Σ-protocols and Commitment Schemes

David Butler, Andreas Lochbihler

March 19, 2025

Abstract

We use CryptHOL [2] to formalise commitment schemes and Σ-protocols. Both are widely used fundamental two party cryptographic primitives. Security for commitment schemes is considered using game-based definitions whereas the security of Σ-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 Σ-protocols as well as a construction that allows for compound (AND and OR) Σ-protocols and the Pedersen and Rivest commitment schemes. We also prove that commitment schemes can be constructed from Σ-protocols. We formalise this proof at an abstract level, only assuming the existence of a Σ-protocol, consequently the instantiations of this result for the concrete Σ-protocols we consider come for free.

Contents

1 Commitment Schemes **2**
1.1 Defining security 2
1.2 Pedersen Commitment Scheme 13
1.3 Rivest Commitment Scheme 17

2 Σ-Protocols **19**
2.1 Defining Σ-protocols 20
2.2 Commitments from Σ-protocols 23
2.3 Schnorr Σ-protocol 25
2.4 Chaum-Pedersen Σ-protocol 29
2.5 Okamoto Σ-protocol 34
2.6 Σ-AND statements 44
2.7 Σ-OR statements 47

1 Commitment Schemes

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

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

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

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

1.1 Defining security

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

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

type-synonym $('vk', 'plain', 'commit', 'state)$ *hid-adv* =
$('vk' \Rightarrow (('plain' \times 'plain') \times 'state)$ *spmf*)
$\times$ $('commit' \Rightarrow 'state \Rightarrow$ *bool spmf*)

type-synonym *'commit' hid* = *'commit'* $\Rightarrow$ *bool spmf*

type-synonym $('ck', 'plain', 'commit', 'opening')$ *bind-adversary* =
$'ck' \Rightarrow ('commit' \times 'plain' \times 'opening' \times 'plain' \times 'opening')$ *spmf*

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

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

and *valid-msg* :: *'plain* ⇒ *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* ⇒ *bool*
 where *lossless* $\mathcal{A}$ ⟷
 ((∀ *pk*. *lossless-spmf* (*fst* $\mathcal{A}$ *pk*)) ∧
 (∀ *commit* σ. *lossless-spmf* (*snd* $\mathcal{A}$ *commit* σ)))

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

definition *correct-game* :: *'plain* ⇒ *bool spmf*
 where *correct-game m* = *do* {
 (*ck*, *vk*) ← *key-gen*;
 (*c*,*d*) ← *commit ck m*;
 return-spmf (*verify vk m c d*)}

lemma ⟦ *lossless-spmf key-gen*; *lossless-spmf TI*;
 ⋀*pk m*. *valid-msg m* ⟹ *lossless-spmf* (*commit pk m*) ⟧
 ⟹ *valid-msg m* ⟹ *lossless-spmf* (*correct-game m*)
 ⟨*proof*⟩

definition *correct* **where** *correct* ≡ (∀ *m*. *valid-msg m* ⟶ *spmf* (*correct-game m*) *True* = *1*)

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

primrec *hiding-game-ind-cpa* :: (*'vk*, *'plain*, *'commit*, *'state*) *hid-adv* ⇒ *bool spmf*
 where *hiding-game-ind-cpa* ($\mathcal{A}$*1*, $\mathcal{A}$*2*) = *TRY do* {
 (*ck*, *vk*) ← *key-gen*;
 ((*m0*, *m1*), σ) ← $\mathcal{A}$*1 vk*;
 - :: *unit* ← *assert-spmf* (*valid-msg m0* ∧ *valid-msg m1*);
 b ← *coin-spmf*;
 (*c*,*d*) ← *commit ck* (*if b then m0 else m1*);
 b' :: *bool* ← $\mathcal{A}$*2 c* σ;
 return-spmf (*b'* = *b*)} *ELSE coin-spmf*

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

lemma *lossless-hiding-game*:
 ⟦ *lossless* $\mathcal{A}$; *lossless-spmf key-gen*;
 ⋀*pk plain*. *valid-msg plain* ⟹ *lossless-spmf* (*commit pk plain*) ⟧
 ⟹ *lossless-spmf* (*hiding-game-ind-cpa* $\mathcal{A}$)
 ⟨*proof*⟩

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

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

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

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

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

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

end

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

context *cyclic-group* **begin**

lemma *generator-pow-order*: $\mathbf{g}$ [^] *order G* = **1**
⟨*proof*⟩

lemma *generator-pow-mult-order*: $\mathbf{g}$ [^] (*order G* ∗ *order G*) = **1**
⟨*proof*⟩

lemma *pow-generator-mod*: $\mathbf{g}$ [^] (*k mod order G*) = $\mathbf{g}$ [^] *k*
⟨*proof*⟩

lemma *pow-carrier-mod*:
assumes *g* ∈ *carrier G*
shows *g* [^] (*k mod order G*) = *g* [^] *k*
⟨*proof*⟩

lemma *pow-generator-mod-int*: $\mathbf{g}$ [^] ((*k*::*int*) *mod order G*) = $\mathbf{g}$ [^] *k*
⟨*proof*⟩

lemma *pow-generator-eq-iff-cong*:
finite (*carrier G*) ⟹ $\mathbf{g}$ [^] *x* = $\mathbf{g}$ [^] *y* ⟷ [*x* = *y*] (*mod order G*)
⟨*proof*⟩

lemma *power-distrib*:
assumes *h* ∈ *carrier G*
shows $\mathbf{g}$ [^] (*e* :: *nat*) ⊗ *h* [^] *e* = ($\mathbf{g}$ ⊗ *h*) [^] *e*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *neg-power-inverse*:
assumes *g* ∈ *carrier G*
and *x* < *order G*
shows *g* [^] (*order G* − (*x* :: *nat*)) = *inv* (*g* [^] *x*)
⟨*proof*⟩

lemma *int-nat-pow*: **assumes** *a* ≥ *0* **shows** ($\mathbf{g}$ [^] (*int* (*a* ::*nat*))) [^] (*b*::*int*) = $\mathbf{g}$ [^] (*a*∗*b*)
⟨*proof*⟩

lemma *pow-gen-mod-mult*:
shows($\mathbf{g}$ [^] (*a*::*nat*) ⊗ $\mathbf{g}$ [^] (*b*::*nat*)) [^] ((*c*::*int*)∗*int* (*d*::*nat*)) = ($\mathbf{g}$ [^] *a* ⊗ $\mathbf{g}$ [^] *b*) [^] ((*c*∗*int d*) *mod* (*order G*))
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

lemma *group-eq-pow-eq-mod*:
fixes *a b* :: *nat*
assumes **g** [^] *a* = **g** [^] *b*
and *order G* > *0*
shows [*a* = *b*] (*mod order G*)
⟨*proof*⟩

end

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

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

type-synonym *'grp' dislog-adv* = *'grp'* ⇒ *nat spmf*

type-synonym *'grp' dislog-adv'* = *'grp'* ⇒ (*nat* × *nat*) *spmf*

type-synonym *'grp' dislog-adv2* = *'grp'* × *'grp'* ⇒ *nat spmf*

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

definition *advantage* :: *'grp dislog-adv* ⇒ *real*
where *advantage* $\mathcal{A}$ ≡ *spmf* (*dis-log* $\mathcal{A}$) *True*

lemma *lossless-dis-log*: ⟦*0* < *order* $\mathcal{G}$; ∀ *h*. *lossless-spmf* ($\mathcal{A}$ *h*)⟧ ⟹ *lossless-spmf* (*dis-log* $\mathcal{A}$)
⟨*proof*⟩

end

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

begin

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

definition $g' = \mathbf{g}$ [⌉] x

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

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

definition *adversary2* :: (*'grp* ⇒ (*nat* × *nat*) *spmf*) ⇒ *'grp* ⇒ *nat spmf*
  where *adversary2* $\mathcal{A}$ h = *do* {
    ($w1$,$w2$) ← $\mathcal{A}$ h;
    return-spmf ($w1$ + x ∗ $w2$)}

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

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

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

end

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

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

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

end

end
theory *Number-Theory-Aux* **imports**
HOL−Number-Theory.Cong
HOL−Number-Theory.Residues
begin

abbreviation *inverse* **where** *inverse x q* ≡ (*fst* (*bezw x q*))

lemma *inverse*: **assumes** *gcd x q* = *1*
shows [*x* ∗ *inverse x q* = *1*] (*mod q*)
⟨*proof*⟩

lemma *prod-not-prime*:
assumes *prime* (*x*::*nat*)
and *prime y*
and *x* > *2*
and *y* > *2*
shows ¬ *prime* ((*x*−*1*)∗(*y*−*1*))
⟨*proof*⟩

lemma *ex-inverse*:
assumes *coprime*: *coprime* (*e* :: *nat*) ((*P*−*1*)∗(*Q*−*1*))
and *prime P*
and *prime Q*
and *P* ≠ *Q*
shows ∃ *d*. [*e*∗*d* = *1*] (*mod* (*P*−*1*)) ∧ *d* ≠ *0*
⟨*proof*⟩

lemma *ex-k1-k2*:
assumes *coprime*: *coprime* (*e* :: *nat*) ((*P*−*1*)∗(*Q*−*1*))
and [*e*∗*d* = *1*] (*mod* (*P*−*1*))
shows ∃ *k1 k2*. *e*∗*d* + *k1*∗(*P*−*1*) = *1* + *k2*∗(*P*−*1*)
⟨*proof*⟩
lemma *a* > *b* ⟹ *int a* − *int b* = *int* (*a* − *b*)
⟨*proof*⟩

lemma *ex-k-mod*:
assumes *coprime*: *coprime* (*e* :: *nat*) ((*P*−*1*)∗(*Q*−*1*))
and *P* ≠ *Q*
and *prime P*
and *prime Q*
and *d* ≠ *0*
and [*e*∗*d* = *1*] (*mod* (*P*−*1*))
shows ∃ *k*. *e*∗*d* = *1* + *k*∗(*P*−*1*)
⟨*proof*⟩

lemma *fermat-little-theorem*:
  assumes *prime* (P :: *nat*)
  shows $[x \hat{} P = x]$ (*mod* P)
⟨*proof*⟩

lemma *prime-field*:
  assumes *prime* (q::*nat*)
    and $a < q$
    and $a \neq 0$
  shows *coprime* a q
  ⟨*proof*⟩

