{G}$ ) = 1*  
**using** *e e' prime-field prime-order* **by** *simp*  
**have** *a = g [ $\ ]$  z  $\otimes$  inv (fst h [ $\ ]$  e)*  
**using** *assms*  
**by** (*simp add: assms inv-solve-right*)  
**moreover have** *a = g [ $\ ]$  z'  $\otimes$  inv (fst h [ $\ ]$  e')*  
**using** *assms*  
**by** (*simp add: assms inv-solve-right*)  
**ultimately have** *g [ $\ ]$  z  $\otimes$  fst h [ $\ ]$  e' = g [ $\ ]$  z'  $\otimes$  fst h [ $\ ]$  e*  
**by** (*metis (no-types, lifting) assms cyclic-group-assoc cyclic-group-commute*  
*nat-pow-closed*)  
**moreover obtain** *t :: nat where t: fst h = g [ $\ ]$  t*  
**using** *assms generatorE* **by** *blast*  
**ultimately have** *g [ $\ ]$  (z + t \* e') = g [ $\ ]$  (z' + t \* e)*  
**using** *nat-pow-pow*  
**by** (*simp add: nat-pow-mult*)  
**hence** *[z + t \* e' = z' + t \* e] (mod order  $\mathcal{G}$ )*  
**using** *group-eq-pow-eq-mod or-gt-0* **by** *blast*  
**hence** *[int z + int t \* int e' = int z' + int t \* int e] (mod order  $\mathcal{G}$ )*  
**using** *cong-int-iff* **by** *force*  
**hence** *[int z - int z' = int t \* int e - int t \* int e'] (mod order  $\mathcal{G}$ )*  
**by** (*smt cong-diff-iff-cong-0*)  
**hence** *[int z - int z' = int t \* (int e - int e')] (mod order  $\mathcal{G}$ )*  
**by** (*simp add: right-diff-distrib*)

**hence**  $[int\ z - int\ z' = int\ t * (e - e')] \text{ (mod order } \mathcal{G})$   
**using** *assms by (simp add: of-nat-diff)*  
**hence**  $[(int\ z - int\ z') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})) = int\ t * (e - e') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G}))] \text{ (mod order } \mathcal{G})$   
**using** *cong-scalar-right by blast*  
**hence**  $[(int\ z - int\ z') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})) = int\ t * ((e - e') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})))] \text{ (mod order } \mathcal{G})$   
**by** *(simp add: more-arith-simps(11))*  
**hence**  $[(int\ z - int\ z') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})) = int\ t * 1] \text{ (mod order } \mathcal{G})$   
**by** *(metis (no-types, opaque-lifting) cong-scalar-left cong-trans inverse gcd)*  
**hence**  $[(int\ z - int\ z') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})) \text{ mod order } \mathcal{G} = t] \text{ (mod order } \mathcal{G})$   
**by** *simp*  
**hence**  $[nat\ ((int\ z - int\ z') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})) \text{ mod order } \mathcal{G}) = t] \text{ (mod order } \mathcal{G})$   
**by** *(metis cong-def int-ops(9) mod-mod-trivial nat-int)*  
**hence**  $\mathbf{g}\ [\wedge] \text{ (nat } ((int\ z - int\ z') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G})) \text{ mod order } \mathcal{G})) = \mathbf{g}\ [\wedge] t$   
**using** *order-gt-0 order-gt-0-iff-finite pow-generator-eq-iff-cong by blast*  
**thus** *?thesis using t by simp*  
**qed**

**lemma** *ss-rewrite2:*

**assumes**  $fst\ h \in carrier\ \mathcal{G}$   
**and**  $snd\ h \in carrier\ \mathcal{G}$   
**and**  $a \in carrier\ \mathcal{G}$   
**and**  $b \in carrier\ \mathcal{G}$   
**and**  $e < order\ \mathcal{G}$   
**and**  $a \otimes fst\ h\ [\wedge] e = \mathbf{g}\ [\wedge] z$   
**and**  $b \otimes snd\ h\ [\wedge] e = \mathbf{g}'\ [\wedge] z$   
**and**  $e' < e$   
**and**  $a \otimes fst\ h\ [\wedge] e' = \mathbf{g}\ [\wedge] z'$   
**and**  $b \otimes snd\ h\ [\wedge] e' = \mathbf{g}'\ [\wedge] z'$   
**shows**  $snd\ h = \mathbf{g}'\ [\wedge] ((int\ z - int\ z') * inverse\ (e - e')\ (order\ \mathcal{G}) \text{ mod int } (order\ \mathcal{G}))$   
**proof**–  
**have**  $gcd: gcd\ (e - e')\ (order\ \mathcal{G}) = 1$   
**using** *prime-field assms prime-order by simp*  
**have**  $b = \mathbf{g}'\ [\wedge] z \otimes inv\ (snd\ h\ [\wedge] e)$   
**by** *(simp add: assms inv-solve-right)*  
**moreover** **have**  $b = \mathbf{g}'\ [\wedge] z' \otimes inv\ (snd\ h\ [\wedge] e')$   
**by** *(metis assms(2) assms(4) assms(10) g'-def generator-closed group.inv-solve-right' group-l-invI l-inv-ex nat-pow-closed)*  
**ultimately** **have**  $\mathbf{g}'\ [\wedge] z \otimes snd\ h\ [\wedge] e' = \mathbf{g}'\ [\wedge] z' \otimes snd\ h\ [\wedge] e$   
**by** *(metis (no-types, lifting) assms cyclic-group-assoc cyclic-group-commute nat-pow-closed)*  
**moreover** **obtain**  $t :: nat$  **where**  $t: snd\ h = \mathbf{g}\ [\wedge] t$   
**using** *assms(2) generatorE by blast*

**ultimately have**  $\mathbf{g} \ [\frown] \ (x * z + t * e') = \mathbf{g} \ [\frown] \ (x * z' + t * e)$   
**using** *g'-def nat-pow-pow*  
**by** (*simp add: nat-pow-mult*)  
**hence**  $[x * z + t * e' = x * z' + t * e] \ (\text{mod order } \mathcal{G})$   
**using** *group-eq-pow-eq-mod order-gt-0* **by** *blast*  
**hence**  $[int \ x * int \ z + int \ t * int \ e' = int \ x * int \ z' + int \ t * int \ e] \ (\text{mod order } \mathcal{G})$   
**by** (*metis Groups.add-ac(2) Groups.mult-ac(2) cong-int-iff int-ops(7) int-plus*)  
**hence**  $[int \ x * int \ z - int \ x * int \ z' = int \ t * int \ e - int \ t * int \ e'] \ (\text{mod order } \mathcal{G})$   
**by** (*smt cong-diff-iff-cong-0*)  
**hence**  $[int \ x * (int \ z - int \ z') = int \ t * (int \ e - int \ e')] \ (\text{mod order } \mathcal{G})$   
**by** (*simp add: int-distrib(4)*)  
**hence**  $[int \ x * (int \ z - int \ z') = int \ t * (e - e')] \ (\text{mod order } \mathcal{G})$   
**using** *assms* **by** (*simp add: of-nat-diff*)  
**hence**  $[(int \ x * (int \ z - int \ z')) * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G})) = int \ t * (e - e') * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G}))] \ (\text{mod order } \mathcal{G})$   
**using** *cong-scalar-right* **by** *blast*  
**hence**  $[(int \ x * (int \ z - int \ z')) * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G})) = int \ t * ((e - e') * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G})))] \ (\text{mod order } \mathcal{G})$   
**by** (*simp add: more-arith-simps(11)*)  
**hence**  $*: [(int \ x * (int \ z - int \ z')) * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G})) = int \ t * 1] \ (\text{mod order } \mathcal{G})$   
**by** (*metis (no-types, opaque-lifting) cong-scalar-left cong-trans gcd inverse*)  
**hence**  $[nat \ ((int \ x * (int \ z - int \ z')) * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G})) \ \text{mod order } \mathcal{G}) = t] \ (\text{mod order } \mathcal{G})$   
**by** (*metis cong-def cong-mod-right more-arith-simps(6) nat-int zmod-int*)  
**hence**  $\mathbf{g} \ [\frown] \ (nat \ ((int \ x * (int \ z - int \ z')) * fst \ (bezw \ (e - e') \ (\text{order } \mathcal{G})) \ \text{mod order } \mathcal{G})) = \mathbf{g} \ [\frown] \ t$   
**using** *order-gt-0 order-gt-0-iff-finite pow-generator-eq-iff-cong* **by** *blast*  
**thus** *?thesis* **using** *t*  
**by** (*metis (mono-tags, opaque-lifting) \* cong-def g'-def generator-closed int-pow-int int-pow-pow mod-mult-right-eq more-arith-simps(11) more-arith-simps(6) pow-generator-mod-int*)  
**qed**

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

**assumes** *e-e'-mod: e' mod order G < e mod order G*  
**and** *h-mem: snd h ∈ carrier G*  
**and** *a-mem: snd a ∈ carrier G*  
**and** *a1: snd a ⊗ snd h [frown] e = g' [frown] z*  
**and** *a2: snd a ⊗ snd h [frown] e' = g' [frown] z'*  
**shows**  $snd \ h = g' \ [\frown] \ ((int \ z - int \ z') * fst \ (bezw \ ((e \ \text{mod order } \mathcal{G}) - e' \ \text{mod order } \mathcal{G}) \ \text{mod int} \ (\text{order } \mathcal{G})))$   
**proof** –  
**have** *gcd: gcd ((e mod order G - e' mod order G) mod order G) (order G) = 1*  
**using** *prime-field*  
**by** (*simp add: assms less-imp-diff-less linorder-not-le prime-order*)  
**have**  $snd \ a = g' \ [\frown] \ z \ \otimes \ inv \ (snd \ h \ [\frown] \ e)$   
**using** *a1*

**by** (*metis* (*no-types*, *lifting*) *Group.group.axioms(1)* *h-mem a-mem group.inv-closed* *group-l-invI l-inv-ex monoid.m-assoc nat-pow-closed r-inv r-one*)  
**moreover have**  $\text{snd } a = g' [\wedge] z' \otimes \text{inv } (\text{snd } h [\wedge] e')$   
**by** (*metis a2 h-mem a-mem g'-def generator-closed group.inv-solve-right' group-l-invI* *l-inv-ex nat-pow-closed*)  
**ultimately have**  $g' [\wedge] z \otimes \text{snd } h [\wedge] e' = g' [\wedge] z' \otimes \text{snd } h [\wedge] e$   
**by** (*metis* (*no-types*, *lifting*) *a2 h-mem a-mem a1 cyclic-group-assoc cyclic-group-commute* *nat-pow-closed*)  
**moreover obtain**  $t :: \text{nat}$  **where**  $t: \text{snd } h = \mathbf{g} [\wedge] t$   
**using** *assms(2) generatorE* **by** *blast*  
**ultimately have**  $\mathbf{g} [\wedge] (x * z + t * e') = \mathbf{g} [\wedge] (x * z' + t * e)$   
**using** *g'-def nat-pow-pow*  
**by** (*simp add: nat-pow-mult*)  
**hence**  $[x * z + t * e' = x * z' + t * e] \text{ (mod order } \mathcal{G})$   
**using** *group-eq-pow-eq-mod order-gt-0* **by** *blast*  
**hence**  $[\text{int } x * \text{int } z + \text{int } t * \text{int } e' = \text{int } x * \text{int } z' + \text{int } t * \text{int } e] \text{ (mod order } \mathcal{G})$   
**by** (*metis* *Groups.add-ac(2) Groups.mult-ac(2) cong-int-iff int-ops(7) int-plus*)  
**hence**  $[\text{int } x * \text{int } z - \text{int } x * \text{int } z' = \text{int } t * \text{int } e - \text{int } t * \text{int } e'] \text{ (mod order } \mathcal{G})$   
**by** (*smt cong-diff-iff-cong-0*)  
**hence**  $[\text{int } x * (\text{int } z - \text{int } z') = \text{int } t * (\text{int } e - \text{int } e')] \text{ (mod order } \mathcal{G})$   
**by** (*simp add: int-distrib(4)*)  
**hence**  $[\text{int } x * (\text{int } z - \text{int } z') = \text{int } t * (\text{int } e \text{ mod order } \mathcal{G} - \text{int } e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}] \text{ (mod order } \mathcal{G})$   
**by** (*metis* (*no-types*, *lifting*) *cong-def mod-diff-eq mod-mod-trivial mod-mult-right-eq*)  
**hence**  $*: [\text{int } x * (\text{int } z - \text{int } z') = \text{int } t * (e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}] \text{ (mod order } \mathcal{G})$   
**by** (*simp add: assms(1) int-ops(9) less-imp-le-nat of-nat-diff*)  
**hence**  $[\text{int } x * (\text{int } z - \text{int } z') * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}))$   
 $= \text{int } t * ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G})$   
 $* \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G})) \text{ (order } \mathcal{G})] \text{ (mod order } \mathcal{G})$   
**by** (*metis* (*no-types*, *lifting*) *cong-mod-right cong-scalar-right less-imp-diff-less* *mod-if more-arith-simps(11) or-gt-0 unique-euclidean-semiring-numeral-class.pos-mod-bound*)  
**hence**  $[\text{int } x * (\text{int } z - \text{int } z') * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}))$   
 $= \text{int } t * 1] \text{ (mod order } \mathcal{G})$   
**by** (*meson Number-Theory-Aux.inverse \* gcd cong-scalar-left cong-trans*)  
**hence**  $\mathbf{g} [\wedge] (\text{int } x * (\text{int } z - \text{int } z') * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G})) \text{ (order } \mathcal{G})) = \mathbf{g} [\wedge] t$   
**by** (*metis cong-def int-pow-int more-arith-simps(6) pow-generator-mod-int*)  
**thus** *?thesis* **using**  $t$   
**by** (*metis* (*mono-tags*, *opaque-lifting*) *g'-def generator-closed int-pow-int int-pow-pow* *mod-mult-right-eq more-arith-simps(11) pow-generator-mod-int*)  
**qed**

**lemma** *special-soundness*:

**shows** *chaum-ped-sigma.special-soundness*  
**unfolding** *chaum-ped-sigma.special-soundness-def*  
**apply**(*auto simp add: challenge-space-def check-def ss-adversary-def R-def valid-pub-def*)  
**using** *ss-rewrite2 ss-rewrite1 by auto*

**theorem**  $\Sigma$ -protocol: *chaum-ped-sigma. $\Sigma$ -protocol*  
**by**(*simp add: chaum-ped-sigma. $\Sigma$ -protocol-def completeness HVZK special-soundness*)

