Game-based cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar and Bhargav Bhatt

October 11, 2017

Abstract

In this AFP entry, we show how to specify game-based cryptographic security notions and formally prove secure several cryptographic constructions from the literature using the CryptHOL framework. Among others, we formalise the notions of a random oracle, a pseudo-random function, an unpredictable function, and of encryption schemes that are indistinguishable under chosen plaintext and/or ciphertext attacks. We prove the random-permutation/random-function switching lemma, security of the Elgamal and hashed Elgamal public-key encryption scheme and correctness and security of several constructions with pseudo-random functions.

Our proofs follow the game-hopping style advocated by Shoup [4] and Bellare and Rogaway [2], from which most of the examples have been taken. We generalise some of their results such that they can be reused in other proofs. Thanks to CryptHOL's integration with Isabelle's parametricity infrastructure, many simple hops are easily justified using the theory of representation independence.

Contents

1 Specifying security using games

theory *Diffie-Hellman* **imports**
CryptHOL.Cyclic-Group-SPMF
CryptHOL.Computational-Model
begin

1.1 The DDH game

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

type-synonym *'grp' adversary* = *'grp'* ⇒ *'grp'* ⇒ *'grp'* ⇒ *bool spmf*

definition *ddh-0* :: *'grp adversary* ⇒ *bool spmf*
where *ddh-0* $\mathcal{A}$ = *do* {
  $x \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
  $y \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
  $\mathcal{A}$ (**g** (^) x) (**g** (^) y) (**g** (^) ($x * y$))
}

definition *ddh-1* :: *'grp adversary* ⇒ *bool spmf*
where *ddh-1* $\mathcal{A}$ = *do* {
  $x \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
  $y \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
  $z \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
  $\mathcal{A}$ (**g** (^) x) (**g** (^) y) (**g** (^) z)
}

definition *advantage* :: *'grp adversary* ⇒ *real*
where *advantage* $\mathcal{A}$ = |*spmf* (*ddh-0* $\mathcal{A}$) *True* − *spmf* (*ddh-1* $\mathcal{A}$) *True*|

definition *lossless* :: *'grp adversary* ⇒ *bool*
where *lossless* $\mathcal{A}$ ⟷ ($\forall\, \alpha\ \beta\ \gamma$. *lossless-spmf* ($\mathcal{A}\ \alpha\ \beta\ \gamma$))

lemma *lossless-ddh-0*:
⟦ *lossless* $\mathcal{A}$; *0* < *order* $\mathcal{G}$ ⟧
⟹ *lossless-spmf* (*ddh-0* $\mathcal{A}$)
by(*auto simp add*: *lossless-def ddh-0-def split-def Let-def*)

lemma *lossless-ddh-1*:
⟦ *lossless* $\mathcal{A}$; *0* < *order* $\mathcal{G}$ ⟧
⟹ *lossless-spmf* (*ddh-1* $\mathcal{A}$)
by(*auto simp add*: *lossless-def ddh-1-def split-def Let-def*)

end

1.2 The LCDH game

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

type-synonym *'grp' adversary* = *'grp'* ⇒ *'grp'* ⇒ *'grp' set spmf*

definition *lcdh* :: *'grp adversary* ⇒ *bool spmf*
where *lcdh* $\mathcal{A}$ = *do* {
    x ← *sample-uniform* (*order* $\mathcal{G}$);
    y ← *sample-uniform* (*order* $\mathcal{G}$);
    zs ← $\mathcal{A}$ (**g** (ˆ) *x*) (**g** (ˆ) *y*);
    return-spmf (**g** (ˆ) (*x* ∗ *y*) ∈ *zs*)
  }

definition *advantage* :: *'grp adversary* ⇒ *real*
where *advantage* $\mathcal{A}$ = *spmf* (*lcdh* $\mathcal{A}$) *True*

definition *lossless* :: *'grp adversary* ⇒ *bool*
where *lossless* $\mathcal{A}$ ⟷ (∀ α β. *lossless-spmf* ($\mathcal{A}$ α β))

lemma *lossless-lcdh*:
  ⟦ *lossless* $\mathcal{A}$; *0* < *order* $\mathcal{G}$ ⟧
  ⟹ *lossless-spmf* (*lcdh* $\mathcal{A}$)
by(*auto simp add*: *lossless-def lcdh-def split-def Let-def*)

end

end

theory *IND-CCA2* **imports**
  CryptHOL.Computational-Model
  CryptHOL.Negligible
  CryptHOL.Environment-Functor
begin

locale *pk-enc* =
  fixes *key-gen* :: *security* ⇒ (*'ekey* × *'dkey*) *spmf* — probabilistic
  and *encrypt* :: *security* ⇒ *'ekey* ⇒ *'plain* ⇒ *'cipher spmf* — probabilistic
  and *decrypt* :: *security* ⇒ *'dkey* ⇒ *'cipher* ⇒ *'plain option* — deterministic, but not used
  and *valid-plain* :: *security* ⇒ *'plain* ⇒ *bool* — checks whether a plain text is valid, i.e., has the right format

1.3 The IND-CCA2 game for public-key encryption

We model an IND-CCA2 security game in the multi-user setting as described in [1].