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) = $\{..< q\} - \{0\}$
  ⟨*proof*⟩

lemma *lossless-sample-uniform-units*:
  assumes (p::*nat*) > 1
  shows *lossless-spmf* (*sample-uniform-units* p)
  ⟨*proof*⟩

lemma *weight-sample-uniform-units*:
  assumes (p::*nat*) > 1
  shows *weight-spmf* (*sample-uniform-units* p) = 1
  ⟨*proof*⟩

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*⟩

lemma *one-time-pad*:
  assumes *inj-on*: *inj-on* f $\{..<q\}$
    and *sur*: f ‘ $\{..<q\}$ = $\{..<q\}$
  shows *map-spmf* f (*sample-uniform* q) = (*sample-uniform* q)
(**is** *?lhs* = *?rhs*)

$\langle proof \rangle$

lemma *plus-inj-eq*:
  assumes *x*: $x < q$
    and *x'*: $x' < q$
    and *map*: $((y :: nat) + x)$ *mod* $q = (y + x')$ *mod* q
shows $x = x'$
$\langle proof \rangle$

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

lemma *surj-uni-samp-plus*:
  assumes *inj*: *inj-on* $(\lambda(b :: nat).\ (y + b)$ *mod* $q\ )\ \{..<q\}$
  shows $(\lambda(b :: nat).\ (y + b)$ *mod* $q)$ ‘ $\{..< q\} = \{..< q\}$
  $\langle proof \rangle$

lemma *samp-uni-plus-one-time-pad*:
shows *map-spmf* $(\lambda b.\ (y + b)$ *mod* $q)$ (*sample-uniform* q) = *sample-uniform* q
  $\langle proof \rangle$

lemma *mult-inj-eq*:
  assumes *coprime*: *coprime* x $(q{::}nat)$
    and *y*: $y < q$
    and *y'*: $y' < q$
    and *map*: $x * y$ *mod* $q = x * y'$ *mod* q
  shows $y = y'$
$\langle proof \rangle$

lemma *inj-on-mult*:
  assumes *coprime*: *coprime* x $(q{::}nat)$
  shows *inj-on* $(\lambda\ b.\ x{*}b$ *mod* $q)\ \{..<q\}$
  $\langle proof \rangle$

lemma *surj-on-mult*:
  assumes *coprime*: *coprime* x $(q{::}nat)$
    and *inj*: *inj-on* $(\lambda\ b.\ x{*}b$ *mod* $q)\ \{..<q\}$
  shows $(\lambda\ b.\ x{*}b$ *mod* $q)$ ‘ $\{..< q\} = \{..< q\}$
  $\langle proof \rangle$

lemma *mult-one-time-pad*:
  assumes *coprime*: *coprime* x q
  shows *map-spmf* $(\lambda\ b.\ x{*}b$ *mod* $q)$ (*sample-uniform* q) = *sample-uniform* q
  $\langle proof \rangle$

lemma *inj-on-mult'*:
  assumes *coprime*: *coprime* x (q::*nat*)
  shows *inj-on* (λ b. $x*b$ *mod* q) ({..<q} − {*0*})
  ⟨*proof*⟩

lemma *surj-on-mult'*:
  assumes *coprime*: *coprime* x (q::*nat*)
    and *inj*: *inj-on* (λ b. $x*b$ *mod* q) ({..<q} − {*0*})
  shows (λ b. $x*b$ *mod* q) ‘ ({..<q} − {*0*}) = ({..<q} − {*0*})
⟨*proof*⟩

lemma *mult-one-time-pad'*:
  assumes *coprime*: *coprime* x q
  shows *map-spmf* (λ b. $x*b$ *mod* q) (*sample-uniform-units* q) = *sample-uniform-units* q
  ⟨*proof*⟩

lemma *samp-uni-add-mult*:
  assumes *coprime*: *coprime* x (q::*nat*)
    and x': $x' < q$
    and y': $y' < q$
    and *map*: $(y + x * x')$ *mod* q = $(y + x * y')$ *mod* q
  shows $x' = y'$
⟨*proof*⟩

lemma *inj-on-add-mult*:
  assumes *coprime*: *coprime* x (q::*nat*)
  shows *inj-on* (λ b. (y + $x*b$) *mod* q) {..<q}
  ⟨*proof*⟩

lemma *surj-on-add-mult*:
  assumes *coprime*: *coprime* x (q::*nat*)
    and *inj*: *inj-on* (λ b. (y + $x*b$) *mod* q) {..<q}
  shows (λ b. (y + $x*b$) *mod* q) ‘ {..< q} = {..< q}
  ⟨*proof*⟩

lemma *add-mult-one-time-pad*:
  assumes *coprime*: *coprime* x q
  shows *map-spmf* (λ b. (y + $x*b$) *mod* q) (*sample-uniform* q) = (*sample-uniform* q)
  ⟨*proof*⟩

lemma *inj-on-minus*: *inj-on* (λ(b :: *nat*). (y + (q − b)) *mod* q) {..<q}
⟨*proof*⟩

lemma *surj-on-minus*:
  assumes *inj*: *inj-on* $(\lambda(b :: nat).\ (y + (q - b))\ mod\ q\ )\ \{..<q\}$
  shows $(\lambda(b :: nat).\ (y + (q - b))\ mod\ q)$ ‘ $\{..< q\} = \{..< q\}$
  ⟨*proof*⟩

lemma *samp-uni-minus-one-time-pad*:
 shows *map-spmf*$(\lambda\ b.\ (y + (q - b))\ mod\ q)$ (*sample-uniform* q) = *sample-uniform* q
  ⟨*proof*⟩

lemma *not-coin-spmf*: *map-spmf* $(\lambda\ a.\ \neg\ a)$ *coin-spmf* = *coin-spmf*
⟨*proof*⟩

lemma *xor-uni-samp*: *map-spmf*$(\lambda\ b.\ y \oplus b)$ (*coin-spmf*) = *map-spmf*$(\lambda\ b.\ b)$ (*coin-spmf*)
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *ped-inv-mapping*:
  assumes $(a{::}nat) < q$
    and $[m \neq 0]\ (mod\ q)$
 shows *map-spmf* $(\lambda\ d.\ (d + a * (m{::}nat))\ mod\ q)$ (*sample-uniform* q) = *map-spmf* $(\lambda\ d.\ (d + q * m - a * m)\ mod\ q)$ (*sample-uniform* q)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

1.2 Pedersen Commitment Scheme

The Pedersen commitment scheme [8] is a commitment scheme based on a cyclic group. We use the construction of cyclic groups from CryptHOL to formalise the commitment scheme. We prove perfect hiding and computational binding, with a reduction to the discrete log problem. We a proof of the Pedersen commitment scheme is realised in the instantiation of the Schnorr Σ-protocol with the general construction of commitment schemes from Σ-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 Σ-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 𝒢 :: 'grp cyclic-group (structure)
  assumes prime-order: prime (order 𝒢)
begin

lemma order-gt-0 [simp]: order 𝒢 > 0
  ⟨proof⟩

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

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

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

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

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

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

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

definition *dis-log-*$\mathcal{A}$ :: (*'grp ck*, *plain*, *'grp commit*, *opening*) *bind-adversary* ⇒ *'grp ck* ⇒ *nat spmf*
where *dis-log-*$\mathcal{A}$ $\mathcal{A}$ *h* = *do* {
  (*c*, *m*, *d*, *m'*, *d'*) ← $\mathcal{A}$ *h*;
  - :: *unit* ← *assert-spmf* (*m* ≠ *m'* ∧ *valid-msg m* ∧ *valid-msg m'*);
  - :: *unit* ← *assert-spmf* (*c* = **g** [^] *d* ⊗ *h* [^] *m* ∧ *c* = **g** [^] *d'* ⊗ *h* [^] *m'*);
  return-spmf (*if* (*m* > *m'*) *then* (*nat* ((*int d'* − *int d*) ∗ *inverse* (*m* − *m'*) (*order* $\mathcal{G}$) *mod order* $\mathcal{G}$)) *else*
    (*nat* ((*int d* − *int d'*) ∗ *inverse* (*m'* − *m*) (*order* $\mathcal{G}$) *mod order* $\mathcal{G}$)))}