**sublocale** *chaum-ped- $\Sigma$ -commit:  $\Sigma$ -protocols-to-commitments init response check*  
*R S2 ss-adversary challenge-space valid-pub G*  
**apply** *unfold-locales*  
**apply**(*auto simp add:  $\Sigma$ -protocol lossless-init lossless-response lossless-G*)  
**by**(*simp add: R-def G-def*)

**sublocale** *dis-log: dis-log  $\mathcal{G}$*   
**unfolding** *dis-log-def by simp*

**sublocale** *dis-log-alt: dis-log-alt  $\mathcal{G}$  x*  
**unfolding** *dis-log-alt-def by simp*

**lemma** *reduction-to-dis-log:*  
**shows** *chaum-ped- $\Sigma$ -commit.rel-advantage  $\mathcal{A} = \text{dis-log.}advantage (\text{dis-log-alt.adversary3}$*   
 *$\mathcal{A}$ )*  
**proof** –  
**have** *chaum-ped- $\Sigma$ -commit.rel-game  $\mathcal{A} = \text{TRY do}$*  {  
*w  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*let (h,w) = ((**g** [ $\wedge$ ] w, **g'** [ $\wedge$ ] w), w);*  
*w'  $\leftarrow$   $\mathcal{A}$  h;*  
*return-**spmf** ((fst h = **g** [ $\wedge$ ] w'  $\wedge$  snd h = **g'** [ $\wedge$ ] w'))} ELSE return-**spmf** False*  
**unfolding** *chaum-ped- $\Sigma$ -commit.rel-game-def*  
**by**(*simp add: G-def R-def*)  
**also have** ... = *TRY do* {  
*w  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*let (h,w) = ((**g** [ $\wedge$ ] w, **g'** [ $\wedge$ ] w), w);*  
*w'  $\leftarrow$   $\mathcal{A}$  h;*  
*return-**spmf** ([w = w'] (mod (order  $\mathcal{G}$ ))  $\wedge$  [x\*w = x\*w'] (mod order  $\mathcal{G}$ ))} ELSE*  
*return-**spmf** False*  
**apply**(*intro try-**spmf**-cong bind-**spmf**-cong[OF refl]; simp add: dis-log-alt.dis-log3-def*  
*dis-log-alt.g'-def g'-def*)  
**by** (*simp add: finite-carrier nat-pow-pow pow-generator-eq-iff-cong*)  
**also have** ... = *dis-log-alt.dis-log3  $\mathcal{A}$*   
**apply**(*auto simp add: dis-log-alt.dis-log3-def dis-log-alt.g'-def g'-def*)  
**by**(*intro try-**spmf**-cong bind-**spmf**-cong[OF refl]; clarsimp?; auto simp add:*  
*cong-scalar-left*)  
**ultimately have** *chaum-ped- $\Sigma$ -commit.rel-advantage  $\mathcal{A} = \text{dis-log-alt.}advantage3$*   
 *$\mathcal{A}$*   
**by**(*simp add: chaum-ped- $\Sigma$ -commit.rel-advantage-def dis-log-alt.advantage3-def*)  
**thus** *?thesis*  
**by** (*simp add: dis-log-alt-reductions.dis-log-adv3 cyclic-group-axioms dis-log-alt.dis-log-alt-axioms*)

*dis-log-alt-reductions.intro*)  
**qed**

**lemma** *commitment-correct: chaum-ped- $\Sigma$ -commit.abstract-com.correct*  
**by**(*simp add: chaum-ped- $\Sigma$ -commit.commit-correct*)

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

**lemma** *binding: chaum-ped- $\Sigma$ -commit.abstract-com.bind-advantage  $\mathcal{A} \leq$  dis-log.advantage*  
*(dis-log-alt.adversary3 ((chaum-ped- $\Sigma$ -commit.adversary  $\mathcal{A}$ )))*  
**using** *chaum-ped- $\Sigma$ -commit.bind-advantage reduction-to-dis-log* **by** *simp*

**end**

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

**sublocale** *chaum-ped- $\Sigma$   $\mathcal{G}$   $\eta$  for  $\eta$*   
**by**(*simp add: cp- $\Sigma$* )

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

**theorem** *sigma-protocol:*  
**shows** *chaum-ped-sigma. $\Sigma$ -protocol  $n$*   
**by**(*simp add:  $\Sigma$ -protocol*)

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  $n$*   
**using** *chaum-ped- $\Sigma$ -commit.commit-correct* **by** *simp*

**lemma** *asymp-perfect-hiding: chaum-ped- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa*  
 *$n (\mathcal{A} \ n)$*   
**using** *chaum-ped- $\Sigma$ -commit.perfect-hiding* **by** *blast*

**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 ( $\lambda \ n. \text{chaum-ped-}\Sigma\text{-commit.abstract-com.bind-advantage } n (\mathcal{A} \ n)$ )*  
**using** *chaum-ped- $\Sigma$ -commit.bind-advantage asms chaum-ped- $\Sigma$ -commit.abstract-com.bind-advantage-def negligible-le binding* **by** *auto*

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

**and**  $x :: nat$

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

**begin**

**definition**  $g' = g$  [ $\wedge$ ]  $x$

**lemma** *order-gt-1*: *order*  $\mathcal{G} > 1$

**using** *prime-order*

**using** *prime-gt-1-nat* **by** *blast*

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

**using** *order-gt-1* **by** *simp*

**definition** *response*  $r w e = do$  {

*let*  $(r1, r2) = r$ ;

*let*  $(x1, x2) = w$ ;

*let*  $z1 = (e * x1 + r1) \bmod (\text{order } \mathcal{G})$ ;

*let*  $z2 = (e * x2 + r2) \bmod (\text{order } \mathcal{G})$ ;

*return-spmf*  $((z1, z2))$ }

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

**by** (*simp add: response-def split-def*)

**type-synonym** *witness* =  $nat \times nat$

**type-synonym** *rand* =  $nat \times nat$

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

**type-synonym** *response* =  $(nat \times nat)$

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

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

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

**where** *init*  $y w = do$  {

*let*  $(x1, x2) = w$ ;

$r1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $r2 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $\text{return-spmf } ((r1, r2), \mathbf{g} [\uparrow] r1 \otimes g' [\uparrow] r2)$

**lemma** *lossless-init*: *lossless-spmf* (*init h w*)  
**by**(*simp add: init-def*)

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

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

**definition** *G* :: (*'grp pub-in*  $\times$  *witness*) *spmf*  
**where**  $G = \text{do } \{$   
 $w1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $w2 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $\text{return-spmf } (\mathbf{g} [\uparrow] w1 \otimes g' [\uparrow] w2, (w1, w2))\}$

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

**lemma** *lossless-G*: *lossless-spmf* *G*  
**by**(*simp add: G-def*)

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

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

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

$(\text{nat } ((\text{int } z1' - \text{int } z1) * \text{inverse } (e' - e) (\text{order } \mathcal{G}) \text{ mod order } \mathcal{G})),$   
 $\text{if } (e > e') \text{ then } (\text{nat } ((\text{int } z2 - \text{int } z2') * \text{inverse } (e - e') (\text{order } \mathcal{G}) \text{ mod order } \mathcal{G})) \text{ else}$   
 $(\text{nat } ((\text{int } z2' - \text{int } z2) * \text{inverse } (e' - e) (\text{order } \mathcal{G}) \text{ mod order } \mathcal{G}))\}$

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

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

**lemma** *g'-in-carrier* [*simp*]:  $g' \in \text{carrier } \mathcal{G}$   
**using** *g'-def* **by** *auto*

**sublocale**  $\Sigma$ -*protocols-base*:  $\Sigma$ -*protocols-base* *init response check R S2 ss-adversary challenge-space valid-pub*  
**by** *unfold-locale* (*auto simp add: R-def valid-pub-def*)

**lemma**  $\Sigma$ -*protocols-base.R*  $h w c = R2 h w c$   
**by** (*simp add: \Sigma-protocols-base.R-def R2-def; simp add: init-def split-def response-def*)

**lemma** *completeness*:

**shows**  $\Sigma$ -*protocols-base.completeness*

**proof** –

**have**  $(\mathbf{g} [\wedge] ((e * \text{fst } w' + y) \text{ mod order } \mathcal{G}) \otimes g' [\wedge] ((e * \text{snd } w' + ya) \text{ mod order } \mathcal{G})) = \mathbf{g} [\wedge] y \otimes g' [\wedge] ya \otimes (\mathbf{g} [\wedge] \text{fst } w' \otimes g' [\wedge] \text{snd } w') [\wedge] e)$

**for**  $e y ya :: \text{nat}$  **and**  $w' :: \text{nat} \times \text{nat}$

**proof** –

**have**  $\mathbf{g} [\wedge] ((e * \text{fst } w' + y) \text{ mod order } \mathcal{G}) \otimes g' [\wedge] ((e * \text{snd } w' + ya) \text{ mod order } \mathcal{G}) = \mathbf{g} [\wedge] ((y + e * \text{fst } w')) \otimes g' [\wedge] ((ya + e * \text{snd } w'))$

**by** (*simp add: cyclic-group.pow-carrier-mod cyclic-group-axioms g'-def add commute pow-generator-mod*)

**also have**  $\dots = \mathbf{g} [\wedge] y \otimes \mathbf{g} [\wedge] (e * \text{fst } w') \otimes g' [\wedge] ya \otimes g' [\wedge] (e * \text{snd } w')$

**by** (*simp add: g'-def m-assoc nat-pow-mult*)

**also have**  $\dots = \mathbf{g} [\wedge] y \otimes g' [\wedge] ya \otimes \mathbf{g} [\wedge] (e * \text{fst } w') \otimes g' [\wedge] (e * \text{snd } w')$

**by** (*smt add commute g'-def generator-closed m-assoc nat-pow-closed nat-pow-mult nat-pow-pow*)

**also have**  $\dots = \mathbf{g} [\wedge] y \otimes g' [\wedge] ya \otimes ((\mathbf{g} [\wedge] \text{fst } w') [\wedge] e \otimes (g' [\wedge] \text{snd } w') [\wedge] e)$

**by** (*simp add: m-assoc mult commute nat-pow-pow*)

**also have**  $\dots = \mathbf{g} [\wedge] y \otimes g' [\wedge] ya \otimes ((\mathbf{g} [\wedge] \text{fst } w' \otimes g' [\wedge] \text{snd } w') [\wedge] e)$

**by** (*smt power-distrib g'-def generator-closed mult commute nat-pow-closed nat-pow-mult nat-pow-pow*)

**ultimately show** *?thesis* **by** *simp*

**qed**

**thus** *?thesis*

**unfolding**  $\Sigma$ -*protocols-base.completeness-def*  $\Sigma$ -*protocols-base.completeness-game-def*

**by**(*simp add: R-def challenge-space-def init-def check-def response-def split-def bind-spmf-const*)

**qed**

**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**(*cases x1 = 0*)

**case** *True*

**then show** *?thesis* **using** *r1* **by** *simp*

**next**

**case** *x1-neq-0: False*

**have** *z1-eq*:  $[(r1 + c * x1) \bmod (\text{order } \mathcal{G}) + \text{order } \mathcal{G} * c * x1 = r1 + c * x1] \bmod (\text{order } \mathcal{G})$

**using** *gr-implies-not-zero order-gt-1*

**by** (*simp add: Groups.mult-ac(1) cong-def*)

**hence**  $[(r1 + c * x1) \bmod (\text{order } \mathcal{G}) + \text{order } \mathcal{G} * c * x1 - c * x1 = r1] \bmod (\text{order } \mathcal{G})$

**proof**(*cases c = 0*)

**case** *True*

**then show** *?thesis*

**using** *z1-eq* **by** *auto*

**next**

**case** *False*

**have**  $\text{order } \mathcal{G} * c * x1 - c * x1 > 0$  **using** *x1-neq-0 False*

**using** *prime-gt-1-nat prime-order* **by** *auto*

**thus** *?thesis*

**by** (*smt Groups.add-ac(2) add-diff-inverse-nat cong-add-lcancel-nat diff-is-0-eq le-simps(1) neq0-conv trans-less-add2 z1-eq zero-less-diff*)

**qed**

**thus** *?thesis*

**by** (*simp add: r1 cong-def*)

**qed**

**lemma** *hvzk-z1-r1-tuple-rewrite*:

**assumes** *r1*:  $r1 < \text{order } \mathcal{G}$

**shows**  $(\mathbf{g} [\ ] r1 \otimes \mathbf{g}' [\ ] r2, c, (r1 + c * x1) \bmod \text{order } \mathcal{G}, (r2 + c * x2) \bmod \text{order } \mathcal{G}) =$

$(\mathbf{g} [\ ] (((r1 + c * x1) \bmod \text{order } \mathcal{G}) + \text{order } \mathcal{G} * c * x1 - c * x1) \bmod \text{order } \mathcal{G})$

$\otimes \mathbf{g}' [\ ] r2, c, (r1 + c * x1) \bmod \text{order } \mathcal{G}, (r2 + c * x2) \bmod \text{order } \mathcal{G})$

**proof**–

**have**  $\mathbf{g} [\ ] r1 = \mathbf{g} [\ ] (((r1 + c * x1) \bmod \text{order } \mathcal{G}) + \text{order } \mathcal{G} * c * x1 - c * x1) \bmod \text{order } \mathcal{G})$

**using** *assms hvzk-z-r* **by** *simp*

**thus** *?thesis* **by** *argo*

**qed**

**lemma** *hvzk-z2-r2-tuple-rewrite:*

**assumes**  $xb < \text{order } \mathcal{G}$   
**shows**  $(\mathbf{g} [\wedge] (((x' + xa * x1) \text{ mod } \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x1 - xa * x1) \text{ mod } \text{order } \mathcal{G})$   
 $\otimes g' [\wedge] xb, xa, (x' + xa * x1) \text{ mod } \text{order } \mathcal{G}, (xb + xa * x2) \text{ mod } \text{order } \mathcal{G}) =$   
 $(\mathbf{g} [\wedge] (((x' + xa * x1) \text{ mod } \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x1 - xa * x1) \text{ mod } \text{order } \mathcal{G})$   
 $\otimes g' [\wedge] (((xb + xa * x2) \text{ mod } \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x2 - xa * x2) \text{ mod } \text{order } \mathcal{G}), xa, (x' + xa * x1) \text{ mod } \text{order } \mathcal{G}, (xb + xa * x2) \text{ mod } \text{order } \mathcal{G})$   
**proof** –  
**have**  $g' [\wedge] xb = g' [\wedge] (((xb + xa * x2) \text{ mod } \text{order } \mathcal{G} + \text{order } \mathcal{G} * xa * x2 - xa * x2) \text{ mod } \text{order } \mathcal{G})$   
**using** *hvzk-z-r assms by simp*  
**thus** *?thesis by argo*  
**qed**