locale *ind-cca2* = *pk-enc* +
  constrains *key-gen* :: *security* ⇒ (*'ekey* × *'dkey*) *spmf*
  and *encrypt* :: *security* ⇒ *'ekey* ⇒ *'plain* ⇒ *'cipher spmf*
  and *decrypt* :: *security* ⇒ *'dkey* ⇒ *'cipher* ⇒ *'plain option*
  and *valid-plain* :: *security* ⇒ *'plain* ⇒ *bool*
begin

type-synonym (*'ekey'*, *'dkey'*, *'cipher'*) *state-oracle* = (*'ekey'* × *'dkey'* × *'cipher'* *list*) *option*

fun *decrypt-oracle*
  :: *security* ⇒ (*'ekey*, *'dkey*, *'cipher*) *state-oracle* ⇒ *'cipher*
  ⇒ (*'plain option* × (*'ekey*, *'dkey*, *'cipher*) *state-oracle*) *spmf*
where
  decrypt-oracle η *None cipher* = *return-spmf* (*None*, *None*)
| *decrypt-oracle* η (*Some* (*ekey*, *dkey*, *cstars*)) *cipher* = *return-spmf*
  (*if cipher* ∈ *set cstars then None else decrypt* η *dkey cipher*, *Some* (*ekey*, *dkey*, *cstars*))

fun *ekey-oracle*
  :: *security* ⇒ (*'ekey*, *'dkey*, *'cipher*) *state-oracle* ⇒ *unit* ⇒ (*'ekey* × (*'ekey*, *'dkey*, *'cipher*) *state-oracle*) *spmf*
where
  ekey-oracle η *None* - = *do* {
    (*ekey*, *dkey*) ← *key-gen* η;
    return-spmf (*ekey*, *Some* (*ekey*, *dkey*, []))
  }
| *ekey-oracle* η (*Some* (*ekey*, *rest*)) - = *return-spmf* (*ekey*, *Some* (*ekey*, *rest*))

lemma *ekey-oracle-conv*:
  ekey-oracle η σ *x* =
  (*case* σ *of None* ⇒ *map-spmf* (λ(*ekey*, *dkey*). (*ekey*, *Some* (*ekey*, *dkey*, []))) (*key-gen* η)
  | *Some* (*ekey*, *rest*) ⇒ *return-spmf* (*ekey*, *Some* (*ekey*, *rest*)))
by(*cases* σ)(*auto simp add*: *map-spmf-conv-bind-spmf split-def*)

context notes *bind-spmf-cong*[*fundef-cong*] **begin**
function *encrypt-oracle*
  :: *bool* ⇒ *security* ⇒ (*'ekey*, *'dkey*, *'cipher*) *state-oracle* ⇒ *'plain* × *'plain*
  ⇒ (*'cipher* × (*'ekey*, *'dkey*, *'cipher*) *state-oracle*) *spmf*
where
  encrypt-oracle b η *None m01* = *do* { (-, σ) ← *ekey-oracle* η *None* (); *encrypt-oracle b* η σ *m01* }
| *encrypt-oracle b* η (*Some* (*ekey*, *dkey*, *cstars*)) (*m0*, *m1*) =
  (*if valid-plain* η *m0* ∧ *valid-plain* η *m1 then do* {

```
    let pb = (if b then m0 else m1);
    cstar ← encrypt η ekey pb;
    return-spmf (cstar, Some (ekey, dkey, cstar # cstars))
  } else return-pmf None)
by pat-completeness auto
termination by(relation Wellfounded.measure (λ(b, η, σ, m01). case σ of None
⇒ 1 | - ⇒ 0)) auto
end
```

1.3.1 Single-user setting

```
type-synonym ('plain', 'cipher') call₁ = unit + 'cipher' + 'plain' × 'plain'
type-synonym ('ekey', 'plain', 'cipher') ret₁ = 'ekey' + 'plain' option + 'cipher'

definition oracle₁ :: bool ⇒ security
  ⇒ (('ekey, 'dkey, 'cipher) state-oracle, ('plain, 'cipher) call₁, ('ekey, 'plain,
'cipher) ret₁) oracle'
where oracle₁ b η = ekey-oracle η ⊕_O (decrypt-oracle η ⊕_O encrypt-oracle b η)

lemma oracle₁-simps [simp]:
  oracle₁ b η s (Inl x) = map-spmf (apfst Inl) (ekey-oracle η s x)
  oracle₁ b η s (Inr (Inl y)) = map-spmf (apfst (Inr ∘ Inl)) (decrypt-oracle η s y)
  oracle₁ b η s (Inr (Inr z)) = map-spmf (apfst (Inr ∘ Inr)) (encrypt-oracle b η
s z)
by(simp-all add: oracle₁-def spmf.map-comp apfst-compose o-def)

type-synonym ('ekey', 'plain', 'cipher') adversary₁' =
  (bool, ('plain', 'cipher') call₁, ('ekey', 'plain', 'cipher') ret₁) gpv
type-synonym ('ekey', 'plain', 'cipher') adversary₁ =
  security ⇒ ('ekey', 'plain', 'cipher') adversary₁'

definition ind-cca2₁ :: ('ekey, 'plain, 'cipher) adversary₁ ⇒ security ⇒ bool spmf
where
  ind-cca2₁ 𝒜 η = TRY do {
    b ← coin-spmf;
    (guess, s) ← exec-gpv (oracle₁ b η) (𝒜 η) None;
    return-spmf (guess = b)
  } ELSE coin-spmf

definition advantage₁ :: ('ekey, 'plain, 'cipher) adversary₁ ⇒ advantage
where advantage₁ 𝒜 η = |spmf (ind-cca2₁ 𝒜 η) True − 1/2|

lemma advantage₁-nonneg: advantage₁ 𝒜 η ≥ 0 by(simp add: advantage₁-def)

abbreviation secure-for₁ :: ('ekey, 'plain, 'cipher) adversary₁ ⇒ bool
where secure-for₁ 𝒜 ≡ negligible (advantage₁ 𝒜)

definition ibounded-by₁' :: ('ekey, 'plain, 'cipher) adversary₁' ⇒ nat ⇒ bool
where ibounded-by₁' 𝒜 q = interaction-any-bounded-by 𝒜 q
```