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

sublocale *discrete-log*: *dis-log* -
  ⟨*proof*⟩

end

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

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

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

lemma *dis-log-break'*:
  assumes *y* < *order* $\mathcal{G}$
    and ¬ *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*⟩

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

⟨*proof*⟩

lemma *correct*:
shows *spmf* (*ped-commit.correct-game* *m*) *True* = *1*
⟨*proof*⟩

theorem *abstract-correct*:
shows *ped-commit.correct*
⟨*proof*⟩

lemma *perfect-hiding*:
shows *spmf* (*ped-commit.hiding-game-ind-cpa* $\mathcal{A}$) *True* − *1/2* = *0*
including *monad-normalisation*
⟨*proof*⟩

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

lemma *bind-output-cong*:
assumes $x <$ *order* $\mathcal{G}$
shows (x = *nat* ((*int* *b* − *int* *ab*) ∗ *fst* (*bezw* (*aa* − *ac*) (*order* $\mathcal{G}$)) *mod* *int* (*order* $\mathcal{G}$)))
⟷ [x = *nat* ((*int* *b* − *int* *ab*) ∗ *fst* (*bezw* (*aa* − *ac*) (*order* $\mathcal{G}$)) *mod* *int* (*order* $\mathcal{G}$))] (*mod* *order* $\mathcal{G}$)
⟨*proof*⟩

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

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

end

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

sublocale *pedersen* $\mathcal{G}$ η **for** η ⟨*proof*⟩

theorem *pedersen-correct-asym*:
shows *ped-commit.correct* *n*
⟨*proof*⟩

theorem *pedersen-perfect-hiding-asym*:

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

theorem *pedersen-bind-asym*:
shows *negligible* (λ n. *ped-commit.bind-advantage* n ($\mathcal{A}$ n))
$\longleftrightarrow$ *negligible* (λ n. *discrete-log.advantage* n (*dis-log-*$\mathcal{A}$ n ($\mathcal{A}$ n)))
⟨*proof*⟩

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*]: $q > 0$
⟨*proof*⟩

type-synonym *ck* = *nat* × *nat*
type-synonym *vk* = *nat* × *nat*
type-synonym *plain* = *nat*
type-synonym *commit* = *nat*
type-synonym *opening* = *nat* × *nat*

definition *key-gen* :: (*ck* × *vk*) *spmf*
where
key-gen = *do* {

$a :: nat \leftarrow sample\text{-}uniform\ q;$
$b :: nat \leftarrow sample\text{-}uniform\ q;$
$x1 :: nat \leftarrow sample\text{-}uniform\ q;$
$let\ y1 = (a * x1 + b)\ mod\ q;$
$return\text{-}spmf\ ((a,b), (x1,y1))\}$

definition *commit* :: $ck \Rightarrow plain \Rightarrow (commit \times opening)\ spmf$
where
$commit\ ck\ m = do\ \{$
$let\ (a,b) = ck;$
$return\text{-}spmf\ ((m + a)\ mod\ q, (a,b))\}$

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

sublocale *rivest-commit*: *abstract-commitment key-gen commit verify valid-msg*
$\langle proof \rangle$

lemma *abstract-correct*: *rivest-commit.correct*
$\langle proof \rangle$

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

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

lemma *samp-uni-break'*:
assumes $fst\text{-}cond$: $m \neq m' \wedge valid\text{-}msg\ m \wedge valid\text{-}msg\ m'$
and c: $c = (m + a)\ mod\ q \wedge y1 = (a * x1 + b)\ mod\ q$
and c': $c = (m' + a')\ mod\ q \wedge y1 = (a' * x1 + b')\ mod\ q$
and $x1$: $x1 < q$
shows $x1 = (if\ (a\ mod\ q > a'\ mod\ q)\ then\ nat\ ((int\ b' - int\ b) * (inverse\ (nat\ ((int\ a\ mod\ q - int\ a'\ mod\ q)\ mod\ q))\ q)\ mod\ q)\ \ else$
$nat\ ((int\ b - int\ b') * (inverse\ (nat\ ((int\ a'\ mod\ q - int\ a\ mod\ q)\ mod\ q))\ q)\ mod\ q))$
$\langle proof \rangle$

lemma *samp-uni-spmf-mod-q*:
shows $spmf\ (sample\text{-}uniform\ q)\ (x\ mod\ q) = 1/q$
$\langle proof \rangle$

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

lemma *bind-game-le-inv-q*:
shows *spmf* (*rivest-commit.bind-game* $\mathcal{A}$) *True* ≤ *1* / *q*
⟨*proof*⟩
including *monad-normalisation*
⟨*proof*⟩

lemma *rivest-bind*:
shows *rivest-commit.bind-advantage* $\mathcal{A}$ ≤ *1* / *q*
⟨*proof*⟩

end

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

sublocale *rivest q* η **for** η ⟨*proof*⟩

theorem *rivest-correct*:
shows *rivest-commit.correct n*
⟨*proof*⟩

theorem *rivest-perfect-hiding-asym*:
assumes *lossless*-$\mathcal{A}$: *rivest-commit.lossless* ($\mathcal{A}$ *n*)
shows *rivest-commit.perfect-hiding-ind-cpa n* ($\mathcal{A}$ *n*)
⟨*proof*⟩

theorem *rivest-binding-asym*:
assumes *negligible* (λ*n*. *1* / (*q n*))
shows *negligible* (λ*n*. *rivest-commit.bind-advantage n* ($\mathcal{A}$ *n*))
⟨*proof*⟩

end

end

2 Σ-Protocols

Σ-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 Σ-protocols and the construction of compound AND and OR statements.

In this section we define Σ-protocols then provide a general proof that they can be used to construct commitment schemes. Defining security for Σ-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 follow the simulation-based formalisation of [1] and [4].

2.1 Defining Σ-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 Σ*-protocols-base* =
fixes *init* :: *'pub-input* ⇒ *'witness* ⇒ (*'rand* × *'msg*) *spmf* — initial message in Σ-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 Σ protocol is considered over
and *S-raw* :: *'pub-input* ⇒ *'challenge* ⇒ (*'msg*, *'response*) *sim-out spmf* — Simulator for the HVZK property
and $\mathcal{A}$*ss* :: (*'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* ⊆ *valid-pub*
begin

lemma assumes *x* ∈ *Domain Rel* **shows** ∃ *w*. (*x*,*w*) ∈ *Rel*
⟨*proof*⟩

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

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

The first property of Σ-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* ≡ (∀ *h w e* . (*h*,*w*) ∈ *Rel* ⟶ *e* ∈ *challenge-space* ⟶ *spmf* (*completeness-game h w e*) *True* = *1*)

Second we consider the honest verifier zero knowledge property (HVZK). To reason about this we construct the real view of the Σ-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* = *map-spmf* (λ (*a*, *z*). (*a*, *e*, *z*)) (*S-raw h e*)

lemma *lossless-S-raw-imp-lossless-S*: *lossless-spmf* (*S-raw h e*) ⟶ *lossless-spmf* (*S h e*)
  ⟨*proof*⟩

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

definition *HVZK* ≡ (∀ *e* ∈ *challenge-space*.
    (∀ (*h*, *w*)∈*Rel*. *R h w e* = *S h e*)
    ∧ (∀ *h* ∈ *valid-pub*. ∀ (*a*, *z*) ∈ *set-spmf* (*S-raw h e*). *check h a e z*))

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

definition *special-soundness* ≡ (∀ *h e e′ a z z′*. *h* ∈ *valid-pub* ⟶ *e* ∈ *challenge-space* ⟶ *e′* ∈ *challenge-space* ⟶ *e* ≠ *e′*
    ⟶ *check h a e z* ⟶ *check h a e′ z′* ⟶ (*lossless-spmf* ($\mathcal{A}$*ss h* (*a*,*e*,*z*) (*a*, *e′*, *z′*)) ∧
    (∀ *w′*∈*set-spmf* ($\mathcal{A}$*ss h* (*a*,*e*,*z*) (*a*,*e′*,*z′*)). (*h*,*w′*) ∈ *Rel*)))

lemma *special-soundness-alt*:
special-soundness $\longleftrightarrow$
$(\forall\ h\ a\ e\ z\ e'\ z'.\ e \in \mathit{challenge\text{-}space} \longrightarrow e' \in \mathit{challenge\text{-}space} \longrightarrow h \in \mathit{valid\text{-}pub}$

$\longrightarrow e \neq e' \longrightarrow \mathit{check}\ h\ a\ e\ z \wedge \mathit{check}\ h\ a\ e'\ z'$
$\longrightarrow \mathit{bind\text{-}spmf}\ (\mathcal{A}\mathit{ss}\ h\ (a,e,z)\ (a,e',z'))\ (\lambda\ w'.\ \mathit{return\text{-}spmf}\ ((h,w') \in \mathit{Rel})) = \mathit{return\text{-}spmf}\ \mathit{True})$
$\langle \mathit{proof} \rangle$

definition Σ-*protocol* $\equiv$ *completeness* $\wedge$ *special-soundness* $\wedge$ *HVZK*