**lemma** *hvzk-sim-inverse-rewrite:*

**assumes**  $h: h = \mathbf{g} [\wedge] (x1 :: \text{nat}) \otimes g' [\wedge] (x2 :: \text{nat})$   
**shows**  $\mathbf{g} [\wedge] (((z1 :: \text{nat}) + \text{order } \mathcal{G} * c * x1 - c * x1) \text{ mod } (\text{order } \mathcal{G}))$   
 $\otimes g' [\wedge] (((z2 :: \text{nat}) + \text{order } \mathcal{G} * c * x2 - c * x2) \text{ mod } (\text{order } \mathcal{G}))$   
 $= (\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2) \otimes (\text{inv } h [\wedge] c)$   
*(is ?lhs = ?rhs)*  
**proof** –  
**have** *in-carrier1*:  $(g' [\wedge] x2) [\wedge] c \in \text{carrier } \mathcal{G}$  **by** *simp*  
**have** *in-carrier2*:  $(\mathbf{g} [\wedge] x1) [\wedge] c \in \text{carrier } \mathcal{G}$  **by** *simp*  
**have** *pow-distrib1*:  $\text{order } \mathcal{G} * c * x1 - c * x1 = (\text{order } \mathcal{G} - 1) * c * x1$   
**and** *pow-distrib2*:  $\text{order } \mathcal{G} * c * x2 - c * x2 = (\text{order } \mathcal{G} - 1) * c * x2$   
**using** *assms by (simp add: diff-mult-distrib)+*  
**have** *?lhs*  $= \mathbf{g} [\wedge] (z1 + \text{order } \mathcal{G} * c * x1 - c * x1) \otimes g' [\wedge] (z2 + \text{order } \mathcal{G} * c * x2 - c * x2)$   
**by** *(simp add: pow-carrier-mod)*  
**also have**  $\dots = \mathbf{g} [\wedge] (z1 + (\text{order } \mathcal{G} * c * x1 - c * x1)) \otimes g' [\wedge] (z2 + (\text{order } \mathcal{G} * c * x2 - c * x2))$   
**using** *h*  
**by** *(smt Nat.add-diff-assoc diff-zero le-simps(1) nat-0-less-mult-iff neq0-conv pow-distrib1 pow-distrib2 prime-gt-1-nat prime-order zero-less-diff)*  
**also have**  $\dots = \mathbf{g} [\wedge] z1 \otimes \mathbf{g} [\wedge] (\text{order } \mathcal{G} * c * x1 - c * x1) \otimes g' [\wedge] z2 \otimes g' [\wedge] (\text{order } \mathcal{G} * c * x2 - c * x2)$   
**using** *nat-pow-mult*  
**by** *(simp add: m-assoc)*  
**also have**  $\dots = \mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes \mathbf{g} [\wedge] (\text{order } \mathcal{G} * c * x1 - c * x1) \otimes g' [\wedge] (\text{order } \mathcal{G} * c * x2 - c * x2)$   
**by** *(smt add.commute g'-def generator-closed m-assoc nat-pow-closed nat-pow-mult nat-pow-pow)*  
**also have**  $\dots = \mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes \mathbf{g} [\wedge] ((\text{order } \mathcal{G} - 1) * c * x1) \otimes g' [\wedge] ((\text{order } \mathcal{G} - 1) * c * x2)$   
**using** *pow-distrib1 pow-distrib2 by argo*

**also have** ... =  $\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes (\mathbf{g} [\wedge] (\text{order } \mathcal{G} - 1)) [\wedge] (c * x1) \otimes (g' [\wedge] ((\text{order } \mathcal{G} - 1))) [\wedge] (c * x2)$   
**by** (*simp add: more-arith-simps(11) nat-pow-pow*)  
**also have** ... =  $\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes (\text{inv } (\mathbf{g} [\wedge] c)) [\wedge] x1 \otimes (\text{inv } (g' [\wedge] c)) [\wedge] x2$   
**using** *assms neg-power-inverse inverse-pow-pow nat-pow-pow prime-gt-1-nat prime-order* **by** *auto*  
**also have** ... =  $\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes (\text{inv } ((\mathbf{g} [\wedge] c) [\wedge] x1)) \otimes (\text{inv } ((g' [\wedge] c) [\wedge] x2))$   
**by** (*simp add: inverse-pow-pow*)  
**also have** ... =  $\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes ((\text{inv } ((\mathbf{g} [\wedge] x1) [\wedge] c)) \otimes (\text{inv } ((g' [\wedge] x2) [\wedge] c)))$   
**by** (*simp add: mult.commute cyclic-group-assoc nat-pow-pow*)  
**also have** ... =  $\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes \text{inv } ((\mathbf{g} [\wedge] x1) [\wedge] c \otimes (g' [\wedge] x2) [\wedge] c)$   
**using** *inverse-split in-carrier2 in-carrier1* **by** *simp*  
**also have** ... =  $\mathbf{g} [\wedge] z1 \otimes g' [\wedge] z2 \otimes \text{inv } (h [\wedge] c)$   
**using** *h cyclic-group-commute monoid-comm-monoidI*  
**by** (*simp add: pow-mult-distrib*)  
**ultimately show** *?thesis*  
**by** (*simp add: h inverse-pow-pow*)  
**qed**

**lemma** *hv-zk*:

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

**proof** –

**have**  $\Sigma\text{-protocols-base.R } h (x1, x2) c = \text{do } \{$   
 $r1 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $r2 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $\text{let } z1 = (r1 + c * x1) \text{ mod } (\text{order } \mathcal{G});$   
 $\text{let } z2 = (r2 + c * x2) \text{ mod } (\text{order } \mathcal{G});$   
 $\text{return-spmf } (\mathbf{g} [\wedge] r1 \otimes g' [\wedge] r2, c, (z1, z2))\}$   
**by** (*simp add:  $\Sigma\text{-protocols-base.R-def R2-def}$ ; simp add: add.commute init-def split-def response-def*)  
**also have** ... =  $\text{do } \{$   
 $r2 \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});$   
 $z1 \leftarrow \text{map-spmf } (\lambda r1. (r1 + c * x1) \text{ mod } (\text{order } \mathcal{G})) (\text{sample-uniform } (\text{order } \mathcal{G}));$   
 $\text{let } z2 = (r2 + c * x2) \text{ mod } (\text{order } \mathcal{G});$   
 $\text{return-spmf } (\mathbf{g} [\wedge] ((z1 + \text{order } \mathcal{G} * c * x1 - c * x1) \text{ mod } (\text{order } \mathcal{G})) \otimes g' [\wedge] r2, c, (z1, z2))\}$   
**by** (*simp add: bind-map-spmf o-def Let-def hvzk-z1-r1-tuple-rewrite assms cong: bind-spmf-cong-simp*)  
**also have** ... =  $\text{do } \{$   
 $z1 \leftarrow \text{map-spmf } (\lambda r1. (r1 + c * x1) \text{ mod } (\text{order } \mathcal{G})) (\text{sample-uniform } (\text{order } \mathcal{G}));$   
 $z2 \leftarrow \text{map-spmf } (\lambda r2. (r2 + c * x2) \text{ mod } (\text{order } \mathcal{G})) (\text{sample-uniform } (\text{order } \mathcal{G}));$

```

    return-spmf (g [⌈] ((z1 + order G * c * x1 - c * x1) mod (order G)) ⊗ g' [⌈]
((z2 + order G * c * x2 - c * x2) mod (order G)) ,c,(z1,z2)))
  by(simp add: bind-map-spmf o-def Let-def hvzk-z2-r2-tuple-rewrite cong: bind-spmf-cong-simp)
  also have ... = do {
    z1 ← map-spmf (λ r1. (c * x1 + r1) mod (order G)) (sample-uniform (order
G));
    z2 ← map-spmf (λ r2. (c * x2 + r2) mod (order G)) (sample-uniform (order
G));
    return-spmf (g [⌈] ((z1 + order G * c * x1 - c * x1) mod (order G)) ⊗ g' [⌈]
((z2 + order G * c * x2 - c * x2) mod (order G)) ,c,(z1,z2)))
  by(simp add: add.commute)
  also have ... = do {
    z1 ← (sample-uniform (order G));
    z2 ← (sample-uniform (order G));
    return-spmf (g [⌈] ((z1 + order G * c * x1 - c * x1) mod (order G)) ⊗ g' [⌈]
((z2 + order G * c * x2 - c * x2) mod (order G)) ,c,(z1,z2)))
  by(simp add: samp-uni-plus-one-time-pad)
  also have ... = do {
    z1 ← (sample-uniform (order G));
    z2 ← (sample-uniform (order G));
    return-spmf ((g [⌈] z1 ⊗ g' [⌈] z2) ⊗ (inv h [⌈] c) ,c,(z1,z2)))
  by(simp add: hvzk-sim-inverse-rewrite assms cong: bind-spmf-cong-simp)
  ultimately show ?thesis
  by(simp add: Σ-protocols-base.S-def S2-def bind-map-spmf map-spmf-conv-bind-spmf)
qed

```

**lemma HVZK:**

```

  shows Σ-protocols-base.HVZK
  unfolding Σ-protocols-base.HVZK-def
  apply(auto simp add: R-def challenge-space-def hv-zk S2-def check-def valid-pub-def)
  by (metis (no-types, lifting) cyclic-group-commute g'-in-carrier generator-closed
inv-closed inv-solve-left inverse-pow-pow m-closed nat-pow-closed)

```

**lemma ss-rewrite:**

```

  assumes h ∈ carrier G
  and a ∈ carrier G
  and e < order G
  and g [⌈] z1 ⊗ g' [⌈] z1' = a ⊗ h [⌈] e
  and e' < e
  and g [⌈] z2 ⊗ g' [⌈] z2' = a ⊗ h [⌈] e'
  shows h = g [⌈] ((int z1 - int z2) * fst (bezw (e - e') (order G)) mod int (order
G)) ⊗ g' [⌈] ((int z1' - int z2') * fst (bezw (e - e') (order G)) mod int (order G))
  proof -
    have gcd: gcd (e - e') (order G) = 1
      using prime-field assms prime-order by simp
    have g [⌈] z1 ⊗ g' [⌈] z1' ⊗ inv (h [⌈] e) = a
      by (simp add: inv-solve-right' assms)
    moreover have g [⌈] z2 ⊗ g' [⌈] z2' ⊗ inv (h [⌈] e') = a
      by (simp add: assms inv-solve-right')
  qed

```

ultimately have  $\mathbf{g} [\ulcorner] z2 \otimes g' [\ulcorner] z2' \otimes \text{inv} (h [\ulcorner] e') = \mathbf{g} [\ulcorner] z1 \otimes g' [\ulcorner] z1' \otimes \text{inv} (h [\ulcorner] e)$   
 using *g'-def* by (*simp add: nat-pow-pow*)  
 moreover obtain  $t :: \text{nat}$  where  $t: h = \mathbf{g} [\ulcorner] t$   
 using *assms generatorE* by *blast*  
 ultimately have  $\mathbf{g} [\ulcorner] z2 \otimes \mathbf{g} [\ulcorner] (x * z2') \otimes \mathbf{g} [\ulcorner] (t * e) = \mathbf{g} [\ulcorner] z1 \otimes \mathbf{g} [\ulcorner] (x * z1') \otimes (\mathbf{g} [\ulcorner] (t * e'))$   
 using *assms(2) assms(4) cyclic-group-commute m-assoc g'-def nat-pow-pow* by *auto*  
 hence  $\mathbf{g} [\ulcorner] (z2 + x * z2' + t * e) = \mathbf{g} [\ulcorner] (z1 + x * z1' + t * e')$   
 by (*simp add: nat-pow-mult*)  
 hence  $[z2 + x * z2' + t * e = z1 + x * z1' + t * e'] \text{ (mod order } \mathcal{G})$   
 using *group-eq-pow-eq-mod order-gt-0* by *blast*  
 hence  $[\text{int } z2 + \text{int } x * \text{int } z2' + \text{int } t * \text{int } e = \text{int } z1 + \text{int } x * \text{int } z1' + \text{int } t * \text{int } e'] \text{ (mod order } \mathcal{G})$   
 using *cong-int-iff* by *force*  
 hence  $[\text{int } z1 + \text{int } x * \text{int } z1' - \text{int } z2 - \text{int } x * \text{int } z2' = \text{int } t * \text{int } e - \text{int } t * \text{int } e'] \text{ (mod order } \mathcal{G})$   
 by (*smt cong-diff-iff-cong-0 cong-sym*)  
 hence  $[\text{int } z1 + \text{int } x * \text{int } z1' - \text{int } z2 - \text{int } x * \text{int } z2' = \text{int } t * (e - e')] \text{ (mod order } \mathcal{G})$   
 using *int-distrib(4) assms* by (*simp add: of-nat-diff*)  
 hence  $[(\text{int } z1 + \text{int } x * \text{int } z1' - \text{int } z2 - \text{int } x * \text{int } z2') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})) = \text{int } t * (e - e') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G}))] \text{ (mod order } \mathcal{G})$   
 using *cong-scalar-right* by *blast*  
 hence  $[(\text{int } z1 + \text{int } x * \text{int } z1' - \text{int } z2 - \text{int } x * \text{int } z2') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})) = \text{int } t * ((e - e') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})))] \text{ (mod order } \mathcal{G})$   
 by (*simp add: mult.assoc*)  
 hence  $[(\text{int } z1 + \text{int } x * \text{int } z1' - \text{int } z2 - \text{int } x * \text{int } z2') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})) = \text{int } t * 1] \text{ (mod order } \mathcal{G})$   
 by (*metis (no-types, opaque-lifting) cong-scalar-left cong-trans inverse gcd*)  
 hence  $[(\text{int } z1 - \text{int } z2 + \text{int } x * \text{int } z1' - \text{int } x * \text{int } z2') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})) = \text{int } t] \text{ (mod order } \mathcal{G})$   
 by *smt*  
 hence  $[(\text{int } z1 - \text{int } z2 + \text{int } x * (\text{int } z1' - \text{int } z2')) * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})) = \text{int } t] \text{ (mod order } \mathcal{G})$   
 by (*simp add: Rings.ring-distrib(4) add-diff-eq*)  
 hence  $[\text{nat} ((\text{int } z1 - \text{int } z2 + \text{int } x * (\text{int } z1' - \text{int } z2')) * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G}))) \text{ mod (order } \mathcal{G}) = \text{int } t] \text{ (mod order } \mathcal{G})$   
 by *auto*  
 hence  $\mathbf{g} [\ulcorner] (\text{nat} ((\text{int } z1 - \text{int } z2 + \text{int } x * (\text{int } z1' - \text{int } z2')) * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G}))) \text{ mod (order } \mathcal{G})) = \mathbf{g} [\ulcorner] t$   
 using *cong-int-iff finite-carrier pow-generator-eq-iff-cong* by *blast*  
 hence  $\mathbf{g} [\ulcorner] ((\text{int } z1 - \text{int } z2 + \text{int } x * (\text{int } z1' - \text{int } z2')) * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G}))) = \mathbf{g} [\ulcorner] t$   
 using *pow-generator-mod-int* by *auto*  
 hence  $\mathbf{g} [\ulcorner] ((\text{int } z1 - \text{int } z2) * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G})) + \text{int } x * (\text{int } z1' - \text{int } z2') * \text{fst} (\text{bezw } (e - e') \text{ (order } \mathcal{G}))) = \mathbf{g} [\ulcorner] t$   
 by (*metis Rings.ring-distrib(2) t*)