abbreviation *ibounded-by*$_1$ :: ('*ekey*, '*plain*, '*cipher*) *adversary*$_1$ ⇒ (*security* ⇒ *nat*) ⇒ *bool*
where *ibounded-by*$_1$ ≡ *rel-envir ibounded-by*$_1$'

definition *lossless*$_1$' :: ('*ekey*, '*plain*, '*cipher*) *adversary*$_1$' ⇒ *bool*
where *lossless*$_1$' $\mathcal{A}$ = *lossless-gpv* $\mathcal{I}$-*full* $\mathcal{A}$

abbreviation *lossless*$_1$ :: ('*ekey*, '*plain*, '*cipher*) *adversary*$_1$ ⇒ *bool*
where *lossless*$_1$ ≡ *pred-envir lossless*$_1$'

lemma *lossless-decrypt-oracle* [*simp*]: *lossless-spmf* (*decrypt-oracle* η σ *cipher*)
by(*cases* (η, σ, *cipher*) *rule*: *decrypt-oracle.cases*) *simp-all*

lemma *lossless-ekey-oracle* [*simp*]:
lossless-spmf (*ekey-oracle* η σ *x*) ⟷ (σ = *None* ⟶ *lossless-spmf* (*key-gen* η))
by(*cases* (η, σ, *x*) *rule*: *ekey-oracle.cases*)(*auto*)

lemma *lossless-encrypt-oracle* [*simp*]:
⟦ σ = *None* ⟹ *lossless-spmf* (*key-gen* η);
⋀*ekey m*. *valid-plain* η *m* ⟹ *lossless-spmf* (*encrypt* η *ekey m*) ⟧
⟹ *lossless-spmf* (*encrypt-oracle b* η σ (*m0*, *m1*)) ⟷ *valid-plain* η *m0* ∧ *valid-plain* η *m1*
apply(*cases* (*b*, η, σ, (*m0*, *m1*)) *rule*: *encrypt-oracle.cases*)
apply(*auto simp add*: *split-beta dest*: *lossless-spmfD-set-spmf-nonempty split*: *if-split-asm*)
done

1.3.2 Multi-user setting

definition *oracle*$_n$ :: *bool* ⇒ *security*
⇒ ('*i* ⇒ ('*ekey*, '*dkey*, '*cipher*) *state-oracle*, '*i* × ('*plain*, '*cipher*) *call*$_1$, ('*ekey*, '*plain*, '*cipher*) *ret*$_1$) *oracle*'
where *oracle*$_n$ *b* η = *family-oracle* (λ-. *oracle*$_1$ *b* η)

lemma *oracle*$_n$-*apply* [*simp*]:
oracle$_n$ *b* η *s* (*i*, *x*) = *map-spmf* (*apsnd* (*fun-upd s i*)) (*oracle*$_1$ *b* η (*s i*) *x*)
by(*simp add*: *oracle*$_n$-*def*)

type-synonym ('*i*, '*ekey*', '*plain*', '*cipher*') *adversary*$_n$' =
(*bool*, '*i* × ('*plain*', '*cipher*') *call*$_1$, ('*ekey*', '*plain*', '*cipher*') *ret*$_1$) *gpv*
type-synonym ('*i*, '*ekey*', '*plain*', '*cipher*') *adversary*$_n$ =
security ⇒ ('*i*, '*ekey*', '*plain*', '*cipher*') *adversary*$_n$'

definition *ind-cca2*$_n$ :: ('*i*, '*ekey*, '*plain*, '*cipher*) *adversary*$_n$ ⇒ *security* ⇒ *bool spmf*
where
ind-cca2$_n$ $\mathcal{A}$ η = *TRY do* {
b ← *coin-spmf*;
(*guess*, σ) ← *exec-gpv* (*oracle*$_n$ *b* η) ($\mathcal{A}$ η) (λ-. *None*);