General lemmas

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

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

lemma *HVZK-unfold1*:
assumes Σ-*protocol*
shows $\forall\ h\ w\ e.\ (h,w) \in \mathit{Rel} \longrightarrow e \in \mathit{challenge\text{-}space} \longrightarrow R\ h\ w\ e = S\ h\ e$
$\langle \mathit{proof} \rangle$

lemma *HVZK-unfold2*:
assumes Σ-*protocol*
shows $\forall\ h\ e\ \mathit{out}.\ e \in \mathit{challenge\text{-}space} \longrightarrow h \in \mathit{valid\text{-}pub} \longrightarrow \mathit{out} \in \mathit{set\text{-}spmf}\ (\mathit{S\text{-}raw}\ h\ e) \longrightarrow \mathit{check}\ h\ (\mathit{fst}\ \mathit{out})\ e\ (\mathit{snd}\ \mathit{out})$
$\langle \mathit{proof} \rangle$

lemma *HVZK-unfold2-alt*:
assumes Σ-*protocol*
shows $\forall\ h\ a\ e\ z.\ e \in \mathit{challenge\text{-}space} \longrightarrow h \in \mathit{valid\text{-}pub} \longrightarrow (a,z) \in \mathit{set\text{-}spmf}\ (\mathit{S\text{-}raw}\ h\ e) \longrightarrow \mathit{check}\ h\ a\ e\ z$
$\langle \mathit{proof} \rangle$

end

2.2 Commitments from Σ-protocols

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

locale *Σ-protocols-to-commitments* = *Σ-protocols-base init response check Rel S-raw*
$\mathcal{A}$*ss challenge-space valid-pub*
  for *init* :: *'pub-input* ⇒ *'witness* ⇒ (*'rand* × *'msg*) *spmf*
    and *response* :: *'rand* ⇒ *'witness* ⇒ *'challenge* ⇒ *'response spmf*
    and *check* :: *'pub-input* ⇒ *'msg* ⇒ *'challenge* ⇒ *'response* ⇒ *bool*
    and *Rel* :: (*'pub-input* × *'witness*) *set*
    and *S-raw* :: *'pub-input* ⇒ *'challenge* ⇒ (*'msg*, *'response*) *sim-out spmf*
    and $\mathcal{A}$*ss* :: (*'pub-input*, *'msg*, *'challenge*, *'response*, *'witness*) *prover-adversary*
    and *challenge-space* :: *'challenge 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*) ∈ *set-spmf G* ⟹ (*h*,*w*) ∈ *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*) ∈ *set-spmf G* ⟹ *h* ∈ *Domain Rel*
  ⟨*proof*⟩

lemma *set-spmf-G-L* [*simp*]: (*h*,*w*) ∈ *set-spmf G* ⟹ *h* ∈ *L*
  ⟨*proof*⟩

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

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

definition *rel-advantage* :: (*'pub-input* ⇒ *'witness spmf*) ⇒ *real*
  where *rel-advantage* $\mathcal{A}$ ≡ *spmf* (*rel-game* $\mathcal{A}$) *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* {

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

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

definition *verify* :: ($'$*pub-input* $\times$ $'$*witness*) $\Rightarrow$ $'$*challenge* $\Rightarrow$ $'$*msg* $\Rightarrow$ $'$*response* $\Rightarrow$ *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 = ($m \in$ *challenge-space*)

Showing the construction of a commitment scheme from a Σ-protocol is a valid commitment scheme is trivial.

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

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

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

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

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

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

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

end

end

2.3 Schnorr Σ-protocol

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

theory *Schnorr-Sigma-Commit* **imports**
Commitment-Schemes
Sigma-Protocols
Cyclic-Group-Ext
Discrete-Log
Number-Theory-Aux
Uniform-Sampling
HOL−Number-Theory.Cong
begin

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

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

The types for the Σ-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* × *witness*) *set*
where *R-DL* = {(*h*, *w*). *h* = **g** [^] *w*}

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

lemma *lossless-init*: *lossless-spmf* (*init h w*)
⟨*proof*⟩

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

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

definition *G* :: (*′grp pub-in* × *witness*) *spmf*
where *G* = *do* {
w ← *sample-uniform* (*order* $\mathcal{G}$);
return-spmf (**g** [^] *w*, *w*)}

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

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

definition *check* :: *′grp pub-in* ⇒ *′grp msg* ⇒ *challenge* ⇒ *response* ⇒ *bool*
where *check h a e z* = (*a* ⊗ (*h* [^] *e*) = **g** [^] *z* ∧ *a* ∈ *carrier* $\mathcal{G}$)

definition *S2* :: *′grp* ⇒ *challenge* ⇒ (*′grp msg*, *response*) *sim-out spmf*
where *S2 h e* = *do* {
c ← *sample-uniform* (*order* $\mathcal{G}$);
let a = **g** [^] *c* ⊗ (*inv* (*h* [^] *e*));
return-spmf (*a*, *c*)}

definition *ss-adversary* :: *′grp* ⇒ (*′grp msg*, *challenge*, *response*) *conv-tuple* ⇒ (*′grp msg*, *challenge*, *response*) *conv-tuple* ⇒ *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 Σ-protocol use Schnorr to construct a commitment scheme.

type-synonym *′grp′ ck* = *′grp′*
type-synonym *′grp′ vk* = *′grp′* × *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-*$\mathcal{A}$ :: (*′grp ck*, *plain*, *′grp commit*, *opening*) *bind-adversary* ⇒ *′grp ck* ⇒ *nat spmf*
where *dis-log-*$\mathcal{A}$ $\mathcal{A}$ *h* = *do* {

$(c,\ e,\ z,\ e',\ z') \leftarrow \mathcal{A}\ h;$
- :: *unit* $\leftarrow$ *assert-spmf* $(e > e' \wedge \neg\ [e = e']\ (\textit{mod order } \mathcal{G}) \wedge (\textit{gcd } (e - e')\ (\textit{order } \mathcal{G}) = 1) \wedge c \in \textit{carrier } \mathcal{G});$
- :: *unit* $\leftarrow$ *assert-spmf* $(((c \otimes h\ [\widehat{\ }]\ e) = \mathbf{g}\ [\widehat{\ }]\ z) \wedge (c \otimes h\ [\widehat{\ }]\ e') = \mathbf{g}\ [\widehat{\ }]\ z');$
return-spmf $(\textit{nat } ((\textit{int } z - \textit{int } z') * \textit{inverse } ((e - e'))\ (\textit{order } \mathcal{G})\ \textit{mod order } \mathcal{G}))\}$

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

end

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

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

The Schnorr Σ-protocol is complete.

lemma *completeness*: *Schnorr-*Σ*.completeness*
⟨*proof*⟩

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

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

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

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

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

⟨*proof*⟩

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

lemma *honest-verifier-ZK*:
  shows *Schnorr-Σ.HVZK*
  ⟨*proof*⟩

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

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

The special soundness property for the Schnorr Σ-protocol.

lemma *special-soundness*:
  shows *Schnorr-Σ.special-soundness*
  ⟨*proof*⟩

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

theorem *sigma-protocol*:
  shows *Schnorr-Σ.Σ-protocol*
  ⟨*proof*⟩

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

sublocale *Schnorr-Σ-commit*: *Σ-protocols-to-commitments init response check R-DL S2 ss-adversary challenge-space valid-pub G*
  ⟨*proof*⟩

lemma *Schnorr-Σ-commit.abstract-com.correct*
  ⟨*proof*⟩

lemma *Schnorr-Σ-commit.abstract-com.perfect-hiding-ind-cpa* $\mathcal{A}$
  ⟨*proof*⟩

lemma *rel-adv-eq-dis-log-adv*:
$\quad$ *Schnorr-Σ-commit.rel-advantage* $\mathcal{A}$ = *discrete-log.advantage* $\mathcal{A}$
$\langle proof \rangle$

lemma *bind-advantage-bound-dis-log*:
Schnorr-Σ-commit.abstract-com.bind-advantage $\mathcal{A} \leq$ *discrete-log.advantage* (*Schnorr-Σ-commit.adversary* $\mathcal{A}$)
$\langle proof \rangle$

end

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

sublocale *schnorr-sigma-protocol* $\mathcal{G}$ η **for** η
$\langle proof \rangle$

The Σ-protocol statement comes easily in the asymptotic setting.

theorem *sigma-protocol*:
shows *Schnorr-Σ.Σ-protocol n*
$\langle proof \rangle$

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

lemma *asymp-correct*: *Schnorr-Σ-commit.abstract-com.correct n*
$\langle proof \rangle$

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

lemma *asymp-computational-binding*:
assumes *negligible* (λ *n*. *discrete-log.advantage n* (*Schnorr-Σ-commit.adversary n* ($\mathcal{A}$ *n*)))
shows *negligible* (λ *n*. *Schnorr-Σ-commit.abstract-com.bind-advantage n* ($\mathcal{A}$ *n*))
$\langle proof \rangle$

end

end

2.4 Chaum-Pedersen Σ-protocol

The Chaum-Pedersen Σ-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-Σ-base* =
fixes $\mathcal{G}$ :: *'grp cyclic-group* (**structure**)
and x :: *nat*
assumes *prime-order*: *prime* (*order* $\mathcal{G}$)
begin