**hence**  $\mathbf{g} [\ulcorner] ((\text{int } z1 - \text{int } z2) * \text{fst } (\text{bezw } (e - e') (\text{order } \mathcal{G}))) \otimes \mathbf{g} [\ulcorner] (\text{int } x * (\text{int } z1' - \text{int } z2') * \text{fst } (\text{bezw } (e - e') (\text{order } \mathcal{G}))) = \mathbf{g} [\ulcorner] t$   
**using** *int-pow-mult by auto*  
**thus** *?thesis*  
**by** (*metis (mono-tags, opaque-lifting) g'-def generator-closed int-pow-int int-pow-pow mod-mult-right-eq more-arith-simps(11) pow-generator-mod-int t*)  
**qed**

**lemma**

**assumes** *h-mem: h ∈ carrier G*  
**and** *a-mem: a ∈ carrier G*  
**and** *a: g [⌋ fst z ⊗ g' [⌋ snd z = a ⊗ h [⌋ e*  
**and** *a': g [⌋ fst z' ⊗ g' [⌋ snd z' = a ⊗ h [⌋ e'*  
**and** *e-e'-mod: e' mod order G < e mod order G*  
**shows**  $h = \mathbf{g} [\ulcorner] ((\text{int } (\text{fst } z) - \text{int } (\text{fst } z')) * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}) (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G}))$   
 $\otimes \mathbf{g}' [\ulcorner] ((\text{int } (\text{snd } z) - \text{int } (\text{snd } z')) * \text{fst } (\text{bezw } ((e \text{ mod order } \mathcal{G} - e' \text{ mod order } \mathcal{G}) \text{ mod order } \mathcal{G}) (\text{order } \mathcal{G}))) \text{ mod int } (\text{order } \mathcal{G}))$   
**proof** –  
**have** *gcd: gcd ((e mod order G - e' mod order G) mod order G) (order G) = 1*  
**using** *prime-field*  
**by** (*simp add: asms less-imp-diff-less linorder-not-le prime-order*)  
**have**  $\mathbf{g} [\ulcorner] \text{fst } z \otimes \mathbf{g}' [\ulcorner] \text{snd } z \otimes \text{inv } (h [\ulcorner] e) = a$   
**using** *a h-mem a-mem by (simp add: inv-solve-right')*  
**moreover have**  $\mathbf{g} [\ulcorner] \text{fst } z' \otimes \mathbf{g}' [\ulcorner] \text{snd } z' \otimes \text{inv } (h [\ulcorner] e') = a$   
**using** *a h-mem a-mem by (simp add: asms(4) inv-solve-right')*  
**ultimately have**  $\mathbf{g} [\ulcorner] \text{fst } z \otimes \mathbf{g} [\ulcorner] (x * \text{snd } z) \otimes \text{inv } (h [\ulcorner] e) = \mathbf{g} [\ulcorner] \text{fst } z' \otimes \mathbf{g} [\ulcorner] (x * \text{snd } z') \otimes \text{inv } (h [\ulcorner] e')$   
**using** *g'-def by (simp add: nat-pow-pow)*  
**moreover obtain** *t :: nat where t: h = g [⌋ t*  
**using** *h-mem generatorE by blast*  
**ultimately have**  $\mathbf{g} [\ulcorner] \text{fst } z \otimes \mathbf{g} [\ulcorner] (x * \text{snd } z) \otimes \mathbf{g} [\ulcorner] (t * e) = \mathbf{g} [\ulcorner] \text{fst } z' \otimes \mathbf{g} [\ulcorner] (x * \text{snd } z') \otimes \mathbf{g} [\ulcorner] (t * e)$   
**using** *a-mem asms(3) asms(4) cyclic-group-assoc cyclic-group-commute g'-def nat-pow-pow by auto*  
**hence**  $\mathbf{g} [\ulcorner] (\text{fst } z + x * \text{snd } z + t * e) = \mathbf{g} [\ulcorner] (\text{fst } z' + x * \text{snd } z' + t * e)$   
**by** (*simp add: nat-pow-mult*)  
**hence**  $[\text{fst } z + x * \text{snd } z + t * e = \text{fst } z' + x * \text{snd } z' + t * e] (\text{mod order } \mathcal{G})$   
**using** *group-eq-pow-eq-mod order-gt-0 by blast*  
**hence**  $[\text{int } (\text{fst } z) + \text{int } x * \text{int } (\text{snd } z) + \text{int } t * \text{int } e = \text{int } (\text{fst } z') + \text{int } x * \text{int } (\text{snd } z') + \text{int } t * \text{int } e] (\text{mod order } \mathcal{G})$   
**using** *cong-int-iff by force*  
**hence**  $[\text{int } (\text{fst } z) - \text{int } (\text{fst } z') + \text{int } x * \text{int } (\text{snd } z) - \text{int } x * \text{int } (\text{snd } z') = \text{int } t * \text{int } e - \text{int } t * \text{int } e'] (\text{mod order } \mathcal{G})$   
**by** (*smt cong-diff-iff-cong-0*)  
**hence**  $[\text{int } (\text{fst } z) - \text{int } (\text{fst } z') + \text{int } x * (\text{int } (\text{snd } z) - \text{int } (\text{snd } z')) = \text{int } t * (\text{int } e - \text{int } e')] (\text{mod order } \mathcal{G})$   
**proof** –  
**have**  $[\text{int } (\text{fst } z) + (\text{int } (x * \text{snd } z) - (\text{int } (\text{fst } z') + \text{int } (x * \text{snd } z'))] = \text{int } t$

$\ast (int\ e - int\ e') \pmod{int\ (order\ \mathcal{G})}$   
**by** (*simp add: Rings.ring-distrib(4)*)  $\langle [int\ (fst\ z) - int\ (fst\ z') + int\ x \ast int\ (snd\ z) - int\ x \ast int\ (snd\ z') = int\ t \ast int\ e - int\ t \ast int\ e'] \pmod{int\ (order\ \mathcal{G})} \rangle$   
*add-diff-add add-diff-eq*  
**then have**  $\exists i. [int\ (fst\ z) + (int\ x \ast int\ (snd\ z) - (int\ (fst\ z') + i \ast int\ (snd\ z')) = int\ t \ast (int\ e - int\ e') + int\ (snd\ z') \ast (int\ x - i)] \pmod{int\ (order\ \mathcal{G})}$   
**by** (*metis (no-types) add.commute arith-simps(49) cancel-comm-monoid-add-class.diff-cancel int-ops(7) mult-eq-0-iff*)  
**then have**  $\exists i. [int\ (fst\ z) - int\ (fst\ z') + (int\ x \ast (int\ (snd\ z) - int\ (snd\ z')) + i) = int\ t \ast (int\ e - int\ e') + i] \pmod{int\ (order\ \mathcal{G})}$   
**by** (*metis (no-types) add-diff-add add-diff-eq mult-diff-mult mult-of-nat-commute*)  
**then show** *?thesis*  
**by** (*metis (no-types) add.assoc cong-add-rcancel*)  
**qed**  
**hence**  $[int\ (fst\ z) - int\ (fst\ z') + int\ x \ast (int\ (snd\ z) - int\ (snd\ z')) = int\ t \ast (int\ e \pmod{order\ \mathcal{G}} - int\ e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}}] \pmod{order\ \mathcal{G}}$   
**by** (*metis (mono-tags, lifting) cong-def mod-diff-eq mod-mod-trivial mod-mult-right-eq*)  
**hence**  $[int\ (fst\ z) - int\ (fst\ z') + int\ x \ast (int\ (snd\ z) - int\ (snd\ z')) = int\ t \ast (e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}}] \pmod{order\ \mathcal{G}}$   
**using** *e-e'-mod*  
**by** (*simp add: int-ops(9) of-nat-diff*)  
**hence**  $[(int\ (fst\ z) - int\ (fst\ z') + int\ x \ast (int\ (snd\ z) - int\ (snd\ z')) \ast fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G}))) \pmod{order\ \mathcal{G}}]$   
 $= int\ t \ast (e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}}$   
 $\ast fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G})) \pmod{order\ \mathcal{G}}$   
**using** *cong-cong-mod-int cong-scalar-right by blast*  
**hence**  $[(int\ (fst\ z) - int\ (fst\ z') + int\ x \ast (int\ (snd\ z) - int\ (snd\ z')) \ast fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G}))) \pmod{order\ \mathcal{G}}]$   
 $= int\ t \ast ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}} \ast fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G}))) \pmod{order\ \mathcal{G}}$   
**by** (*metis (no-types, lifting) Groups.mult-ac(1) cong-mod-right less-imp-diff-less mod-less mod-mult-left-eq mod-mult-right-eq order-gt-0 unique-euclidean-semiring-numeral-class.pos-mod-bound*)  
**hence**  $[(int\ (fst\ z) - int\ (fst\ z') + int\ x \ast (int\ (snd\ z) - int\ (snd\ z')) \ast fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G}))) \pmod{order\ \mathcal{G}}]$   
 $= int\ t \ast 1] \pmod{order\ \mathcal{G}}$   
**using** *inverse gcd*  
**by** (*smt Num.of-nat-simps(5) Number-Theory-Aux.inverse cong-def mod-mult-right-eq more-arith-simps(6) of-nat-1*)  
**hence**  $[(int\ (fst\ z) - int\ (fst\ z')) + (int\ x \ast (int\ (snd\ z) - int\ (snd\ z')))] \ast fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G})) \pmod{order\ \mathcal{G}}$   
 $= int\ t] \pmod{order\ \mathcal{G}}$   
**by** *auto*  
**hence**  $[(int\ (fst\ z) - int\ (fst\ z')) \ast (fst\ (bezw\ ((e \pmod{order\ \mathcal{G}} - e' \pmod{order\ \mathcal{G}}) \pmod{order\ \mathcal{G}})\ (order\ \mathcal{G}))) \pmod{order\ \mathcal{G}}]$



```

(snd z') * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ ))
mod int (order  $\mathcal{G}$ ) = int t] (mod int (order  $\mathcal{G}$ ))
  using <[(int (fst z) - int (fst z')) * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order
 $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ )) mod int (order  $\mathcal{G}$ ) + int x * (int (snd z) - int (snd
z')) * (fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ )) mod
int (order  $\mathcal{G}$ ) = int t] (mod int (order  $\mathcal{G}$ ))> cong-trans by blast
  then show ?thesis
    by (metis (no-types) Groups.mult-ac(1))
qed
hence g [↑] ((int (fst z) - int (fst z')) * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod
order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ )) mod order  $\mathcal{G}$  + (int x * (int (snd z) - int (snd
z'))))
  * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ ))
mod order  $\mathcal{G}$ )
  = g [↑] t
  by (metis cong-def int-pow-int pow-generator-mod-int)
hence g [↑] ((int (fst z) - int (fst z')) * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod
order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ )) mod order  $\mathcal{G}$ ) ⊗ g [↑] ((int x * (int (snd z) -
int (snd z'))))
  * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ ))
mod order  $\mathcal{G}$ )
  = g [↑] t
  using int-pow-mult by auto
hence g [↑] ((int (fst z) - int (fst z')) * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod
order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ )) mod order  $\mathcal{G}$ ) ⊗ g [↑] ((int x * ((int (snd z) -
int (snd z'))))
  * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ ))
mod order  $\mathcal{G}$ ))
  = g [↑] t
  by blast
hence g [↑] ((int (fst z) - int (fst z')) * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod
order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ )) mod order  $\mathcal{G}$ ) ⊗ g' [↑] (((int (snd z) - int (snd
z'))))
  * fst (bezw ((e mod order  $\mathcal{G}$  - e' mod order  $\mathcal{G}$ ) mod order  $\mathcal{G}$ ) (order  $\mathcal{G}$ ))
mod order  $\mathcal{G}$ ))
  = g [↑] t
  by (smt g'-def cyclic-group.generator-closed int-pow-int int-pow-pow mod-mult-right-eq
more-arith-simps(11) okamoto-axioms okamoto-def pow-generator-mod-int)
  thus ?thesis using t by simp
qed

```

**lemma** *special-soundness*:

**shows**  $\Sigma$ -protocols-base.special-soundness

**unfolding**  $\Sigma$ -protocols-base.special-soundness-def

**by**(auto simp add: valid-pub-def check-def R-def ss-adversary-def Let-def ss-rewrite
challenge-space-def split-def)

**theorem**  $\Sigma$ -protocol:

**shows**  $\Sigma$ -protocols-base. $\Sigma$ -protocol

**by**(*simp add:  $\Sigma$ -protocols-base. $\Sigma$ -protocol-def completeness HVZK special-soundness*)

**sublocale** *okamoto- $\Sigma$ -commit:  $\Sigma$ -protocols-to-commitments init response check R*  
*S2 ss-adversary challenge-space valid-pub G*  
**apply** *unfold-locales*  
**apply**(*auto simp add:  $\Sigma$ -protocol*)  
**by**(*auto simp add: G-def R-def lossless-init lossless-response*)

**sublocale** *dis-log: dis-log  $\mathcal{G}$*   
**unfolding** *dis-log-def by simp*

**sublocale** *dis-log-alt: dis-log-alt  $\mathcal{G}$  x*  
**unfolding** *dis-log-alt-def*  
**by**(*simp add:*)

**lemma** *reduction-to-dis-log:*  
**shows** *okamoto- $\Sigma$ -commit.rel-advantage  $\mathcal{A}$  = dis-log.advantage (dis-log-alt.adversary2  $\mathcal{A}$ )*  
**proof** –

**have** *exp-rewrite:  $\mathbf{g} [\wedge] w1 \otimes g' [\wedge] w2 = \mathbf{g} [\wedge] (w1 + x * w2)$  for  $w1 w2 :: nat$*   
**by** (*simp add: nat-pow-mult nat-pow-pow g'-def*)