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

definition $\mathit{advantage}_n$ :: ('*i*, '*ekey*, '*plain*, '*cipher*) $\mathit{adversary}_n$ ⇒ *advantage*
where $\mathit{advantage}_n$ $\mathcal{A}$ η = |*spmf* (*ind-cca2*$_n$ $\mathcal{A}$ η) *True* − *1/2*|

lemma $\mathit{advantage}_n$*-nonneg*: $\mathit{advantage}_n$ $\mathcal{A}$ $\eta \geq 0$ **by**(*simp add*: $\mathit{advantage}_n$*-def*)

abbreviation *secure-for*$_n$ :: ('*i*, '*ekey*, '*plain*, '*cipher*) $\mathit{adversary}_n$ ⇒ *bool*
where *secure-for*$_n$ $\mathcal{A}$ ≡ *negligible* ($\mathit{advantage}_n$ $\mathcal{A}$)

definition *ibounded-by*$_n$' :: ('*i*, '*ekey*, '*plain*, '*cipher*) $\mathit{adversary}_n$' ⇒ *nat* ⇒ *bool*
where *ibounded-by*$_n$' $\mathcal{A}$ *q* = *interaction-any-bounded-by* $\mathcal{A}$ *q*

abbreviation *ibounded-by*$_n$ :: ('*i*, '*ekey*, '*plain*, '*cipher*) $\mathit{adversary}_n$ ⇒ (*security* ⇒ *nat*) ⇒ *bool*
where *ibounded-by*$_n$ ≡ *rel-envir ibounded-by*$_n$'

definition *lossless*$_n$' :: ('*i*, '*ekey*, '*plain*, '*cipher*) $\mathit{adversary}_n$' ⇒ *bool*
where *lossless*$_n$' $\mathcal{A}$ = *lossless-gpv* $\mathcal{I}$*-full* $\mathcal{A}$

abbreviation *lossless*$_n$ :: ('*i*, '*ekey*, '*plain*, '*cipher*) $\mathit{adversary}_n$ ⇒ *bool*
where *lossless*$_n$ ≡ *pred-envir lossless*$_n$'

definition *cipher-queries* :: ('*i* ⇒ ('*ekey*, '*dkey*, '*cipher*) *state-oracle*) ⇒ '*cipher set*
where *cipher-queries ose* = (⋃(-, -, *ciphers*)∈*ran ose*. *set ciphers*)

lemma *cipher-queriesI*:
⟦ *ose n* = *Some* (*ek*, *dk*, *ciphers*); *x* ∈ *set ciphers* ⟧ ⟹ *x* ∈ *cipher-queries ose*
by(*auto simp add*: *cipher-queries-def ran-def*)

lemma *cipher-queriesE*:
assumes *x* ∈ *cipher-queries ose*
obtains (*cipher-queries*) *n ek dk ciphers* **where** *ose n* = *Some* (*ek*, *dk*, *ciphers*) *x* ∈ *set ciphers*
using *assms* **by**(*auto simp add*: *cipher-queries-def ran-def*)

lemma *cipher-queries-updE*:
assumes *x* ∈ *cipher-queries* (*ose*(*n* ↦ (*ek*, *dk*, *ciphers*)))
obtains (*old*) *x* ∈ *cipher-queries ose x* ∉ *set ciphers* | (*new*) *x* ∈ *set ciphers*
using *assms* **by**(*cases x* ∈ *set ciphers*)(*fastforce elim*!: *cipher-queriesE split*: *if-split-asm intro*: *cipher-queriesI*)+

lemma *cipher-queries-empty* [*simp*]: *cipher-queries Map.empty* = {}
by(*simp add*: *cipher-queries-def*)

end

end

1.4 The IND-CCA2 security for symmetric encryption schemes

theory *IND-CCA2-sym* **imports**
  CryptHOL.Computational-Model
begin

locale *ind-cca* =
  fixes *key-gen* :: *'key spmf*
  and *encrypt* :: *'key* ⇒ *'message* ⇒ *'cipher spmf*
  and *decrypt* :: *'key* ⇒ *'cipher* ⇒ *'message option*
  and *msg-predicate* :: *'message* ⇒ *bool*
begin

type-synonym (*'message'*, *'cipher'*) *adversary* =
  (*bool*, *'message'* × *'message'* + *'cipher'*, *'cipher' option* + *'message' option*) *gpv*

definition *oracle-encrypt* :: *'key* ⇒ *bool* ⇒ (*'message* × *'message*, *'cipher option*, *'cipher set*) *callee*
where
  oracle-encrypt k b L = (λ(*msg1*, *msg0*).
    (*case msg-predicate msg1* ∧ *msg-predicate msg0 of*
      True ⇒ *do* {
       c ← *encrypt k* (*if b then msg1 else msg0*);
       return-spmf (*Some c*, {*c*} ∪ *L*)
      }
    | *False* ⇒ *return-spmf* (*None*, *L*)))

lemma *lossless-oracle-encrypt* [*simp*]:
  assumes *lossless-spmf* (*encrypt k m1*) **and** *lossless-spmf* (*encrypt k m0*)
  shows *lossless-spmf* (*oracle-encrypt k b L* (*m1*, *m0*))
using *assms* **by** (*simp add*: *oracle-encrypt-def split*: *bool.split*)