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

lemma *or-gt-1*: *order* $\mathcal{G} > 1$
⟨*proof*⟩

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

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* {
$w \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
return-spmf (($\mathbf{g}\ [\widehat{\ }]\ w$, $g'\ [\widehat{\ }]\ w$), w)}

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

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

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

lemma *lossless-init*: *lossless-spmf* (*init h w*)
⟨*proof*⟩

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

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

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

definition *R* :: (*'grp pub-in* × *witness*) *set*
  where *R* = {(*h*, *w*). (*fst h* = **g** [^] *w* ∧ *snd h* = *g'* [^] *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* $\mathcal{G}$));
  let a = **g** [^] *z* ⊗ *inv* (*h* [^] *c*);
  let a' = *g'* [^] *z* ⊗ *inv* (*h'* [^] *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-Σ* = *chaum-ped-Σ-base* + *cyclic-group* $\mathcal{G}$
begin

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

sublocale *chaum-ped-sigma*: *Σ-protocols-base init response check R S2 ss-adversary challenge-space valid-pub*
  ⟨*proof*⟩

lemma *completeness*:
  shows *chaum-ped-sigma.completeness*
⟨*proof*⟩

lemma *hvzk-xr'-rewrite*:

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

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

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

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*
$\langle proof \rangle$

lemma *HVZK*:
shows *chaum-ped-sigma.HVZK*
$\langle proof \rangle$

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

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 $a \otimes fst\ h\ [\widehat{\ }]\ e = \mathbf{g}\ [\widehat{\ }]\ z$
and $b \otimes snd\ h\ [\widehat{\ }]\ e = g'\ [\widehat{\ }]\ z$

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

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

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

theorem *Σ-protocol*: *chaum-ped-sigma.Σ-protocol*
$\langle \mathit{proof} \rangle$

sublocale *chaum-ped-Σ-commit*: *Σ-protocols-to-commitments init response check R S2 ss-adversary challenge-space valid-pub G*
$\langle \mathit{proof} \rangle$

sublocale *dis-log*: *dis-log* $\mathcal{G}$
$\langle \mathit{proof} \rangle$

sublocale *dis-log-alt*: *dis-log-alt* $\mathcal{G}\ x$
$\langle \mathit{proof} \rangle$

lemma *reduction-to-dis-log*:
shows *chaum-ped-Σ-commit.rel-advantage* $\mathcal{A}$ = *dis-log.advantage* (*dis-log-alt.adversary3* $\mathcal{A}$)
$\langle \mathit{proof} \rangle$

lemma *commitment-correct*: *chaum-ped-Σ-commit.abstract-com.correct*
$\langle \mathit{proof} \rangle$

lemma *chaum-ped-Σ-commit.abstract-com.perfect-hiding-ind-cpa* $\mathcal{A}$
$\langle \mathit{proof} \rangle$

lemma *binding*: *chaum-ped-Σ-commit.abstract-com.bind-advantage* $\mathcal{A} \leq$ *dis-log.advantage* (*dis-log-alt.adversary3* ((*chaum-ped-Σ-commit.adversary* $\mathcal{A}$)))
$\langle \mathit{proof} \rangle$

end

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

sublocale *chaum-ped-*Σ $\mathcal{G}$ η **for** η
  ⟨*proof*⟩

The Σ-protocol statement comes easily in the asympotic setting.

theorem *sigma-protocol*:
  shows *chaum-ped-sigma.*Σ*-protocol n*
  ⟨*proof*⟩

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

lemma *asymp-correct*: *chaum-ped-*Σ*-commit.abstract-com.correct n*
  ⟨*proof*⟩

lemma *asymp-perfect-hiding*: *chaum-ped-*Σ*-commit.abstract-com.perfect-hiding-ind-cpa n* ($\mathcal{A}$ *n*)
  ⟨*proof*⟩

lemma *asymp-computational-binding*:
  assumes *negligible* (λ *n*. *dis-log.advantage n* (*dis-log-alt.adversary3 n* ((*chaum-ped-*Σ*-commit.adversary n* ($\mathcal{A}$ *n*)))))
  shows *negligible* (λ *n*. *chaum-ped-*Σ*-commit.abstract-com.bind-advantage n* ($\mathcal{A}$ *n*))
  ⟨*proof*⟩

end

end

2.5 Okamoto Σ-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 $g' = \mathbf{g}$ [^] *x*

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

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

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

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

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

definition *init* :: *'grp pub-in* ⇒ *witness* ⇒ (*rand* × *'grp msg*) *spmf*
  where *init y w* = *do* {
    let (*x1*,*x2*) = *w*;
    r1 ← *sample-uniform* (*order* $\mathcal{G}$);
    r2 ← *sample-uniform* (*order* $\mathcal{G}$);
    return-spmf ((*r1*,*r2*), **g** [^] *r1* ⊗ *g'* [^] *r2*)}

lemma *lossless-init*: *lossless-spmf* (*init h w*)
  ⟨*proof*⟩

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

definition *R* :: (*'grp pub-in* × *witness*) *set*
  where *R* ≡ {(*h*, *w*). (*h* = **g** [^] (*fst w*) ⊗ *g'* [^] (*snd w*))}

definition *G* :: (*'grp pub-in* × *witness*) *spmf*

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

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

lemma $lossless\text{-}G$: $lossless\text{-}spmf\ G$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma Σ-*protocols-base.R h w c* = *R2 h w c*
⟨*proof*⟩

lemma *completeness*:
shows Σ-*protocols-base.completeness*
⟨*proof*⟩

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

lemma *hvzk-z1-r1-tuple-rewrite*:
assumes *r1*: *r1* < *order* $\mathcal{G}$
shows (**g** [^] *r1* ⊗ *g*′ [^] *r2*, *c*, (*r1* + *c* ∗ *x1*) *mod order* $\mathcal{G}$, (*r2* + *c* ∗ *x2*) *mod order* $\mathcal{G}$) =
(**g** [^] (((*r1* + *c* ∗ *x1*) *mod order* $\mathcal{G}$ + *order* $\mathcal{G}$ ∗ *c* ∗ *x1* − *c* ∗ *x1*) *mod order* $\mathcal{G}$)
⊗ *g*′ [^] *r2*, *c*, (*r1* + *c* ∗ *x1*) *mod order* $\mathcal{G}$, (*r2* + *c* ∗ *x2*) *mod order* $\mathcal{G}$)
⟨*proof*⟩

lemma *hvzk-z2-r2-tuple-rewrite*:
assumes *xb* < *order* $\mathcal{G}$
shows (**g** [^] (((*x*′ + *xa* ∗ *x1*) *mod order* $\mathcal{G}$ + *order* $\mathcal{G}$ ∗ *xa* ∗ *x1* − *xa* ∗ *x1*) *mod order* $\mathcal{G}$)
⊗ *g*′ [^] *xb*, *xa*, (*x*′ + *xa* ∗ *x1*) *mod order* $\mathcal{G}$, (*xb* + *xa* ∗ *x2*) *mod order* $\mathcal{G}$) =
(**g** [^] (((*x*′ + *xa* ∗ *x1*) *mod order* $\mathcal{G}$ + *order* $\mathcal{G}$ ∗ *xa* ∗ *x1* − *xa* ∗ *x1*) *mod order* $\mathcal{G}$)
⊗ *g*′ [^] (((*xb* + *xa* ∗ *x2*) *mod order* $\mathcal{G}$ + *order* $\mathcal{G}$ ∗ *xa* ∗ *x2* − *xa* ∗ *x2*) *mod order* $\mathcal{G}$), *xa*, (*x*′ + *xa* ∗ *x1*) *mod order* $\mathcal{G}$, (*xb* + *xa* ∗ *x2*) *mod order* $\mathcal{G}$)
⟨*proof*⟩

lemma *hvzk-sim-inverse-rewrite*:
assumes *h*: *h* = **g** [^] (*x1* :: *nat*) ⊗ *g*′ [^] (*x2* :: *nat*)
shows g [^] (((*z1*::*nat*) + *order* $\mathcal{G}$ ∗ *c* ∗ *x1* − *c* ∗ *x1*) *mod* (*order* $\mathcal{G}$))
⊗ *g*′ [^] (((*z2*::*nat*) + *order* $\mathcal{G}$ ∗ *c* ∗ *x2* − *c* ∗ *x2*) *mod* (*order* $\mathcal{G}$))
= (**g** [^] *z1* ⊗ *g*′ [^] *z2*) ⊗ (*inv h* [^] *c*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *hv-zk*:

assumes $h = \ \mathbf{g} \ [\char94] \ x1 \otimes g' \ [\char94] \ x2$
shows Σ*-protocols-base.R h (x1,x2) c* = Σ*-protocols-base.S h c*
including *monad-normalisation*
⟨*proof*⟩

lemma *HVZK*:
shows Σ*-protocols-base.HVZK*
⟨*proof*⟩

lemma *ss-rewrite*:
assumes $h \in carrier\ \mathcal{G}$
and $a \in carrier\ \mathcal{G}$
and $e < order\ \mathcal{G}$
and $\mathbf{g}\ [\char94]\ z1 \otimes g'\ [\char94]\ z1' = a \otimes h\ [\char94]\ e$
and $e' < e$
and $\mathbf{g}\ [\char94]\ z2 \otimes g'\ [\char94]\ z2' = a \otimes h\ [\char94]\ e'$
shows $h = \mathbf{g}\ [\char94]\ ((int\ z1 - int\ z2) * fst\ (bezw\ (e - e')\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G})) \otimes g'\ [\char94]\ ((int\ z1' - int\ z2') * fst\ (bezw\ (e - e')\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G}))$
⟨*proof*⟩