**have** *okamoto- $\Sigma$ -commit.rel-game  $\mathcal{A}$  = TRY do {*  
*w1  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*w2  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*let h = ( $\mathbf{g} [\wedge] w1 \otimes g' [\wedge] w2$ );*  
*(w1',w2')  $\leftarrow$   $\mathcal{A}$  h;*  
*return-spmf (h =  $\mathbf{g} [\wedge] w1' \otimes g' [\wedge] w2'$ )} ELSE return-spmf False*  
**unfolding** *okamoto- $\Sigma$ -commit.rel-game-def*  
**by**(*simp add: Let-def split-def R-def G-def*)

**also have** *... = TRY do {*  
*w1  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*w2  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*let w = (w1 + x \* w2) mod (order  $\mathcal{G}$ );*  
*let h =  $\mathbf{g} [\wedge] w$ ;*  
*(w1',w2')  $\leftarrow$   $\mathcal{A}$  h;*  
*return-spmf (h =  $\mathbf{g} [\wedge] w1' \otimes g' [\wedge] w2'$ )} ELSE return-spmf False*  
**using** *g'-def exp-rewrite pow-generator-mod by simp*

**also have** *... = TRY do {*  
*w2  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*w  $\leftarrow$  map-spmf ( $\lambda w1. (x * w2 + w1) \text{ mod } (\text{order } \mathcal{G})$ ) (sample-uniform (order  $\mathcal{G}$ ))*  
*);*  
*let h =  $\mathbf{g} [\wedge] w$ ;*  
*(w1',w2')  $\leftarrow$   $\mathcal{A}$  h;*  
*return-spmf (h =  $\mathbf{g} [\wedge] w1' \otimes g' [\wedge] w2'$ )} ELSE return-spmf False*  
**including** *monad-normalisation*  
**by**(*simp add: bind-map-spmf o-def Let-def add commute*)

**also have** *... = TRY do {*  
*w2 :: nat  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*  
*w  $\leftarrow$  sample-uniform (order  $\mathcal{G}$ );*

```

    let h = g [∧] w;
    (w1',w2') ← A h;
    return-spmf (h = g [∧] w1' ⊗ g' [∧] w2')} ELSE return-spmf False
  using samp-uni-plus-one-time-pad add.commute by simp
also have ... = TRY do {
  w ← sample-uniform (order G);
  let h = g [∧] w;
  (w1',w2') ← A h;
  return-spmf (h = g [∧] w1' ⊗ g' [∧] w2')} ELSE return-spmf False
by(simp add: bind-spmf-const)
also have ... = dis-log-alt.dis-log2 A
apply(simp add: dis-log-alt.dis-log2-def Let-def dis-log-alt.g'-def g'-def)
apply(intro try-spmf-cong)
apply(intro bind-spmf-cong[OF refl]; clarsimp?)
apply auto
using exp-rewrite pow-generator-mod g'-def
  apply (metis group-eq-pow-eq-mod okamoto-axioms okamoto-base.order-gt-0
okamoto-def)
  using exp-rewrite g'-def order-gt-0-iff-finite pow-generator-eq-iff-cong by auto
ultimately have okamoto-Σ-commit.rel-game A = dis-log-alt.dis-log2 A
  by simp
hence okamoto-Σ-commit.rel-advantage A = dis-log-alt.advantage2 A
by(simp add: okamoto-Σ-commit.rel-advantage-def dis-log-alt.advantage2-def)
thus ?thesis
  by (simp add: dis-log-alt-reductions.dis-log-adv2 cyclic-group-axioms dis-log-alt.dis-log-alt-axioms
dis-log-alt-reductions.intro)
qed

lemma commitment-correct: okamoto-Σ-commit.abstract-com.correct
  by(simp add: okamoto-Σ-commit.commit-correct)

lemma okamoto-Σ-commit.abstract-com.perfect-hiding-ind-cpa A
  using okamoto-Σ-commit.perfect-hiding by blast

lemma binding:
  shows okamoto-Σ-commit.abstract-com.bind-advantage A
    ≤ dis-log.advantage (dis-log-alt.adversary2 (okamoto-Σ-commit.adversary
A))
  using okamoto-Σ-commit.bind-advantage reduction-to-dis-log by auto

end

locale okamoto-asymp =
  fixes G :: nat ⇒ 'grp cyclic-group
  and x :: nat
  assumes okamoto: ∧η. okamoto (G η)
begin

sublocale okamoto G η for η

```

**by**(*simp add: okamoto*)

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

**theorem** *sigma-protocol*:

**shows**  $\Sigma$ -protocols-base. $\Sigma$ -protocol *n*  
**by**(*simp add:  $\Sigma$ -protocol*)

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** *asyp-correct: okamoto- $\Sigma$ -commit.abstract-com.correct n*  
**using** *okamoto- $\Sigma$ -commit.commit-correct* **by** *simp*

**lemma** *asyp-perfect-hiding: okamoto- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa n (A n)*  
**using** *okamoto- $\Sigma$ -commit.perfect-hiding* **by** *blast*

**lemma** *asyp-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* ( $\lambda n. \text{okamoto-}\Sigma\text{-commit.abstract-com.bind-advantage}\ n\ (\mathcal{A}\ n)$ )  
**using** *okamoto- $\Sigma$ -commit.bind-advantage assms okamoto- $\Sigma$ -commit.abstract-com.bind-advantage-def negligible-le binding* **by** *auto*

**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]:  $x \in \text{carrier } L \implies \top \sqcup x = \top$*   
**using** *eq-is-equal top-join* **by** *auto*

**lemma** *join-top [simp]:  $x \in \text{carrier } L \implies x \sqcup \top = \top$*   
**using** *le-iff-meet* **by** *blast*

**lemma** *bot-join [simp]:  $x \in \text{carrier } L \implies \perp \sqcup x = x$*   
**using** *le-iff-meet* **by** *blast*

**lemma** *join-bot* [*simp*]:  $x \in \text{carrier } L \implies x \sqcup \perp = x$   
**by** (*metis bot-join join-comm*)

**lemma** *bot-meet* [*simp*]:  $x \in \text{carrier } L \implies \perp \sqcap x = \perp$   
**using** *bottom-meet* **by** *auto*

**lemma** *meet-bot* [*simp*]:  $x \in \text{carrier } L \implies x \sqcap \perp = \perp$   
**by** (*metis bot-meet meet-comm*)

**lemma** *top-meet* [*simp*]:  $x \in \text{carrier } L \implies \top \sqcap x = x$   
**by** (*metis le-iff-join meet-comm top-closed top-higher*)

**lemma** *meet-top* [*simp*]:  $x \in \text{carrier } L \implies x \sqcap \top = x$   
**by** (*metis meet-comm top-meet*)

**lemma** *join-idem* [*simp*]:  $x \in \text{carrier } L \implies x \sqcup x = x$   
**using** *le-iff-meet* **by** *blast*

**lemma** *meet-idem* [*simp*]:  $x \in \text{carrier } L \implies x \sqcap x = x$   
**using** *le-iff-join le-refl* **by** *presburger*

**lemma** *meet-leftcomm*:  $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 $z \in \text{carrier } L$   
**by** (*metis meet-assoc meet-comm that*)

**lemma** *join-leftcomm*:  $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 $z \in \text{carrier } L$   
**by** (*metis join-assoc join-comm that*)

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

**locale** *boolean-algebra* = *bounded-lattice*  $L$   
**for**  $L$  (**structure**) +  
**assumes** *compl-closed* [*intro, simp*]:  $x \in \text{carrier } L \implies \neg x \in \text{carrier } L$   
**and** *meet-compl-bot* [*simp*]:  $x \in \text{carrier } L \implies \neg x \sqcap x = \perp$   
**and** *join-compl-top* [*simp*]:  $x \in \text{carrier } L \implies \neg x \sqcup x = \top$   
**and** *join-meet-distrib1*:  $\llbracket x \in \text{carrier } L; y \in \text{carrier } L; z \in \text{carrier } L \rrbracket \implies x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$



**begin**

**lemma** *join-meet-distrib2*:  $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$   
**if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$   
**by** (*simp add: join-comm join-meet-distrib1 that*)

**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** –

**have**  $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$

**using** *join-left le-iff-join* **by** *auto*

**also have**  $\dots = x \sqcap (z \sqcup (x \sqcap y))$

**by** (*simp add: join-comm join-meet-distrib1 meet-assoc*)

**also have**  $\dots = ((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$

**by** (*metis join-comm le-iff-meet meet-closed meet-left that(1) that(2)*)

**also have**  $\dots = (x \sqcap y) \sqcup (x \sqcap z)$

**by** (*simp add: join-meet-distrib1*)

**finally show** *?thesis* .

**qed**

**lemma** *meet-join-distrib2*:  $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$   
**if**  $[simp]: x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$   
**by** (*simp add: meet-comm meet-join-distrib1*)

**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*  $[simp]: x \in \text{carrier } L \implies x \sqcap - x = \perp$   
**by** (*metis meet-comm meet-compl-bot*)

**lemma** *join-compl2-top*  $[simp]: x \in \text{carrier } L \implies x \sqcup - x = \top$   
**by** (*metis join-comm join-compl-top*)

**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** –

**have**  $(x \sqcap - x) \sqcup (- x \sqcap y) = (x \sqcap y) \sqcup (- x \sqcap y)$

**using** *inf-compl-bot assms(1)* **by** *simp*

**then have**  $(- x \sqcap x) \sqcup (- x \sqcap y) = (y \sqcap x) \sqcup (y \sqcap - x)$

**by** (*simp add: meet-comm*)

**then have**  $- x \sqcap (x \sqcup y) = y \sqcap (x \sqcup - x)$

**using** *assms(3) assms(4) compl-closed meet-join-distrib1* **by** *presburger*

**then have**  $- x \sqcap \top = y \sqcap \top$

by (*simp add: assms(2)*)  
 then show  $- x = y$   
 using *le-iff-join top-higher* by *auto*  
 qed

**lemma** *double-compl* [*simp*]:  $-( - x ) = x$  **if** [*simp*]:  $x \in \text{carrier } L$   
 by(*rule compl-unique*)(*simp-all*)

**lemma** *compl-eq-compl-iff* [*simp*]:  $- x = - y \iff x = y$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 by (*metis double-compl that that*)

**lemma** *compl-bot-eq* [*simp*]:  $- \perp = \top$   
 using *le-iff-join le-iff-meet local.compl-unique top-higher* by *auto*

**lemma** *compl-top-eq* [*simp*]:  $- \top = \perp$   
 by (*metis bottom-closed compl-bot-eq double-compl*)

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

**proof** (*rule compl-unique*)  
 have  $(x \sqcap y) \sqcap (- x \sqcup - y) = (y \sqcap (x \sqcap - x)) \sqcup (x \sqcap (y \sqcap - y))$   
 by (*smt compl-closed meet-assoc meet-closed meet-comm meet-join-distrib1 that*)  
 then show  $(x \sqcap y) \sqcap (- x \sqcup - y) = \perp$   
 by (*metis bottom-closed bottom-lower le-iff-join le-iff-meet meet-comm meet-compl2-bot that*)  
**next**  
 have  $(x \sqcap y) \sqcup (- x \sqcup - y) = (- y \sqcup (x \sqcup - x)) \sqcap (- x \sqcup (y \sqcup - y))$   
 by (*smt compl-closed join-meet-distrib2 join-assoc join-comm local.boolean-algebra-axioms that join-closed*)  
 then show  $(x \sqcap y) \sqcup (- x \sqcup - y) = \top$   
 by (*metis compl-closed join-compl2-top join-right le-iff-join le-iff-meet that top-closed*)  
 qed *simp-all*

**lemma** *compl-sup* [*simp*]:  $-( x \sqcup y ) = - x \sqcap - y$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 by (*metis compl-closed compl-inf double-compl meet-closed that*)

**lemma** *compl-mono*:  
 assumes  $x \sqsubseteq y$   
 and  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 shows  $- y \sqsubseteq - x$   
 by (*metis assms(1) assms(2) assms(3) compl-closed join-comm le-iff-join le-iff-meet compl-inf*)

**lemma** *compl-le-compl-iff* [*simp*]:  $- x \sqsubseteq - y \iff y \sqsubseteq x$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
 using *that* by (*auto dest: compl-mono*)

**lemma** *compl-le-swap1*:  
**assumes**  $y \sqsubseteq - x$   $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**shows**  $x \sqsubseteq - y$   
**by** (*metis assms compl-closed compl-le-compl-iff double-compl*)

**lemma** *compl-le-swap2*:  
**assumes**  $- y \sqsubseteq x$   $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**shows**  $- x \sqsubseteq y$   
**by** (*metis assms compl-closed compl-le-compl-iff double-compl*)

**lemma** *join-compl-top-left1* [*simp*]:  $- x \sqcup (x \sqcup y) = \top$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**by** (*simp add: join-assoc[symmetric]*)

**lemma** *join-compl-top-left2* [*simp*]:  $x \sqcup (- x \sqcup y) = \top$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**using** *join-compl-top-left1* [*of - x y*] **by** *simp*

**lemma** *meet-compl-bot-left1* [*simp*]:  $- x \sqcap (x \sqcap y) = \perp$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**by** (*simp add: meet-assoc[symmetric]*)

**lemma** *meet-compl-bot-left2* [*simp*]:  $x \sqcap (- x \sqcap y) = \perp$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**using** *meet-compl-bot-left1* [*of - x y*] **by** *simp*

**lemma** *meet-compl-bot-right* [*simp*]:  $x \sqcap (y \sqcap - x) = \perp$  **if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   
**by** (*metis meet-compl-bot-left2 meet-comm that*)

**lemma** *xor-closed* [*intro, simp*]:  $\llbracket x \in \text{carrier } L; y \in \text{carrier } L \rrbracket \implies x \oplus y \in \text{carrier } L$   
**by**(*simp add: xor-def*)

**lemma** *xor-comm*:  $\llbracket x \in \text{carrier } L; y \in \text{carrier } L \rrbracket \implies x \oplus y = y \oplus x$   
**by**(*simp add: xor-def meet-join-distrib join-comm*)

**lemma** *xor-assoc*:  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$   
**if** [*simp*]:  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$   
**by**(*simp add: xor-def*)(*simp add: meet-join-distrib meet-ac join-ac*)

**lemma** *xor-left-comm*:  $x \oplus (y \oplus z) = y \oplus (x \oplus z)$  **if**  $x \in \text{carrier } L$   $y \in \text{carrier } L$   $z \in \text{carrier } L$   
**using** *that xor-assoc xor-comm* **by** *auto*