definition *oracle-decrypt* :: *'key* ⇒ (*'cipher*, *'message option*, *'cipher set*) *callee*
where *oracle-decrypt k L c* = *return-spmf* (*if c* ∈ *L then None else decrypt k c*, *L*)

lemma *lossless-oracle-decrypt* [*simp*]: *lossless-spmf* (*oracle-decrypt k L c*)
by(*simp add*: *oracle-decrypt-def*)

definition *game* :: (*'message*, *'cipher*) *adversary* ⇒ *bool spmf*
where
  game $\mathcal{A}$ = *do* {
    key ← *key-gen*;
    b ← *coin-spmf*;
    (*b'*, *L'*) ← *exec-gpv* (*oracle-encrypt key b* $\oplus_O$ *oracle-decrypt key*) $\mathcal{A}$ {};
    return-spmf (*b* = *b'*)

}

definition *advantage* :: ($'message$, $'cipher$) *adversary* ⇒ *real*
where *advantage* $\mathcal{A}$ = |*spmf* (*game* $\mathcal{A}$) *True* − *1* / *2*|

lemma *advantage-nonneg*: *0* ≤ *advantage* $\mathcal{A}$ **by**(*simp add*: *advantage-def*)

end

end

theory *IND-CPA* **imports**
CryptHOL.Generative-Probabilistic-Value
CryptHOL.Computational-Model
CryptHOL.Negligible
begin

1.5 The IND-CPA game for symmetric encryption schemes

locale *ind-cpa* =
fixes *key-gen* :: $'key$ *spmf* — probabilistic
and *encrypt* :: $'key$ ⇒ $'plain$ ⇒ $'cipher$ *spmf* — probabilistic
and *decrypt* :: $'key$ ⇒ $'cipher$ ⇒ $'plain$ *option* — deterministic, but not used
and *valid-plain* :: $'plain$ ⇒ *bool* — checks whether a plain text is valid, i.e., has the right format
begin

We cannot incorporate the predicate *valid-plain* in the type $'plain$ of plaintexts, because the single $'plain$ must contain plaintexts for all values of the security parameter, as HOL does not have dependent types. Consequently, the oracle has to ensure that the received plaintexts are valid.

type-synonym ($'plain'$, $'cipher'$, $'state$) *adversary* =
(($'plain'$ × $'plain'$) × $'state$, $'plain'$, $'cipher'$) *gpv*
× ($'cipher'$ ⇒ $'state$ ⇒ (*bool*, $'plain'$, $'cipher'$) *gpv*)

definition *encrypt-oracle* :: $'key$ ⇒ *unit* ⇒ $'plain$ ⇒ ($'cipher$ × *unit*) *spmf*
where
encrypt-oracle key σ *plain* = *do* {
cipher ← *encrypt key plain*;
return-spmf (*cipher*, ())
}

definition *ind-cpa* :: ($'plain$, $'cipher$, $'state$) *adversary* ⇒ *bool spmf*
where
ind-cpa $\mathcal{A}$ = *do* {
let ($\mathcal{A}1$, $\mathcal{A}2$) = $\mathcal{A}$;
key ← *key-gen*;
b ← *coin-spmf*;

(*guess*, -) ← *exec-gpv* (*encrypt-oracle key*) (*do* {
((*m0*, *m1*), σ) ← 𝒜*1*;
if valid-plain m0 ∧ *valid-plain m1 then do* {
cipher ← *lift-spmf* (*encrypt key* (*if b then m0 else m1*));
𝒜*2 cipher* σ
} *else lift-spmf coin-spmf*
}) ();
return-spmf (*guess* = *b*)
}

definition *advantage* :: (′*plain*, ′*cipher*, ′*state*) *adversary* ⇒ *real*
where *advantage* 𝒜 = |*spmf* (*ind-cpa* 𝒜) *True* − *1*/*2*|

lemma *advantage-nonneg*: *advantage* 𝒜 ≥ *0* **by**(*simp add*: *advantage-def*)

definition *ibounded-by* :: (′*plain*, ′*cipher*, ′*state*) *adversary* ⇒ *enat* ⇒ *bool*
where
ibounded-by = (λ(𝒜*1*, 𝒜*2*) *q*.
(∃ *q1 q2*. *interaction-any-bounded-by* 𝒜*1 q1* ∧ (∀ *cipher* σ. *interaction-any-bounded-by* (𝒜*2 cipher* σ) *q2*) ∧ *q1* + *q2* ≤ *q*))

lemma *ibounded-byE* [*consumes 1*, *case-names ibounded-by*, *elim?*]:
assumes *ibounded-by* (𝒜*1*, 𝒜*2*) *q*
obtains *q1 q2*
where *q1* + *q2* ≤ *q*
and *interaction-any-bounded-by* 𝒜*1 q1*
and ⋀*cipher* σ. *interaction-any-bounded-by* (𝒜*2 cipher* σ) *q2*
using *assms* **by**(*auto simp add*: *ibounded-by-def*)