lemma
assumes *h-mem*: $h \in carrier\ \mathcal{G}$
and *a-mem*: $a \in carrier\ \mathcal{G}$
and *a*: $\mathbf{g}\ [\char94]\ fst\ z \otimes g'\ [\char94]\ snd\ z = a \otimes h\ [\char94]\ e$
and *a'*: $\mathbf{g}\ [\char94]\ fst\ z' \otimes g'\ [\char94]\ snd\ z' = a \otimes h\ [\char94]\ e'$
and *e-e'-mod*: $e'\ mod\ order\ \mathcal{G} < e\ mod\ order\ \mathcal{G}$
shows $h = \mathbf{g}\ [\char94]\ ((int\ (fst\ z) - int\ (fst\ z')) * fst\ (bezw\ ((e\ mod\ order\ \mathcal{G} - e'\ mod\ order\ \mathcal{G})\ mod\ order\ \mathcal{G})\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G}))$
$\otimes g'\ [\char94]\ ((int\ (snd\ z) - int\ (snd\ z')) * fst\ (bezw\ ((e\ mod\ order\ \mathcal{G} - e'\ mod\ order\ \mathcal{G})\ mod\ order\ \mathcal{G})\ (order\ \mathcal{G}))\ mod\ int\ (order\ \mathcal{G}))$
⟨*proof*⟩

lemma *special-soundness*:
shows Σ*-protocols-base.special-soundness*
⟨*proof*⟩

theorem Σ*-protocol*:
shows Σ*-protocols-base.*Σ*-protocol*
⟨*proof*⟩

sublocale *okamoto-*Σ*-commit*: Σ*-protocols-to-commitments init response check R S2 ss-adversary challenge-space valid-pub G*
⟨*proof*⟩

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

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

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

lemma *commitment-correct*: *okamoto-Σ-commit.abstract-com.correct*
⟨*proof*⟩

lemma *okamoto-Σ-commit.abstract-com.perfect-hiding-ind-cpa* $\mathcal{A}$
⟨*proof*⟩

lemma *binding*:
shows *okamoto-Σ-commit.abstract-com.bind-advantage* $\mathcal{A}$
≤ *dis-log.advantage* (*dis-log-alt.adversary2* (*okamoto-Σ-commit.adversary* $\mathcal{A}$))
⟨*proof*⟩

end

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

sublocale *okamoto* $\mathcal{G}$ η **for** η
⟨*proof*⟩

The Σ-protocol statement comes easily in the asympotic setting.

theorem *sigma-protocol*:
shows *Σ-protocols-base.Σ-protocol n*
⟨*proof*⟩

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

lemma *asymp-correct*: *okamoto-Σ-commit.abstract-com.correct n*
⟨*proof*⟩

lemma *asymp-perfect-hiding*: *okamoto-Σ-commit.abstract-com.perfect-hiding-ind-cpa n* ($\mathcal{A}$ *n*)
⟨*proof*⟩

lemma *asymp-computational-binding*:
assumes *negligible* (λ *n*. *dis-log.advantage n* (*dis-log-alt.adversary2* (*okamoto-Σ-commit.adversary n* ($\mathcal{A}$ *n*))))

shows *negligible* (λ *n*. *okamoto-Σ-commit.abstract-com.bind-advantage* *n* ($\mathcal{A}$ *n*))
⟨*proof*⟩

end

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

unbundle *no lattice-syntax*

context *bounded-lattice* **begin**

lemma *top-join* [*simp*]: $x \in$ *carrier* $L \Longrightarrow \top \sqcup x = \top$
⟨*proof*⟩

lemma *join-top* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcup \top = \top$
⟨*proof*⟩

lemma *bot-join* [*simp*]: $x \in$ *carrier* $L \Longrightarrow \bot \sqcup x = x$
⟨*proof*⟩

lemma *join-bot* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcup \bot = x$
⟨*proof*⟩

lemma *bot-meet* [*simp*]: $x \in$ *carrier* $L \Longrightarrow \bot \sqcap x = \bot$
⟨*proof*⟩

lemma *meet-bot* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcap \bot = \bot$
⟨*proof*⟩

lemma *top-meet* [*simp*]: $x \in$ *carrier* $L \Longrightarrow \top \sqcap x = x$
⟨*proof*⟩

lemma *meet-top* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcap \top = x$
⟨*proof*⟩

lemma *join-idem* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcup x = x$
⟨*proof*⟩

lemma *meet-idem* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcap x = x$
⟨*proof*⟩

lemma *meet-leftcomm*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$ **if** $x \in$ *carrier* L $y \in$ *carrier* L $z \in$ *carrier* L
⟨*proof*⟩

lemma *join-leftcomm*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$ **if** $x \in$ *carrier* L $y \in$ *carrier* L $z \in$ *carrier* L
⟨*proof*⟩

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

end

record *'a boolean-algebra* = *'a gorder* +
compl :: *'a* ⇒ *'a* (‹−ı› *1000*)

definition *xor* :: (*'a*, *'b*) *boolean-algebra-scheme* ⇒ *'a* ⇒ *'a* ⇒ *'a* (**infixr** ‹⊕ı› *100*)
where
$x \oplus y = (x \sqcup y) \sqcap (-\ (x \sqcap y))$ **for** L (**structure**)

locale *boolean-algebra* = *bounded-lattice* L
for L (**structure**) +
assumes *compl-closed* [*intro*, *simp*]: $x \in$ *carrier* $L \Longrightarrow -\ x \in$ *carrier* L
and *meet-compl-bot* [*simp*]: $x \in$ *carrier* $L \Longrightarrow -\ x \sqcap x = \bot$
and *join-compl-top* [*simp*]: $x \in$ *carrier* $L \Longrightarrow -\ x \sqcup x = \top$
and *join-meet-distrib1*: ⟦ $x \in$ *carrier* L; $y \in$ *carrier* L; $z \in$ *carrier* L ⟧ $\Longrightarrow x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
begin

lemma *join-meet-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
if $x \in$ *carrier* L $y \in$ *carrier* L $z \in$ *carrier* L
⟨*proof*⟩

lemma *meet-join-distrib1*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
if [*simp*]: $x \in$ *carrier* L $y \in$ *carrier* L $z \in$ *carrier* L
⟨*proof*⟩

lemma *meet-join-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
if [*simp*]: $x \in$ *carrier* L $y \in$ *carrier* L $z \in$ *carrier* L
⟨*proof*⟩

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

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

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

lemma *meet-compl2-bot* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcap -\ x = \bot$
⟨*proof*⟩

lemma *join-compl2-top* [*simp*]: $x \in$ *carrier* $L \Longrightarrow x \sqcup -\ x = \top$
⟨*proof*⟩

lemma *compl-unique*:
  assumes $x \sqcap y = \bot$
    and $x \sqcup y = \top$
    and [*simp*]: $x \in$ *carrier* L $y \in$ *carrier* L
  shows $- x = y$
⟨*proof*⟩

lemma *double-compl* [*simp*]: $- (- x) = x$ **if** [*simp*]: $x \in$ *carrier* L
  ⟨*proof*⟩

lemma *compl-eq-compl-iff* [*simp*]: $- x = - y \longleftrightarrow x = y$ **if** $x \in$ *carrier* L $y \in$ *carrier* L
  ⟨*proof*⟩

lemma *compl-bot-eq* [*simp*]: $- \bot = \top$
  ⟨*proof*⟩

lemma *compl-top-eq* [*simp*]: $- \top = \bot$
  ⟨*proof*⟩

lemma *compl-inf* [*simp*]: $- (x \sqcap y) = - x \sqcup - y$ **if** [*simp*]: $x \in$ *carrier* L $y \in$ *carrier* L
⟨*proof*⟩

lemma *compl-sup* [*simp*]: $- (x \sqcup y) = - x \sqcap - y$ **if** $x \in$ *carrier* L $y \in$ *carrier* L
  ⟨*proof*⟩

lemma *compl-mono*:
  assumes $x \sqsubseteq y$
    and $x \in$ *carrier* L $y \in$ *carrier* L
  shows $- y \sqsubseteq - x$
  ⟨*proof*⟩

lemma *compl-le-compl-iff* [*simp*]: $- x \sqsubseteq - y \longleftrightarrow y \sqsubseteq x$ **if** $x \in$ *carrier* L $y \in$ *carrier* L
  ⟨*proof*⟩

lemma *compl-le-swap1*:
  assumes $y \sqsubseteq - x$ $x \in$ *carrier* L $y \in$ *carrier* L
  shows $x \sqsubseteq - y$
  ⟨*proof*⟩

lemma *compl-le-swap2*:
  assumes $- y \sqsubseteq x$ $x \in$ *carrier* L $y \in$ *carrier* L
  shows $- x \sqsubseteq y$
  ⟨*proof*⟩

lemma *join-compl-top-left1* [*simp*]: $- x \sqcup (x \sqcup y) = \top$ **if** [*simp*]: $x \in$ *carrier* L $y \in$ *carrier* L