**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:  $x \oplus \top = \neg x$ 
and top-xor:  $\top \oplus x = \neg x$ 
by(simp-all add: xor-def assms)

lemma xor-inverse [simp]:  $x \oplus x = \perp$  if  $x \in \text{carrier } L$ 
by(simp add: xor-def that)

lemma xor-left-inverse [simp]:  $x \oplus x \oplus y = y$  if  $x \in \text{carrier } L$   $y \in \text{carrier } L$ 
using that xor-assoc by fastforce

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

lemma inj-on-xor: inj-on  $((\oplus) x)$  (carrier L) if  $x \in \text{carrier } L$ 
by(rule inj-onI)(metis that xor-left-inverse)

lemma surj-xor:  $(\oplus) x \text{ 'carrier } L = \text{carrier } L$  if [simp]:  $x \in \text{carrier } L$ 
proof(rule Set.set-eqI, rule iffI)
  fix y
  assume [simp]:  $y \in \text{carrier } L$ 
  have  $x \oplus y \in \text{carrier } L$  by(simp)
  moreover have  $y = x \oplus (x \oplus y)$  by simp
  ultimately show  $y \in (\oplus) x \text{ 'carrier } L$  by(rule rev-image-eqI)
qed auto

lemma one-time-pad:  $\text{map-spmf } ((\oplus) x) (\text{spmof-of-set } (\text{carrier } L)) = \text{spmof-of-set}$ 
(carrier L)
if  $x \in \text{carrier } L$ 
apply(subst map-spmf-of-set-inj-on)
apply(rule inj-on-xor[OF that])
by(simp add: surj-xor that)

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 ⇒ '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 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) ∈ set-spmf G ⇒ Rel h w
begin

```

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

**definition** *Rel-AND* :: (('pub0 × 'pub1) × 'witness0 × 'witness1) set  
**where** *Rel-AND* = {(*x0*,*x1*), (*w0*,*w1*)). ((*x0*,*w0*) ∈ *Rel0* ∧ (*x1*,*w1*) ∈ *Rel1*)}

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

**lemma** *lossless-init-AND*: lossless-spmf (*init-AND X W*)  
**by** (*simp add: lossless-init init-AND-def split-def*)

**definition** *response-AND* :: ('rand0 × 'rand1) ⇒ ('witness0 × 'witness1) ⇒ 'bool  
⇒ ('response0 × 'response1) spmf  
**where** *response-AND R W s* = *do* {

```

let (r0,r1) = R;
let (w0,w1) = W;
z0 ← response0 r0 w0 s;
z1 :: 'response1 ← response1 r1 w1 s;
return-spmf (z0,z1)}

```

**lemma** *lossless-response-AND*: *lossless-spmf (response-AND R W s)*  
**by** (*simp add: response-AND-def lossless-response split-def*)

```

fun check-AND :: ('pub0 × 'pub1) ⇒ ('msg0 × 'msg1) ⇒ 'bool ⇒ ('response0 ×
'response1) ⇒ bool
where check-AND (x0,x1) (a0,a1) s (z0,z1) = (check0 x0 a0 s z0 ∧ check1 x1
a1 s z1)

```

**definition** *S-AND* :: 'pub0 × 'pub1 ⇒ 'bool ⇒ (('msg0 × 'msg1) × 'response0 × '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**  $\Sigma$ -AND:  $\Sigma$ -protocols-base *init-AND response-AND check-AND Rel-AND S-AND Ass-AND challenge-space valid-pub-AND*

**apply** *unfold-locales* **apply** (*simp add: Rel-AND-def valid-pub-AND-def*)  
**using**  $\Sigma 1$ .*domain-subset-valid-pub*  $\Sigma 0$ .*domain-subset-valid-pub* **by** *blast*

**end**

**locale**  $\Sigma$ -AND =  $\Sigma$ -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**  $\Sigma$ -AND.*R* (x0,x1) (w0,w1) e =  $\Sigma$ -AND.*S* (x0,x1) e

**including** *monad-normalisation*

**proof** –

**have**  $x\text{-in-dom}: x0 \in \text{Domain Rel0}$  **and**  $x1 \in \text{Domain Rel1}$

**using**  $\text{Rel-AND Rel-AND-def}$  **by**  $\text{auto}$

**have**  $\Sigma\text{-AND.R} (x0,x1) (w0,w1) e = \text{do}$  {

$((r0,r1),(a0,a1)) \leftarrow \text{init-AND} (x0,x1) (w0,w1);$

$(z0,z1) \leftarrow \text{response-AND} (r0,r1) (w0,w1) e;$

$\text{return-spmf} ((a0,a1),e,(z0,z1))$ }

**by**( $\text{simp add: } \Sigma\text{-AND.R-def split-def}$ )

**also have**  $\dots = \text{do}$  {

$(r0, a0) \leftarrow \text{init0 } x0 w0;$

$z0 \leftarrow \text{response0 } r0 w0 e;$

$(r1, a1) \leftarrow \text{init1 } x1 w1;$

$z1 :: 'f \leftarrow \text{response1 } r1 w1 e;$

$\text{return-spmf} ((a0,a1),e,(z0,z1))$ }

**apply**( $\text{simp add: init-AND-def response-AND-def split-def}$ )

**apply**( $\text{rewrite bind-commute-spmf[of response0 - w0 e]}$ )

**by**  $\text{simp}$

**also have**  $\dots = \text{do}$  {

$(a0, c0, z0) \leftarrow \Sigma0.R x0 w0 e;$

$(a1, c1, z1) \leftarrow \Sigma1.R x1 w1 e;$

$\text{return-spmf} ((a0,a1),e,(z0,z1))$ }

**by**( $\text{simp add: } \Sigma0.R-def \Sigma1.R-def \text{split-def}$ )

**also have**  $\dots = \text{do}$  {

$(a0, c0, z0) \leftarrow \Sigma0.S x0 e;$

$(a1, c1, z1) \leftarrow \Sigma1.S x1 e;$

$\text{return-spmf} ((a0,a1),e,(z0,z1))$ }

**using**  $\text{Rel-AND-def S-AND-def } \Sigma\text{-prot1 } \Sigma\text{-prot0 assms } \Sigma0.HVZK\text{-unfold1}$

$\Sigma1.HVZK\text{-unfold1}$

$\text{valid-pub-AND-def split-def challenge-space-def } x\text{-in-dom}$

**by**  $\text{auto}$

**ultimately show**  $?thesis$

**by**( $\text{simp add: } \Sigma0.S-def \Sigma1.S-def \text{bind-map-spmf o-def split-def Let-def } \Sigma\text{-AND.S-def}$

$\text{map-spmf-conv-bind-spmf S-AND-def}$ )

**qed**

**lemma**  $HVZK: \Sigma\text{-AND.HVZK}$

**using**  $\Sigma\text{-AND.HVZK-def hvzk challenge-space-def}$

**apply**( $\text{simp add: S-AND-def split-def}$ )

**using**  $\Sigma\text{-prot1 } \Sigma\text{-prot0 } \Sigma0.HVZK\text{-unfold2 } \Sigma1.HVZK\text{-unfold2 } \text{valid-pub-AND-def}$

**by**  $\text{auto}$

**lemma**  $\text{correct:}$

**assumes**  $\text{Rel-AND: } ((x0,x1), (w0,w1)) \in \text{Rel-AND}$

**and**  $e \in \text{challenge-space}$

**shows**  $\Sigma\text{-AND.completeness-game} (x0,x1) (w0,w1) e = \text{return-spmf True}$

**including**  $\text{monad-normalisation}$

**proof** –

**have**  $\Sigma\text{-AND.completeness-game} (x0,x1) (w0,w1) e = \text{do}$  {

$((r0,r1),(a0,a1)) \leftarrow \text{init-AND} (x0,x1) (w0,w1);$

```

    (z0,z1) ← response-AND (r0,r1) (w0,w1) e;
    return-spmf (check-AND (x0,x1) (a0,a1) e (z0,z1))}
  by(simp add: Σ-AND.completeness-game-def split-def del: check-AND.simps)
also have ... = do {
  (r0, a0) ← init0 x0 w0;
  z0 ← response0 r0 w0 e;
  (r1, a1) ← init1 x1 w1;
  z1 ← response1 r1 w1 e;
  return-spmf ((check0 x0 a0 e z0 ∧ check1 x1 a1 e z1))}
apply(simp add: init-AND-def response-AND-def split-def)
apply(rewrite bind-commute-spmf[of response0 - w0 e])
by simp
ultimately show ?thesis
using Σ1.complete-game-return-true Σ-prot1 Σ1.Σ-protocol-def Σ1.completeness-game-def
  assms
  Σ0.complete-game-return-true Σ-prot0 Σ0.Σ-protocol-def Σ0.completeness-game-def
  challenge-space-def
apply(auto simp add: Let-def split-def bind-eq-return-spmf lossless-init loss-
  less-response Rel-AND-def)
by(metis (mono-tags, lifting) assms(2) fst-conv snd-conv)+
qed

```

**lemma completeness:**  $\Sigma$ -AND.completeness  
**using**  $\Sigma$ -AND.completeness-def correct challenge-space-def **by force**

**lemma ss:**

```

assumes e-neq-e': s ≠ s'
and valid-pub: (x0,x1) ∈ valid-pub-AND
and challenge-space: s ∈ challenge-space s' ∈ challenge-space
and check-AND (x0,x1) (a0,a1) s (z0,z1)
and check-AND (x0,x1) (a0,a1) s' (z0',z1')
shows lossless-spmf (Ass-AND (x0,x1) ((a0,a1), s, (z0,z1)) ((a0,a1), s',
(z0',z1')))
  ∧ (∀ w' ∈ set-spmf (Ass-AND (x0,x1) ((a0,a1), s, (z0,z1)) ((a0,a1),
s', (z0',z1'))). ((x0,x1), w') ∈ Rel-AND)

```

**proof**–

```

have x0-in-dom: x0 ∈ valid-pub0 and x1-in-dom: x1 ∈ valid-pub1
using valid-pub valid-pub-AND-def by auto
moreover have 3: check0 x0 a0 s z0
using assms by simp
moreover have 4: check1 x1 a1 s' z1'
using assms by simp
moreover have w0 ∈ set-spmf (Ass0 x0 (a0, s, z0) (a0, s', z0')) → (x0,w0)
∈ Rel0 for w0
using 3 4 Σ0.special-soundness-def Σ-prot0 Σ0.Σ-protocol-def x0-in-dom chal-
lenge-space-def assms valid-pub-AND-def valid-pub by fastforce
moreover have w1 ∈ set-spmf (Ass1 x1 (a1, s, z1) (a1, s', z1')) → (x1,w1)
∈ Rel1 for w1
using 3 4 Σ1.special-soundness-def Σ-prot1 Σ1.Σ-protocol-def x1-in-dom chal-

```



*lenge-space-def* *assms* *valid-pub-AND-def* *valid-pub* **by** *fastforce*  
**ultimately show** *?thesis*  
**by**(*auto simp add: lossless-Ass Rel-AND-def*)  
**qed**

**lemma** *special-soundness*:  
**shows**  $\Sigma$ -*AND*.*special-soundness*  
**using**  $\Sigma$ -*AND*.*special-soundness-def* *ss* **by** *fast*

**theorem**  $\Sigma$ -*protocol*:  
**shows**  $\Sigma$ -*AND*. $\Sigma$ -*protocol*  
**by**(*auto simp add: \Sigma-AND.\Sigma-protocol-def completeness HVZK special-soundness*)

**sublocale** *AND-\Sigma-commit: \Sigma-protocols-to-commitments init-AND response-AND check-AND Rel-AND S-AND Ass-AND challenge-space valid-pub-AND G*  
**apply** *unfold-locales*  
**by**(*auto simp add: \Sigma-protocol set-mpmf-G-L lossless-G lossless-init-AND lossless-response-AND*)

**lemma** *AND-\Sigma-commit.abstract-com.correct*  
**using** *AND-\Sigma-commit.commit-correct* **by** *simp*

**lemma** *AND-\Sigma-commit.abstract-com.perfect-hiding-ind-cpa A*  
**using** *AND-\Sigma-commit.perfect-hiding* **by** *blast*

**lemma** *bind-advantage-bound-dis-log*:  
**shows** *AND-\Sigma-commit.abstract-com.bind-advantage*  $\mathcal{A} \leq$  *AND-\Sigma-commit.rel-advantage*  
(*AND-\Sigma-commit.adversary A*)  
**using** *AND-\Sigma-commit.bind-advantage* **by** *simp*

**end**

**end**

## 2.7 $\Sigma$ -OR statements

**theory** *Sigma-OR* **imports**  
*Sigma-Protocols*  
*Xor*  
**begin**

**locale**  $\Sigma$ -*OR-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 ⇒ '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)
  using assms
  by (meson DomainE Rel-OR.cases)

lemma set-spmf-lists-sample [simp]: set-spmf (spmf-of-set (carrier L)) = (carrier
L)
  using finite-L by simp

definition challenge-space = carrier L

fun init-OR :: ('pub0 × 'pub1) ⇒ ('witness0 + 'witness1) ⇒ (((('rand0 × 'bool

```

$\times 'response1 + 'rand1 \times 'bool \times 'response0)) \times 'msg0 \times 'msg1)) \text{ spmf}$   
**where**  $init\text{-}OR (x0,x1) (Inl w0) = do \{$   
 $(r0,a0) \leftarrow init0 x0 w0;$   
 $e1 \leftarrow \text{spmf-of-set } (carrier L);$   
 $(a1, e'1, z1) \leftarrow \Sigma1.S x1 e1;$   
 $return\text{-}spmf (Inl (r0, e1, z1), a0, a1)\} |$   
 $init\text{-}OR (x0, x1) (Inr w1) = do \{$   
 $(r1, a1) \leftarrow init1 x1 w1;$   
 $e0 \leftarrow \text{spmf-of-set } (carrier L);$   
 $(a0, e'0, z0) \leftarrow \Sigma0.S x0 e0;$   
 $return\text{-}spmf ((Inr (r1, e0, z0), a0, a1))\}$

**lemma**  $lossless\text{-}\Sigma\text{-}S$ :  $lossless\text{-}spmf (\Sigma1.S x1 e1) lossless\text{-}spmf (\Sigma0.S x0 e0)$   
**using**  $lossless\text{-}S$  **by**  $fast +$

**lemma**  $lossless\text{-}init\text{-}OR$ :  $lossless\text{-}spmf (init\text{-}OR (x0,x1) w)$

**by**( $cases w$ ;  $simp$   $add$ :  $lossless\text{-}\Sigma\text{-}S$   $split\text{-}def$   $lossless\text{-}init$   $lossless\text{-}S$   $finite\text{-}L$   $carrier\text{-}L\text{-}not\text{-}empty$ )