lemma *ibounded-byI* [*intro?*]:
⟦ *interaction-any-bounded-by* 𝒜*1 q1*; ⋀*cipher* σ. *interaction-any-bounded-by* (𝒜*2 cipher* σ) *q2*; *q1* + *q2* ≤ *q* ⟧
⟹ *ibounded-by* (𝒜*1*, 𝒜*2*) *q*
by(*auto simp add*: *ibounded-by-def*)

definition *lossless* :: (′*plain*, ′*cipher*, ′*state*) *adversary* ⇒ *bool*
where *lossless* = (λ(𝒜*1*, 𝒜*2*). *lossless-gpv* ℐ*-full* 𝒜*1* ∧ (∀ *cipher* σ. *lossless-gpv* ℐ*-full* (𝒜*2 cipher* σ)))

end

end

theory *IND-CPA-PK* **imports**
CryptHOL.Computational-Model
CryptHOL.Negligible
begin

1.6 The IND-CPA game for public-key encryption with oracle access

locale *ind-cpa-pk* =
fixes *key-gen* :: $('pubkey \times 'privkey, 'call, 'ret)$ *gpv* — probabilistic
and *aencrypt* :: $'pubkey \Rightarrow 'plain \Rightarrow ('cipher, 'call, 'ret)$ *gpv* — probabilistic w/ access to an oracle
and *adecrypt* :: $'privkey \Rightarrow 'cipher \Rightarrow ('plain, 'call, 'ret)$ *gpv* — not used
and *valid-plains* :: $'plain \Rightarrow 'plain \Rightarrow bool$ — checks whether a pair of plaintexts is valid, i.e., they have the right format
begin

We cannot incorporate the predicate *valid-plain* in the type *'plain* of plaintexts, because the single *'plain* must contain plaintexts for all values of the security parameter, as HOL does not have dependent types. Consequently, the game has to ensure that the received plaintexts are valid.

type-synonym $('pubkey', 'plain', 'cipher', 'call', 'ret', 'state)$ *adversary* =
$('pubkey' \Rightarrow (('plain' \times 'plain') \times 'state, 'call', 'ret')\ gpv)$
$\times\ ('cipher' \Rightarrow 'state \Rightarrow (bool, 'call', 'ret')\ gpv)$

fun *ind-cpa* :: $('pubkey, 'plain, 'cipher, 'call, 'ret, 'state)\ adversary \Rightarrow (bool, 'call, 'ret)\ gpv$
where
ind-cpa $(\mathcal{A}1, \mathcal{A}2)$ = *TRY do* {
$(pk, sk) \leftarrow$ *key-gen*;
$b \leftarrow$ *lift-spmf coin-spmf*;
$((m0, m1), \sigma) \leftarrow (\mathcal{A}1\ pk)$;
assert-gpv (*valid-plains m0 m1*);
cipher $\leftarrow$ *aencrypt pk* (*if b then m0 else m1*);
guess $\leftarrow \mathcal{A}2$ *cipher* σ;
Done (*guess* = *b*)
} *ELSE lift-spmf coin-spmf*

definition *advantage* :: $('\sigma \Rightarrow 'call \Rightarrow ('ret \times '\sigma)\ spmf) \Rightarrow '\sigma \Rightarrow ('pubkey, 'plain, 'cipher, 'call, 'ret, 'state)\ adversary \Rightarrow real$
where *advantage oracle* $\sigma\ \mathcal{A} = |spmf\ (run\text{-}gpv\ oracle\ (ind\text{-}cpa\ \mathcal{A})\ \sigma)\ True - 1/2|$

lemma *advantage-nonneg*: *advantage oracle* $\sigma\ \mathcal{A} \geq 0$ **by**(*simp add*: *advantage-def*)

definition *ibounded-by* :: $('call \Rightarrow bool) \Rightarrow ('pubkey, 'plain, 'cipher, 'call, 'ret, 'state)\ adversary \Rightarrow enat \Rightarrow bool$
where
ibounded-by consider = $(\lambda(\mathcal{A}1, \mathcal{A}2)\ q.$
$(\exists q1\ q2.\ (\forall pk.\ interaction\text{-}bounded\text{-}by\ consider\ (\mathcal{A}1\ pk)\ q1) \wedge (\forall cipher\ \sigma.\ interaction\text{-}bounded\text{-}by\ consider\ (\mathcal{A}2\ cipher\ \sigma)\ q2) \wedge q1 + q2 \leq q))$

lemma *ibounded-by'E* [*consumes 1*, *case-names ibounded-by'*, *elim?*]:
assumes *ibounded-by consider* $(\mathcal{A}1, \mathcal{A}2)\ q$
obtains *q1 q2*

where $q1 + q2 \leq q$
and $\bigwedge pk.$ *interaction-bounded-by consider* $(\mathcal{A}1\ pk)\ q1$
and $\bigwedge cipher\ \sigma.$ *interaction-bounded-by consider* $(\mathcal{A}2\ cipher\ \sigma)\ q2$
using *assms* **by**(*auto simp add*: *ibounded-by-def*)