⟨*proof*⟩

lemma *join-compl-top-left2* [*simp*]: $x \sqcup (-\ x \sqcup y) = \top$ **if** [*simp*]: $x \in$ *carrier L* $y \in$ *carrier L*
⟨*proof*⟩

lemma *meet-compl-bot-left1* [*simp*]: $-\ x \sqcap (x \sqcap y) = \bot$ **if** [*simp*]: $x \in$ *carrier L* $y \in$ *carrier L*
⟨*proof*⟩

lemma *meet-compl-bot-left2* [*simp*]: $x \sqcap (-\ x \sqcap y) = \bot$ **if** [*simp*]: $x \in$ *carrier L* $y \in$ *carrier L*
⟨*proof*⟩

lemma *meet-compl-bot-right* [*simp*]: $x \sqcap (y \sqcap -\ x) = \bot$ **if** [*simp*]: $x \in$ *carrier L* $y \in$ *carrier L*
⟨*proof*⟩

lemma *xor-closed* [*intro*, *simp*]: $[\![\ x \in$ *carrier L*; $y \in$ *carrier L* $]\!] \Longrightarrow x \oplus y \in$ *carrier L*
⟨*proof*⟩

lemma *xor-comm*: $[\![\ x \in$ *carrier L*; $y \in$ *carrier L* $]\!] \Longrightarrow x \oplus y = y \oplus x$
⟨*proof*⟩

lemma *xor-assoc*: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
if [*simp*]: $x \in$ *carrier L* $y \in$ *carrier L* $z \in$ *carrier L*
⟨*proof*⟩

lemma *xor-left-comm*: $x \oplus (y \oplus z) = y \oplus (x \oplus z)$ **if** $x \in$ *carrier L* $y \in$ *carrier L* $z \in$ *carrier L*
⟨*proof*⟩

lemma [*simp*]:
assumes $x \in$ *carrier L*
shows *xor-bot*: $x \oplus \bot = x$
and *bot-xor*: $\bot \oplus x = x$
and *xor-top*: $x \oplus \top = -\ x$
and *top-xor*: $\top \oplus x = -\ x$
⟨*proof*⟩

lemma *xor-inverse* [*simp*]: $x \oplus x = \bot$ **if** $x \in$ *carrier L*
⟨*proof*⟩

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

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

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

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

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

end

end

2.6 Σ-AND statements

theory *Sigma-AND* **imports**
Sigma-Protocols
Xor
begin

locale *Σ-AND-base* = *Σ0*: *Σ-protocols-base init0 response0 check0 Rel0 S0-raw* $\mathcal{A}$*ss0 carrier L valid-pub0*
+ *Σ1*: *Σ-protocols-base init1 response1 check1 Rel1 S1-raw* $\mathcal{A}$*ss1 carrier L valid-pub1*
for *init1* :: *'pub1* ⇒ *'witness1* ⇒ (*'rand1* × *'msg1*) *spmf*
and *response1* :: *'rand1* ⇒ *'witness1* ⇒ *'bool* ⇒ *'response1 spmf*
and *check1* :: *'pub1* ⇒ *'msg1* ⇒ *'bool* ⇒ *'response1* ⇒ *bool*
and *Rel1* :: (*'pub1* × *'witness1*) *set*
and *S1-raw* :: *'pub1* ⇒ *'bool* ⇒ (*'msg1* × *'response1*) *spmf*
and $\mathcal{A}$*ss1* :: *'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 $\mathcal{A}$*ss0* :: *'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-$\mathcal{A}$ss*: *lossless-spmf* ($\mathcal{A}$*ss0 x0* (*a0*,*e*,*z0*) (*a0*,*e'*,*z0'*)) *lossless-spmf* ($\mathcal{A}$*ss1 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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 $\mathcal{A}ss\text{-}AND$ :: $'pub0 \times 'pub1 \Rightarrow ('msg0 \times 'msg1) \times 'bool \times 'response0 \times 'response1 \Rightarrow ('msg0 \times 'msg1) \times 'bool \times 'response0 \times 'response1 \Rightarrow ('witness0 \times 'witness1)$ *spmf*
where $\mathcal{A}ss\text{-}AND\ (x0,x1)\ ((a0,a1),\ e,\ (z0,z1))\ ((a0',a1'),\ e',\ (z0',z1')) = do\ \{$
$w0 :: 'witness0 \leftarrow \mathcal{A}ss0\ x0\ (a0,e,z0)\ (a0',e',z0')$;
$w1 \leftarrow \mathcal{A}ss1\ x1\ (a1,e,z1)\ (a1',e',z1')$;
return-spmf $(w0,w1)\}$

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

sublocale Σ-*AND*: Σ-*protocols-base init-AND response-AND check-AND Rel-AND S-AND* $\mathcal{A}$*ss-AND challenge-space valid-pub-AND*
⟨*proof*⟩

end

locale Σ-*AND* = Σ-*AND-base* +
assumes *set-spmf-G-L*: $((x0,\ x1),\ w0,\ w1) \in$ *set-spmf G* $\Longrightarrow ((x0,\ x1),\ (w0,w1)) \in$ *Rel-AND*
begin

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

lemma *HVZK*: Σ-*AND.HVZK*
⟨*proof*⟩

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

lemma *completeness*: Σ-*AND.completeness*
⟨*proof*⟩

lemma *ss*:
assumes *e-neq-e'*: $s \neq s'$
and *valid-pub*: $(x0,x1) \in$ *valid-pub-AND*
and *challenge-space*: $s \in$ *challenge-space* $s' \in$ *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* ($\mathcal{A}$*ss-AND* (*x0*,*x1*) ((*a0*,*a1*), *s*, (*z0*,*z1*)) ((*a0*,*a1*), *s'*, (*z0'*,*z1'*)))
∧ (∀ *w'*∈*set-spmf* ($\mathcal{A}$*ss-AND* (*x0*,*x1*) ((*a0*,*a1*), *s*, (*z0*,*z1*)) ((*a0*,*a1*), *s'*, (*z0'*,*z1'*))). ((*x0*,*x1*), *w'*) ∈ *Rel-AND*)
⟨*proof*⟩

lemma *special-soundness*:
shows Σ-*AND.special-soundness*
⟨*proof*⟩

theorem Σ-*protocol*:
shows Σ-*AND*.Σ-*protocol*
⟨*proof*⟩

sublocale *AND*-Σ-*commit*: Σ-*protocols-to-commitments init-AND response-AND check-AND Rel-AND S-AND* $\mathcal{A}$*ss-AND challenge-space valid-pub-AND G*
⟨*proof*⟩

lemma *AND*-Σ-*commit.abstract-com.correct*
⟨*proof*⟩

lemma *AND*-Σ-*commit.abstract-com.perfect-hiding-ind-cpa* $\mathcal{A}$
⟨*proof*⟩

lemma *bind-advantage-bound-dis-log*:
shows *AND*-Σ-*commit.abstract-com.bind-advantage* $\mathcal{A}$ ≤ *AND*-Σ-*commit.rel-advantage* (*AND*-Σ-*commit.adversary* $\mathcal{A}$)
⟨*proof*⟩

end

end

2.7 Σ-OR statements

theory *Sigma-OR* **imports**
Sigma-Protocols
Xor
begin

locale Σ-*OR-base* = Σ*0*: Σ-*protocols-base init0 response0 check0 Rel0 S0-raw* $\mathcal{A}$*ss0 carrier L valid-pub0*
\+ Σ*1*: Σ-*protocols-base init1 response1 check1 Rel1 S1-raw* $\mathcal{A}$*ss1 carrier L valid-pub1*
for *init1* :: *'pub1* ⇒ *'witness1* ⇒ (*'rand1* × *'msg1*) *spmf*
and *response1* :: *'rand1* ⇒ *'witness1* ⇒ *'bool* ⇒ *'response1 spmf*
and *check1* :: *'pub1* ⇒ *'msg1* ⇒ *'bool* ⇒ *'response1* ⇒ *bool*
and *Rel1* :: (*'pub1* × *'witness1*) *set*
and *S1-raw* :: *'pub1* ⇒ *'bool* ⇒ (*'msg1* × *'response1*) *spmf*
and $\mathcal{A}$*ss1* :: *'pub1* ⇒ *'msg1* × *'bool* × *'response1* ⇒ *'msg1* × *'bool* × *'response1*