**fun**  $response\text{-}OR :: (('rand0 \times 'bool \times 'response1 + 'rand1 \times 'bool \times 'response0))$   
 $\Rightarrow ('witness0 + 'witness1)$

$\Rightarrow 'bool \Rightarrow (('bool \times 'response0) \times ('bool \times 'response1)) \text{ spmf}$

**where**  $response\text{-}OR (Inl (r0, e-1, z1)) (Inl w0) s = do \{$

$let e0 = s \oplus e-1;$

$z0 \leftarrow response0 r0 w0 e0;$

$return\text{-}spmf ((e0,z0), (e-1,z1))\} |$

$response\text{-}OR (Inr (r1, e-0, z0)) (Inr w1) s = do \{$

$let e1 = s \oplus e-0;$

$z1 \leftarrow response1 r1 w1 e1;$

$return\text{-}spmf ((e-0, z0), (e1, z1))\}$

**definition**  $check\text{-}OR :: ('pub0 \times 'pub1) \Rightarrow ('msg0 \times 'msg1) \Rightarrow 'bool \Rightarrow (('bool \times 'response0) \times ('bool \times 'response1)) \Rightarrow bool$

**where**  $check\text{-}OR X A s Z$

$= (s = (fst (fst Z)) \oplus (fst (snd Z)))$

$\wedge (fst (fst Z)) \in challenge\text{-}space \wedge (fst (snd Z)) \in challenge\text{-}space$

$\wedge check0 (fst X) (fst A) (fst (fst Z)) (snd (fst Z)) \wedge check1 (snd$

$X) (snd A) (fst (snd Z)) (snd (snd Z)))$

**lemma**  $check\text{-}OR (x0,x1) (a0,a1) s ((e0,z0), (e1,z1))$

$= (s = e0 \oplus e1$

$\wedge e0 \in challenge\text{-}space \wedge e1 \in challenge\text{-}space$

$\wedge check0 x0 a0 e0 z0 \wedge check1 x1 a1 e1 z1)$

**by**( $simp$   $add$ :  $check\text{-}OR\text{-}def$ )

**fun**  $S\text{-}OR$  **where**  $S\text{-}OR (x0,x1) c = do \{$

$e1 \leftarrow \text{spmf-of-set } (carrier L);$

$(a1, e1', z1) \leftarrow \Sigma1.S x1 e1;$

$let e0 = c \oplus e1;$

```

(a0, e0', z0) ← Σ0.S x0 e0;
let z = ((e0',z0), (e1',z1));
return-spmf ((a0, a1),z)

```

**definition**  $\mathcal{A}ss\text{-}OR'$  :: 'pub0 × 'pub1 ⇒ ('msg0 × 'msg1) × 'bool × ('bool × 'response0) × 'bool × 'response1  
⇒ ('msg0 × 'msg1) × 'bool × ('bool × 'response0) × 'bool × 'response1 ⇒ ('witness0 + 'witness1) spmf  
**where**  $\mathcal{A}ss\text{-}OR'$  X C1 C2 = TRY do {  
- :: unit ← assert-spmf ((fst (fst (snd (snd C1)))) ≠ (fst (fst (snd (snd C2))));  
w0 :: 'witness0 ← 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 ← 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**  $\mathcal{A}ss\text{-}OR$  :: 'pub0 × 'pub1 ⇒ ('msg0 × 'msg1) × 'bool × ('bool × 'response0) × 'bool × 'response1  
⇒ ('msg0 × 'msg1) × 'bool × ('bool × 'response0) × 'bool × 'response1 ⇒ ('witness0 + 'witness1) spmf  
**where**  $\mathcal{A}ss\text{-}OR$  X C1 C2 = do {  
if ((fst (fst (snd (snd C1)))) ≠ (fst (fst (snd (snd C2))))) then do  
{w0 :: 'witness0 ← 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 ← 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**  $\mathcal{A}ss\text{-}OR\text{-}alt\text{-}def$ :  $\mathcal{A}ss\text{-}OR$  (x0,x1) ((a0,a1),s,(e0,z0),e1,z1) ((a0,a1),s',(e0',z0'),e1',z1')  
= do {  
if (e0 ≠ e0') then do {w0 :: 'witness0 ← Ass0 x0 (a0,e0,z0) (a0,e0',z0');  
return-spmf (Inl w0)}  
else do {w1 :: 'witness1 ← Ass1 x1 (a1,e1,z1) (a1,e1',z1'); return-spmf (Inr w1)}

**by**(simp add:  $\mathcal{A}ss\text{-}OR\text{-}def$ )

**definition**  $valid\text{-}pub\text{-}OR$  = {(x0,x1). x0 ∈  $valid\text{-}pub0$  ∧ x1 ∈  $valid\text{-}pub1$ }

**sublocale** Σ-OR: Σ-protocols-base init-OR response-OR check-OR Rel-OR S-OR  
 $\mathcal{A}ss\text{-}OR$  challenge-space valid-pub-OR

**unfolding** Σ-protocols-base-def

**proof**(goal-cases)

**case** 1

**then show** ?case

**proof**

```

fix  $x$ 
assume  $asm: x \in \text{Domain Rel-OR}$ 
then obtain  $x0\ x1$  where  $x: (x0,x1) = x$ 
  by  $(metis\ surj-pair)$ 
show  $x \in \text{valid-pub-OR}$ 
proof  $(cases\ \exists\ w0. (x0,w0) \in \text{Rel0} \wedge x1 \in \text{valid-pub1})$ 
  case  $\text{True}$ 
    then show  $?thesis$ 
      using  $\Sigma0.\text{domain-subset-valid-pub valid-pub-OR-def } x$  by  $auto$ 
  next
    case  $\text{False}$ 
      hence  $\exists\ w1. (x1,w1) \in \text{Rel1} \wedge x0 \in \text{valid-pub0}$ 
        using  $\text{Domain-Rel-cases } asm\ x$  by  $auto$ 
      then show  $?thesis$ 
        using  $\Sigma1.\text{domain-subset-valid-pub valid-pub-OR-def } x$  by  $auto$ 
    qed
  qed
qed

end

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

lemma  $HVZK1:$ 
  assumes  $(x1,w1) \in \text{Rel1}$ 
  shows  $\forall\ c \in \text{challenge-space. } \Sigma\text{-OR.R } (x0,x1) (Inr\ w1)\ c = \Sigma\text{-OR.S } (x0,x1)\ c$ 
  including  $\text{monad-normalisation}$ 
proof
  fix  $c$ 
  assume  $c: c \in \text{challenge-space}$ 
  show  $\Sigma\text{-OR.R } (x0,x1) (Inr\ w1)\ c = \Sigma\text{-OR.S } (x0,x1)\ c$ 
  proof–
    have  $*$ :  $x \in \text{carrier } L \longrightarrow c \oplus c \oplus x = x$  for  $x$ 
      using  $c\ \text{challenge-space-def}$  by  $auto$ 
    have  $\Sigma\text{-OR.R } (x0,x1) (Inr\ w1)\ c = do\ \{$ 
       $(r1, ab1) \leftarrow \text{init1 } x1\ w1;$ 
       $eb' \leftarrow \text{spmof-of-set } (\text{carrier } L);$ 
       $(ab0', eb0'', zb0') \leftarrow \Sigma0.S\ x0\ eb';$ 
       $let\ ((r, eb', zb'), a) = ((r1, eb', zb0'),\ ab0',\ ab1);$ 
       $let\ eb = c \oplus eb';$ 
       $zb1 \leftarrow \text{response1 } r\ w1\ eb;$ 
       $let\ z = ((eb', zb'), (eb, zb1));$ 
       $\text{return-spmf } (a,c,z)\}$ 
    supply  $[[\text{simproc del: monad-normalisation}]]$ 
    by  $(\text{simp add: } \Sigma\text{-OR.R-def split-def Let-def})$ 
    also have  $\dots = do\ \{$ 

```

```

    eb' ← spmf-of-set (carrier L);
    (ab0', eb0'', zb0') ← Σ0.S x0 eb';
    let eb = c ⊕ eb';
    (ab1, c', zb1) ← Σ1.R x1 w1 eb;
    let z = ((eb', zb0'), (eb, zb1));
    return-spmf ((ab0', ab1), c, z)
  by(simp add: Σ1.R-def split-def Let-def)
also have ... = do {
  eb' ← spmf-of-set (carrier L);
  (ab0', eb0'', zb0') ← Σ0.S x0 eb';
  let eb = c ⊕ eb';
  (ab1, c', zb1) ← Σ1.S x1 eb;
  let z = ((eb', zb0'), (eb, zb1));
  return-spmf ((ab0', ab1), c, z)
  using c
  by(simp add: split-def Let-def Σ-prot1 Σ1.HVZK-unfold1 assms challenge-space-def
  cong: bind-spmf-cong-simp)
also have ... = do {
  eb ← map-spmf (λ eb'. c ⊕ eb') (spmf-of-set (carrier L));
  (ab1, c', zb1) ← Σ1.S x1 eb;
  (ab0', eb0'', zb0') ← Σ0.S x0 (c ⊕ eb);
  let z = ((c ⊕ eb, zb0'), (eb, zb1));
  return-spmf ((ab0', ab1), c, z)
  apply(simp add: bind-map-spmf o-def Let-def)
  by(simp add: * split-def cong: bind-spmf-cong-simp)
also have ... = do {
  eb ← (spmf-of-set (carrier L));
  (ab1, c', zb1) ← Σ1.S x1 eb;
  (ab0', eb0'', zb0') ← Σ0.S x0 (c ⊕ eb);
  let z = ((c ⊕ eb, zb0'), (eb, zb1));
  return-spmf ((ab0', ab1), c, z)
  using assms assms one-time-pad c challenge-space-def by simp
also have ... = do {
  eb ← (spmf-of-set (carrier L));
  (ab1, c', zb1) ← Σ1.S x1 eb;
  (ab0', eb0'', zb0') ← Σ0.S x0 (c ⊕ eb);
  let z = ((eb0'', zb0'), (c', zb1));
  return-spmf ((ab0', ab1), c, z)
  by(simp add: Σ0.S-def Σ1.S-def bind-map-spmf o-def split-def)
ultimately show ?thesis by(simp add: Let-def map-spmf-conv-bind-spmf Σ-OR.S-def
  split-def)
qed
qed

```

**lemma HVZK0:**

**assumes**  $(x0, w0) \in Rel0$

**shows**  $\forall c \in challenge-space. \Sigma-OR.R (x0, x1) (Inl w0) c = \Sigma-OR.S (x0, x1) c$

**proof**

**fix**  $c$

```

assume  $c: c \in \text{challenge-space}$ 
show  $\Sigma\text{-OR.R } (x0,x1) (\text{Inl } w0) c = \Sigma\text{-OR.S } (x0,x1) c$ 
proof –
  have  $\Sigma\text{-OR.R } (x0,x1) (\text{Inl } w0) c = \text{do } \{$ 
     $(r0,ab0) \leftarrow \text{init0 } x0 w0;$ 
     $eb' \leftarrow \text{spmof-of-set } (\text{carrier } L);$ 
     $(ab1', eb1'', zb1') \leftarrow \Sigma1.S x1 eb';$ 
     $\text{let } ((r, eb', zb'), a) = ((r0, eb', zb1'), ab0, ab1');$ 
     $\text{let } eb = c \oplus eb';$ 
     $zb0 \leftarrow \text{response0 } r w0 eb;$ 
     $\text{let } z = ((eb, zb0), (eb', zb1'));$ 
     $\text{return-spmf } (a, c, z)$ 
    by(simp add:  $\Sigma\text{-OR.R-def split-def Let-def}$ )
  also have  $\dots = \text{do } \{$ 
     $eb' \leftarrow (\text{spmof-of-set } (\text{carrier } L));$ 
     $(ab1', eb1'', zb1') \leftarrow \Sigma1.S x1 eb';$ 
     $\text{let } eb = c \oplus eb';$ 
     $(ab0, c', zb0) \leftarrow \Sigma0.R x0 w0 eb;$ 
     $\text{let } z = ((eb, zb0), (eb', zb1'));$ 
     $\text{return-spmf } ((ab0, ab1'), c, z)$ 
    apply(simp add:  $\Sigma0.R-def split-def Let-def$ )
    apply(rewrite bind-commute-spmf)
    apply(rewrite bind-commute-spmf[of -  $\Sigma1.S$  - -])
    by simp
  also have  $\dots = \text{do } \{$ 
     $eb' \leftarrow (\text{spmof-of-set } (\text{carrier } L));$ 
     $(ab1', eb1'', zb1') \leftarrow \Sigma1.S x1 eb';$ 
     $\text{let } eb = c \oplus eb';$ 
     $(ab0, c', zb0) \leftarrow \Sigma0.S x0 eb;$ 
     $\text{let } z = ((eb, zb0), (eb', zb1'));$ 
     $\text{return-spmf } ((ab0, ab1'), c, z)$ 
    using  $c$ 
    by(simp add:  $\Sigma\text{-prot0 } \Sigma0.HVZK\text{-unfold1 assms challenge-space-def split-def$ 
Let-def cong: bind-spmf-cong-simp)
  ultimately show ?thesis
    by(simp add:  $\Sigma\text{-OR.S-def } \Sigma1.S\text{-def } \Sigma0.S\text{-def Let-def o-def bind-map-spmf$ 
split-def map-spmf-conv-bind-spmf)
  qed
qed

lemma HVZK:
  shows  $\Sigma\text{-OR.HVZK}$ 
  unfolding  $\Sigma\text{-OR.HVZK-def}$ 
  apply auto
  subgoal for  $e a b w$ 
    apply(cases w)
    using  $HVZK0 HVZK1$  by auto
  apply(auto simp add: valid-pub-OR-def  $\Sigma\text{-OR.S-def bind-map-spmf o-def check-OR-def$ 
image-def  $\Sigma0.S\text{-def } \Sigma1.S\text{-def split-def challenge-space-def local.xor-ac(1)$ )