lemma *ibounded-byI* [*intro?*]:
$[\![\ \bigwedge pk.$ *interaction-bounded-by consider* $(\mathcal{A}1\ pk)\ q1;\ \bigwedge cipher\ \sigma.$ *interaction-bounded-by consider* $(\mathcal{A}2\ cipher\ \sigma)\ q2;\ q1 + q2 \leq q\ ]\!]$
$\Longrightarrow$ *ibounded-by consider* $(\mathcal{A}1, \mathcal{A}2)\ q$
by(*auto simp add*: *ibounded-by-def*)

definition *lossless* :: $('pubkey, 'plain, 'cipher, 'call, 'ret, 'state)$ *adversary* $\Rightarrow$ *bool*
where *lossless* $= (\lambda(\mathcal{A}1, \mathcal{A}2).\ (\forall pk.$ *lossless-gpv* $\mathcal{I}$*-full* $(\mathcal{A}1\ pk)) \wedge (\forall cipher\ \sigma.$ *lossless-gpv* $\mathcal{I}$*-full* $(\mathcal{A}2\ cipher\ \sigma)))$

end

end

theory *IND-CPA-PK-Single* **imports**
CryptHOL.Computational-Model
begin

1.7 The IND-CPA game (public key, single instance)

locale *ind-cpa* =
fixes *key-gen* :: $('pub\text{-}key \times 'priv\text{-}key)$ *spmf* — probabilistic
and *aencrypt* :: $'pub\text{-}key \Rightarrow 'plain \Rightarrow 'cipher$ *spmf* — probabilistic
and *adecrypt* :: $'priv\text{-}key \Rightarrow 'cipher \Rightarrow 'plain$ *option* — deterministic, but not used
and *valid-plains* :: $'plain \Rightarrow 'plain \Rightarrow bool$ — checks whether a pair of plaintexts is valid, i.e., they both have the right format
begin

We cannot incorporate the predicate *valid-plain* in the type *'plain* of plaintexts, because the single *'plain* must contain plaintexts for all values of the security parameter, as HOL does not have dependent types. Consequently, the oracle has to ensure that the received plaintexts are valid.

type-synonym $('pub\text{-}key', 'plain', 'cipher', 'state)$ *adversary* =
$('pub\text{-}key' \Rightarrow (('plain' \times 'plain') \times 'state)\ spmf)$
$\times\ ('cipher' \Rightarrow 'state \Rightarrow bool\ spmf)$

primrec *ind-cpa* :: $('pub\text{-}key, 'plain, 'cipher, 'state)$ *adversary* $\Rightarrow$ *bool spmf*
where
ind-cpa $(\mathcal{A}1, \mathcal{A}2) =$ *TRY do* {
$(pk, sk) \leftarrow$ *key-gen*;
$((m0, m1), \sigma) \leftarrow \mathcal{A}1\ pk$;
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains m0 m1*);

```
    b ← coin-spmf;
    cipher ← aencrypt pk (if b then m0 else m1);
    b' ← A2 cipher σ;
    return-spmf (b = b')
  } ELSE coin-spmf
```

declare *ind-cpa.simps* [*simp del*]

definition *advantage* :: ('*pub-key*, '*plain*, '*cipher*, '*state*) *adversary* ⇒ *real*
where *advantage* $\mathcal{A}$ = |*spmf* (*ind-cpa* $\mathcal{A}$) *True* − *1/2*|

definition *lossless* :: ('*pub-key*, '*plain*, '*cipher*, '*state*) *adversary* ⇒ *bool*
where
  lossless $\mathcal{A}$ ⟷
   ((∀ *pk*. *lossless-spmf* (*fst* $\mathcal{A}$ *pk*)) ∧
       (∀ *cipher* σ. *lossless-spmf* (*snd* $\mathcal{A}$ *cipher* σ)))

lemma *lossless-ind-cpa*:
  ⟦ *lossless* $\mathcal{A}$; *lossless-spmf* (*key-gen*) ⟧ ⟹ *lossless-spmf* (*ind-cpa* $\mathcal{A}$)
by(*auto simp add*: *lossless-def ind-cpa-def split-def Let-def*)

end

end

theory *SUF-CMA* **imports**
  CryptHOL.Computational-Model
  CryptHOL.Negligible
  CryptHOL.Environment-Functor
begin

1.8 Strongly existentially unforgeable signature scheme

locale *sig-scheme* =
  fixes *key-gen* :: *security* ⇒ ('*vkey* × '*sigkey*) *spmf*
  and *sign* :: *security* ⇒ '*sigkey* ⇒ '*message* ⇒ '*signature spmf*
  and *verify* :: *security* ⇒ '*vkey* ⇒ '*message* ⇒ '*signature* ⇒ *bool* — verification is deterministic
  and *valid-message* :: *security* ⇒ '*message* ⇒ *bool*

locale *suf-cma* = *sig-scheme* +
  constrains *key-gen* :: *security* ⇒ ('*vkey* × '*sigkey*) *spmf*
  and *sign* :: *security* ⇒ '*sigkey* ⇒ '*message* ⇒ '*signature spmf*
  and *verify* :: *security* ⇒ '*vkey* ⇒ '*message* ⇒ '*signature* ⇒ *bool*
  and *valid-message* :: *security* ⇒ '*message* ⇒ *bool*
begin