$\Rightarrow$ *'witness1 spmf*
and *challenge-space1* :: *'bool set*
and *valid-pub1* :: *'pub1 set*
and *init0* :: *'pub0* $\Rightarrow$ *'witness0* $\Rightarrow$ (*'rand0* $\times$ *'msg0*) *spmf*
and *response0* :: *'rand0* $\Rightarrow$ *'witness0* $\Rightarrow$ *'bool* $\Rightarrow$ *'response0 spmf*
and *check0* :: *'pub0* $\Rightarrow$ *'msg0* $\Rightarrow$ *'bool* $\Rightarrow$ *'response0* $\Rightarrow$ *bool*
and *Rel0* :: (*'pub0* $\times$ *'witness0*) *set*
and *S0-raw* :: *'pub0* $\Rightarrow$ *'bool* $\Rightarrow$ (*'msg0* $\times$ *'response0*) *spmf*
and $\mathcal{A}$*ss0* :: *'pub0* $\Rightarrow$ *'msg0* $\times$ *'bool* $\times$ *'response0* $\Rightarrow$ *'msg0* $\times$ *'bool* $\times$ *'response0* $\Rightarrow$ *'witness0 spmf*
and *challenge-space0* :: *'bool set*
and *valid-pub0* :: *'pub0 set*
and *G* :: ((*'pub0* $\times$ *'pub1*) $\times$ (*'witness0* + *'witness1*)) *spmf*
and *L* :: *'bool boolean-algebra* (**structure**)
+
assumes Σ*-prot1*: Σ*1*.Σ*-protocol*
and Σ*-prot0*: Σ*0*.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* $\neq$ {}
and *lossless-G*: *lossless-spmf G*
begin

inductive-set *Rel-OR* :: ((*'pub0* $\times$ *'pub1*) $\times$ (*'witness0* + *'witness1*)) *set* **where**
Rel-OR-I0: ((*x0*, *x1*), *Inl w0*) $\in$ *Rel-OR* **if** (*x0*, *w0*) $\in$ *Rel0* $\wedge$ *x1* $\in$ *valid-pub1*
| *Rel-OR-I1*: ((*x0*, *x1*), *Inr w1*) $\in$ *Rel-OR* **if** (*x1*, *w1*) $\in$ *Rel1* $\wedge$ *x0* $\in$ *valid-pub0*

inductive-simps *Rel-OR-simps* [*simp*]:
((*x0*, *x1*), *Inl w0*) $\in$ *Rel-OR*
((*x0*, *x1*), *Inr w1*) $\in$ *Rel-OR*

lemma *Domain-Rel-cases*:
assumes (*x0*,*x1*) $\in$ *Domain Rel-OR*
shows ($\exists$ *w0*. (*x0*,*w0*) $\in$ *Rel0* $\wedge$ *x1* $\in$ *valid-pub1*) $\vee$ ($\exists$ *w1*. (*x1*,*w1*) $\in$ *Rel1* $\wedge$ *x0* $\in$ *valid-pub0*)
⟨*proof*⟩

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

definition *challenge-space* = *carrier L*

fun *init-OR* :: (*'pub0* $\times$ *'pub1*) $\Rightarrow$ (*'witness0* + *'witness1*) $\Rightarrow$ ((((*'rand0* $\times$ *'bool* $\times$ *'response1* + *'rand1* $\times$ *'bool* $\times$ *'response0*)) $\times$ *'msg0* $\times$ *'msg1*)) *spmf*
where *init-OR* (*x0*,*x1*) (*Inl w0*) = *do* {

$(r0,a0) \leftarrow init0\ x0\ w0;$
$e1 \leftarrow spmf\text{-}of\text{-}set\ (carrier\ L);$
$(a1,\ e'1,\ z1) \leftarrow \Sigma1.S\ x1\ e1;$
$return\text{-}spmf\ (Inl\ (r0,\ e1,\ z1),\ a0,\ a1)\} \mid$
$init\text{-}OR\ (x0,\ x1)\ (Inr\ w1) = do\ \{$
$(r1,\ a1) \leftarrow init1\ x1\ w1;$
$e0 \leftarrow spmf\text{-}of\text{-}set\ (carrier\ L);$
$(a0,\ e'0,\ z0) \leftarrow \Sigma0.S\ x0\ e0;$
$return\text{-}spmf\ ((Inr\ (r1,\ e0,\ z0),\ a0,\ a1))\}$

lemma *lossless-Σ-S*: $lossless\text{-}spmf\ (\Sigma1.S\ x1\ e1)\ lossless\text{-}spmf\ (\Sigma0.S\ x0\ e0)$
⟨*proof*⟩

lemma *lossless-init-OR*: $lossless\text{-}spmf\ (init\text{-}OR\ (x0,x1)\ w)$
⟨*proof*⟩

fun *response-OR* :: $(('rand0 \times 'bool \times 'response1 + 'rand1 \times 'bool \times 'response0)) \Rightarrow ('witness0 + 'witness1)$
$\Rightarrow 'bool \Rightarrow (('bool \times 'response0) \times ('bool \times 'response1))\ spmf$
where $response\text{-}OR\ (Inl\ (r0\ ,\ e\text{-}1,\ z1))\ (Inl\ w0)\ s = do\ \{$
$let\ e0 = s \oplus e\text{-}1;$
$z0 \leftarrow response0\ r0\ w0\ e0;$
$return\text{-}spmf\ ((e0,z0),\ (e\text{-}1,z1))\} \mid$
$response\text{-}OR\ (Inr\ (r1,\ e\text{-}0,\ z0))\ (Inr\ w1)\ s = do\ \{$
$let\ e1 = s \oplus e\text{-}0;$
$z1 \leftarrow response1\ r1\ w1\ e1;$
$return\text{-}spmf\ ((e\text{-}0,\ z0),\ (e1,\ z1))\}$

definition *check-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 $check\text{-}OR\ (x0,x1)\ (a0,a1)\ s\ ((e0,z0),\ (e1,z1))$
$= (s = e0 \oplus e1$
$\wedge e0 \in challenge\text{-}space \wedge e1 \in challenge\text{-}space$
$\wedge check0\ x0\ a0\ e0\ z0 \wedge check1\ x1\ a1\ e1\ z1)$
⟨*proof*⟩

fun *S-OR* **where** $S\text{-}OR\ (x0,x1)\ c = do\ \{$
$e1 \leftarrow spmf\text{-}of\text{-}set\ (carrier\ L);$
$(a1,\ e1',\ z1) \leftarrow \Sigma1.S\ x1\ e1;$
$let\ e0 = c \oplus e1;$
$(a0,\ e0',\ z0) \leftarrow \Sigma0.S\ x0\ e0;$
$let\ z = ((e0',z0),\ (e1',z1));$
$return\text{-}spmf\ ((a0,\ a1),z)\}$

definition *𝒜ss-OR′* :: *′pub0* × *′pub1* ⇒ (*′msg0* × *′msg1*) × *′bool* × (*′bool* × *′response0*) × *′bool* × *′response1*
⇒ (*′msg0* × *′msg1*) × *′bool* × (*′bool* × *′response0*) × *′bool* × *′response1* ⇒ (*′witness0* + *′witness1*) *spmf*
where *𝒜ss-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 *𝒜ss-OR* :: *′pub0* × *′pub1* ⇒ (*′msg0* × *′msg1*) × *′bool* × (*′bool* × *′response0*) × *′bool* × *′response1*
⇒ (*′msg0* × *′msg1*) × *′bool* × (*′bool* × *′response0*) × *′bool* × *′response1* ⇒ (*′witness0* + *′witness1*) *spmf*
where *𝒜ss-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 *𝒜ss-OR-alt-def*: *𝒜ss-OR* (*x0*,*x1*) ((*a0*,*a1*),*s*,(*e0*,*z0*),*e1*,*z1*) ((*a0*,*a1*),*s′*,(*e0′*,*z0′*),*e1′*,*z1′*) = *do* {
if (*e0* ≠ *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*)}}
⟨*proof*⟩

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

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

end

locale Σ-*OR-proofs* = Σ-*OR-base* + *boolean-algebra L* +
assumes *G-Rel-OR*: ((*x0*, *x1*), *w*) ∈ *set-spmf G* ⟹ ((*x0*, *x1*), *w*) ∈ *Rel-OR*
and *lossless-response-OR*: *lossless-spmf* (*response-OR R W s*)
begin

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

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

lemma *HVZK*:
  shows Σ-*OR.HVZK*
  ⟨*proof*⟩

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

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

lemma *special-soundness*:
  shows Σ-*OR.special-soundness*
  ⟨*proof*⟩

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

lemma *correct1*:
  assumes *rel1*: $(x1,w1) \in Rel1$
    and *valid-pub*: $x0 \in valid\text{-}pub0$
    and *e-in-carrier*: $e \in carrier\ L$
  shows $\Sigma\text{-}OR.completeness\text{-}game\ (x0,x1)\ (Inr\ w1)\ e = return\text{-}spmf\ True$

(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *completeness′*:
assumes *Rel-OR-asm*: $((x0,x1), w) \in$ *Rel-OR*
shows $\forall$ *e* $\in$ *carrier L. spmf* (Σ-*OR.completeness-game* (*x0*,*x1*) *w e*) *True* = *1*
⟨*proof*⟩

lemma *completeness*: **shows** Σ-*OR.completeness*
⟨*proof*⟩

lemma Σ-*protocol*: **shows** Σ-*OR*.Σ-*protocol*
⟨*proof*⟩

sublocale *OR*-Σ-*commit*: Σ-*protocols-to-commitments init-OR response-OR check-OR Rel-OR S-OR* $\mathcal{A}$*ss-OR challenge-space valid-pub-OR G*
⟨*proof*⟩

lemma *OR*-Σ-*commit.abstract-com.correct*
⟨*proof*⟩

lemma *OR*-Σ-*commit.abstract-com.perfect-hiding-ind-cpa* $\mathcal{A}$
⟨*proof*⟩

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

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 Σ-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. Damgard. On Σ-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 thesisPhD 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.