```

```

using  $\Sigma 0.HVZK\text{-unfold2}$   $\Sigma\text{-prot0}$  challenge-space-def apply force
using  $\Sigma 1.HVZK\text{-unfold2}$   $\Sigma\text{-prot1}$  challenge-space-def by force

lemma assumes  $(x0,x1) \in \text{Domain } \text{Rel-OR}$ 
shows  $(\exists w0. (x0,w0) \in \text{Rel0}) \vee (\exists w1. (x1,w1) \in \text{Rel1})$ 
using assms Rel-OR.simps by blast

lemma ss:
assumes valid-pub-OR:  $(x0,x1) \in \text{valid-pub-OR}$ 
and check: check-OR  $(x0,x1) (a0,a1) s ((e0,z0), (e1,z1))$ 
and check': check-OR  $(x0,x1) (a0,a1) s' ((e0',z0'), (e1',z1'))$ 
and  $s \neq s'$ 
and challenge-space:  $s \in \text{challenge-space } s' \in \text{challenge-space}$ 
shows lossless-spmf  $(\text{Ass-OR } (x0,x1) ((a0,a1),s,(e0,z0), e1,z1) ((a0,a1),s',(e0',z0'), e1',z1')) \wedge$ 
 $(\forall w' \in \text{set-spmf } (\text{Ass-OR } (x0,x1) ((a0,a1),s,(e0,z0), e1,z1) ((a0,a1),s',(e0',z0'), e1',z1')). ((x0,x1), w') \in \text{Rel-OR})$ 
proof -
have e-or:  $e0 \neq e0' \vee e1 \neq e1'$  using assms check-OR-def by auto
show ?thesis
proof (cases  $e0 \neq e0'$ )
case True
moreover have 2:  $x0 \in \text{valid-pub0}$ 
using valid-pub-OR valid-pub-OR-def by simp
moreover have 3: check0  $x0 a0 e0 z0$ 
using assms check-OR-def by simp
moreover have 4: check0  $x0 a0 e0' z0'$ 
using assms check-OR-def by simp
moreover have e:  $e0 \in \text{carrier } L e0' \in \text{carrier } L$ 
using challenge-space-def check check' check-OR-def by auto
ultimately have  $(\forall w' \in \text{set-spmf } (\text{Ass0 } x0 (a0,e0,z0) (a0,e0',z0')). (x0, w') \in \text{Rel0})$ 
using True  $\Sigma 0.\Sigma\text{-protocol-def}$   $\Sigma 0.\text{special-soundness-def}$   $\Sigma\text{-prot0}$  challenge-space assms by blast
moreover have lossless-spmf  $(\text{Ass0 } x0 (a0, e0, z0) (a0, e0', z0'))$ 
using 2 3 4 Ass-OR-def True  $\Sigma\text{-prot0}$   $\Sigma 0.\Sigma\text{-protocol-def}$   $\Sigma 0.\text{special-soundness-def}$  challenge-space-def e by blast
ultimately have  $\forall w' \in \text{set-spmf } (\text{Ass-OR } (x0,x1) ((a0,a1),s,(e0,z0), e1,z1) ((a0,a1),s',(e0',z0'), e1',z1')). ((x0,x1), w') \in \text{Rel-OR}$ 
apply (auto simp only: Ass-OR-alt-def True)
apply (auto simp add: o-def Ass-OR-def)
using assms valid-pub-OR-def by blast
moreover have lossless-spmf  $(\text{Ass-OR } (x0,x1) ((a0,a1),s,(e0,z0), e1,z1) ((a0,a1),s',(e0',z0'), e1',z1'))$ 
apply (simp add: Ass-OR-def)
using 2 3 4 True  $\Sigma\text{-prot0}$   $\Sigma 0.\Sigma\text{-protocol-def}$   $\Sigma 0.\text{special-soundness-def}$  challenge-space e by blast
ultimately show ?thesis by simp
next

```



**case** *False*  
**hence**  $e1\text{-neq-}e1'$ :  $e1 \neq e1'$  **using** *e-or* **by** *simp*  
**moreover** **have**  $2$ :  $x1 \in \text{valid-pub1}$   
**using** *valid-pub-OR valid-pub-OR-def* **by** *simp*  
**moreover** **have**  $3$ :  $\text{check1 } x1 \ a1 \ e1 \ z1$   
**using** *assms check-OR-def* **by** *simp*  
**moreover** **have**  $4$ :  $\text{check1 } x1 \ a1 \ e1' \ z1'$   
**using** *assms check-OR-def* **by** *simp*  
**moreover** **have**  $e$ :  $e1 \in \text{carrier } L \ e1' \in \text{carrier } L$   
**using** *challenge-space-def check check' check-OR-def* **by** *auto*  
**ultimately** **have**  $(\forall w' \in \text{set-spmf } (\text{Ass1 } x1 \ (a1, e1, z1) \ (a1, e1', z1')). (x1, w') \in \text{Rel1})$   
**using** *False  $\Sigma 1$ . $\Sigma$ -protocol-def  $\Sigma 1$ .special-soundness-def  $\Sigma$ -prot1 e1-neq-e1' challenge-space* **by** *blast*  
**hence**  $\forall w' \in \text{set-spmf } (\text{Ass-OR } (x0, x1) \ ((a0, a1), s, (e0, z0), e1, z1) \ ((a0, a1), s', (e0', z0'), e1', z1'))$ .  $((x0, x1), w') \in \text{Rel-OR}$   
**apply**(*auto simp add: o-def Ass-OR-def*)  
**using** *False assms  $\Sigma 1$ .L-def assms valid-pub-OR-def* **by** *auto*  
**moreover** **have**  $\text{lossless-spmf } (\text{Ass-OR } (x0, x1) \ ((a0, a1), s, (e0, z0), e1, z1) \ ((a0, a1), s', (e0', z0'), e1', z1'))$   
**apply**(*simp add: Ass-OR-def*)  
**using**  $2 \ 3 \ 4 \ \Sigma\text{-prot1 } \Sigma 1$ . $\Sigma$ -protocol-def  $\Sigma 1$ .special-soundness-def *False e1-neq-e1' challenge-space e* **by** *blast*  
**ultimately** **show** *?thesis* **by** *simp*  
**qed**  
**qed**

**lemma** *special-soundness*:  
**shows**  $\Sigma$ -OR.special-soundness  
**unfolding**  $\Sigma$ -OR.special-soundness-def  
**using** *ss prod.collapse* **by** *fastforce*

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

**proof** –  
**have**  $x \in \text{carrier } L \longrightarrow e = (e \oplus x) \oplus x$  **for**  $x$   
**using** *e-in-carrier xor-assoc* **by** *simp*  
**hence** *?lhs* = *do* {  
 $(r0, ab0) \leftarrow \text{init0 } x0 \ w0$ ;  
 $eb' \leftarrow \text{spmf-of-set } (\text{carrier } L)$ ;  
 $(ab1', eb1'', zb1') \leftarrow \Sigma 1.S \ x1 \ eb'$ ;  
 $\text{let } eb = e \oplus eb'$ ;  
 $zb0 \leftarrow \text{response0 } r0 \ w0 \ eb$ ;  
 $\text{return-spmf } ((\text{check0 } x0 \ ab0 \ eb \ zb0 \ \wedge \ \text{check1 } x1 \ ab1' \ eb' \ zb1'))$ ;  
**by**(*simp add:  $\Sigma$ -OR.completeness-game-def split-def Let-def challenge-space-def*)

```

assms check-OR-def cong: bind-spmf-cong-simp)
also have ... = do {
  eb' ← spmf-of-set (carrier L);
  (ab1', eb1'', zb1') ←  $\Sigma 1.S$  x1 eb';
  let eb = e  $\oplus$  eb';
  (r0, ab0) ← init0 x0 w0;
  zb0 ← response0 r0 w0 eb;
  return-spmf ((check0 x0 ab0 eb zb0  $\wedge$  check1 x1 ab1' eb' zb1'))}
apply(simp add: Let-def split-def)
apply(rewrite bind-commute-spmf)
apply(rewrite bind-commute-spmf[of -  $\Sigma 1.S$  - ])
by simp
also have ... = do {
  eb' :: 'e ← spmf-of-set (carrier L);
  (ab1', eb1'', zb1') ←  $\Sigma 1.S$  x1 eb';
  return-spmf (check1 x1 ab1' eb' zb1')}
apply(simp add: Let-def)
apply(intro bind-spmf-cong; clarsimp?)
subgoal for e' a e z
  apply(cases check1 x1 a e' z)
using  $\Sigma 0$ .complete-game-return-true  $\Sigma$ -prot0  $\Sigma 0$ .completeness-game-def  $\Sigma 0$ . $\Sigma$ -protocol-def

  by(auto simp add: assms bind-spmf-const lossless-init lossless-response lossless-weight-spmfD split-def cong: bind-spmf-cong-simp)
done
also have ... = do {
  eb' :: 'e ← spmf-of-set (carrier L);
  (ab1', eb1'', zb1') ←  $\Sigma 1.S$  x1 eb';
  return-spmf (True)}
apply(intro bind-spmf-cong; clarsimp?)
subgoal for x a aa b
  using  $\Sigma$ -prot1
  apply(auto simp add:  $\Sigma 1.S$ -def split-def image-def  $\Sigma 1.HVZK$ -unfold2-alt)
  using  $\Sigma 1.S$ -def split-def image-def  $\Sigma 1.HVZK$ -unfold2-alt  $\Sigma$ -prot1 valid-pub)
by blast
done
ultimately show ?thesis
  using  $\Sigma 1.HVZK$ -unfold2-alt
  by(simp add: bind-spmf-const Let-def  $\Sigma 1.HVZK$ -unfold2-alt split-def lossless- $\Sigma$ -S lossless-weight-spmfD carrier-L-not-empty finite-L)
qed

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)
proof –

```

```

have  $x1\text{-in}L$ :  $x1 \in \Sigma1.L$ 
  using  $\Sigma1.L\text{-def}$   $rel1$  by auto
have  $x \in \text{carrier } L \longrightarrow e = x \oplus e \oplus x$  for  $x$ 
  by (simp add: e-in-carrier xor-assoc xor-commute local.xor-ac(3))
hence  $?lhs = do \{$ 
  ( $r1, ab1$ )  $\leftarrow \text{init1 } x1 w1$ ;
   $eb' \leftarrow \text{spmf-of-set } (\text{carrier } L)$ ;
  ( $ab0', eb0'', zb0'$ )  $\leftarrow \Sigma0.S x0 eb'$ ;
   $\text{let } eb = e \oplus eb'$ ;
   $zb1 \leftarrow \text{response1 } r1 w1 eb$ ;
   $\text{return-spmf } (\text{check0 } x0 ab0' eb' zb0' \wedge \text{check1 } x1 ab1 eb zb1)$ 
by(simp add:  $\Sigma$ -OR.completeness-game-def split-def Let-def assms challenge-space-def
check-OR-def cong: bind-spmf-cong-simp)
also have  $\dots = do \{$ 
   $eb' \leftarrow \text{spmf-of-set } (\text{carrier } L)$ ;
  ( $ab0', eb0'', zb0'$ )  $\leftarrow \Sigma0.S x0 eb'$ ;
   $\text{let } eb = e \oplus eb'$ ;
  ( $r1, ab1$ )  $\leftarrow \text{init1 } x1 w1$ ;
   $zb1 \leftarrow \text{response1 } r1 w1 eb$ ;
   $\text{return-spmf } (\text{check0 } x0 ab0' eb' zb0' \wedge \text{check1 } x1 ab1 eb zb1)$ 
apply(simp add: Let-def split-def)
apply(rewrite bind-commute-spmf)
apply(rewrite bind-commute-spmf[of -  $\Sigma0.S$  - -])
by simp
also have  $\dots = do \{$ 
   $eb' \leftarrow \text{spmf-of-set } (\text{carrier } L)$ ;
  ( $ab0', eb0'', zb0'$ )  $\leftarrow \Sigma0.S x0 eb'$ ;
   $\text{return-spmf } (\text{check0 } x0 ab0' eb' zb0')$ 
apply(simp add: Let-def)
apply(intro bind-spmf-cong; clarsimp?)
subgoal for  $e' a e z$ 
  apply(cases check0 x0 a e' z)
using  $\Sigma1.complete-game-return-true \Sigma\text{-prot1 } \Sigma1.completeness-game-def \Sigma1.\Sigma\text{-protocol-def}$ 
by(auto simp add:  $x1\text{-in}L$  assms bind-spmf-const lossless-init lossless-response
lossless-weight-spmfD split-def)
done
also have  $\dots = do \{$ 
   $eb' \leftarrow \text{spmf-of-set } (\text{carrier } L)$ ;
  ( $ab0', eb0'', zb0'$ )  $\leftarrow \Sigma0.S x0 eb'$ ;
   $\text{return-spmf } (True)$ 
apply(intro bind-spmf-cong; clarsimp?)
subgoal for  $x a aa b$ 
  using  $\Sigma\text{-prot0}$ 
by(auto simp add: valid-pub valid-pub-OR-def  $\Sigma0.S\text{-def}$  split-def image-def
 $\Sigma0.HVZK\text{-unfold2-alt}$ )
done
ultimately show  $?thesis$ 
apply(simp add:  $\Sigma0.HVZK\text{-unfold2}$  Let-def)
using  $\Sigma0.complete-game-return-true \Sigma\text{-OR.completeness-game-def}$ 

```

**by**(*simp add: bind-spmf-const split-def lossless- $\Sigma$ -S(2) lossless-weight-spmfD*  
*Let-def carrier-L-not-empty finite-L*)

**qed**

**lemma** *completeness'*:

**assumes** *Rel-OR-asm:  $((x0,x1), w) \in Rel-OR$*

**shows**  $\forall e \in carrier L. spmf (\Sigma-OR.completeness-game (x0,x1) w e) True = 1$

**proof**

**fix** *e*

**assume** *asm:  $e \in carrier L$*

**hence**  $(\Sigma-OR.completeness-game (x0,x1) w e) = return-spmf True$

**proof**(*cases w*)

**case** *inl: (Inl a)*

**then show** *?thesis*

**using** *asm correct0 assms inl by auto*

**next**

**case** *inr: (Inr b)*

**then show** *?thesis*

**using** *asm correct1 assms inr by auto*

**qed**

**thus**  $spmf (\Sigma-OR.completeness-game (x0,x1) w e) True = 1$

**by** *simp*

**qed**

**lemma** *completeness: shows  $\Sigma-OR.completeness$*

**unfolding**  *$\Sigma-OR.completeness-def$*

**using** *completeness' challenge-space-def by auto*

**lemma**  *$\Sigma$ -protocol: shows  $\Sigma-OR.\Sigma$ -protocol*

**by**(*simp add: completeness HVZK special-soundness  $\Sigma-OR.\Sigma$ -protocol-def*)

**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*

**by** *unfold-locales (auto simp add:  $\Sigma$ -protocol lossless-G lossless-init-OR G-Rel-OR*  
*lossless-response-OR)*

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

**using** *OR- $\Sigma$ -commit.commit-correct by simp*

**lemma** *OR- $\Sigma$ -commit.abstract-com.perfect-hiding-ind-cpa A*

**using** *OR- $\Sigma$ -commit.perfect-hiding by blast*

**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 A)*

**using** *OR- $\Sigma$ -commit.bind-advantage by simp*

**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.