type-synonym ('*vkey*', '*sigkey*', '*message*', '*signature*') *state-oracle*

= ('vkey' × 'sigkey' × ('message' × 'signature') list) option

fun *vkey-oracle* :: *security* ⇒ (('vkey, 'sigkey, 'message, 'signature) state-oracle, unit, 'vkey) oracle'
where
vkey-oracle η *None* - = *do* {
(*vkey*, *sigkey*) ← *key-gen* η;
return-spmf (*vkey*, *Some* (*vkey*, *sigkey*, []))
}
| ⋀*log*. *vkey-oracle* η (*Some* (*vkey*, *sigkey*, *log*)) - = *return-spmf* (*vkey*, *Some* (*vkey*, *sigkey*, *log*))

context notes *bind-spmf-cong*[*fundef-cong*] **begin**
function *sign-oracle*
:: *security* ⇒ (('vkey, 'sigkey, 'message, 'signature) state-oracle, 'message, 'signature) oracle'
where
sign-oracle η *None m* = *do* { (-, σ) ← *vkey-oracle* η *None* (); *sign-oracle* η σ *m*
}
| ⋀*log*. *sign-oracle* η (*Some* (*vkey*, *skey*, *log*)) *m* =
(*if valid-message* η *m then do* {
sig ← *sign* η *skey m*;
return-spmf (*sig*, *Some* (*vkey*, *skey*, (*m*, *sig*) # *log*))
} *else return-pmf None*)
by *pat-completeness auto*
termination by(*relation Wellfounded.measure* (λ(η, σ, *m*). *case* σ *of None* ⇒ *1* | - ⇒ *0*)) *auto*
end

lemma *lossless-vkey-oracle* [*simp*]:
lossless-spmf (*vkey-oracle* η σ *x*) ⟷ (σ = *None* ⟶ *lossless-spmf* (*key-gen* η))
by(*cases* (η, σ, *x*) *rule*: *vkey-oracle.cases*) *auto*

lemma *lossless-sign-oracle* [*simp*]:
⟦ σ = *None* ⟹ *lossless-spmf* (*key-gen* η);
⋀*skey m*. *valid-message* η *m* ⟹ *lossless-spmf* (*sign* η *skey m*) ⟧
⟹ *lossless-spmf* (*sign-oracle* η σ *m*) ⟷ *valid-message* η *m*
apply(*cases* (η, σ, *m*) *rule*: *sign-oracle.cases*)
apply(*auto simp add*: *split-beta dest*: *lossless-spmfD-set-spmf-nonempty*)
done

lemma *lossless-sign-oracle-Some*: **fixes** *log* **shows**
lossless-spmf (*sign-oracle* η (*Some* (*vkey*, *skey*, *log*)) *m*) ⟷ *lossless-spmf* (*sign* η *skey m*) ∧ *valid-message* η *m*
by(*simp*)

1.8.1 Single-user setting

type-synonym 'message' $call_1$ = *unit* + 'message'

type-synonym ($'vkey$, $'signature$) ret_1 = $'vkey$ + $'signature$

definition $oracle_1$ :: $security$
⇒ (($'vkey$, $'sigkey$, $'message$, $'signature$) $state\text{-}oracle$, $'message$ $call_1$, ($'vkey$, $'signature$) ret_1) $oracle'$
where $oracle_1\ \eta = vkey\text{-}oracle\ \eta \oplus_O sign\text{-}oracle\ \eta$

lemma $oracle_1$-$simps$ [$simp$]:
$oracle_1\ \eta\ s\ (Inl\ x) = map\text{-}spmf\ (apfst\ Inl)\ (vkey\text{-}oracle\ \eta\ s\ x)$
$oracle_1\ \eta\ s\ (Inr\ y) = map\text{-}spmf\ (apfst\ Inr)\ (sign\text{-}oracle\ \eta\ s\ y)$
by($simp\text{-}all$ add: $oracle_1$-def)

type-synonym ($'vkey$, $'message$, $'signature$) $adversary_1'$ =
(($'message$ × $'signature$), $'message$ $call_1$, ($'vkey$, $'signature$) ret_1) gpv
type-synonym ($'vkey$, $'message$, $'signature$) $adversary_1$ =
$security$ ⇒ ($'vkey$, $'message$, $'signature$) $adversary_1'$

definition $suf\text{-}cma_1$ :: ($'vkey$, $'message$, $'signature$) $adversary_1$ ⇒ $security$ ⇒ $bool$ $spmf$
where
$\bigwedge log.\ suf\text{-}cma_1\ \mathcal{A}\ \eta = do$ {
((m, sig), σ) ← $exec\text{-}gpv\ (oracle_1\ \eta)\ (\mathcal{A}\ \eta)\ None$;
$return\text{-}spmf$ (
$case\ \sigma\ of\ None \Rightarrow False$
| $Some\ (vkey, skey, log) \Rightarrow verify\ \eta\ vkey\ m\ sig \wedge (m, sig) \notin set\ log$)
}

definition $advantage_1$ :: ($'vkey$, $'message$, $'signature$) $adversary_1$ ⇒ $advantage$
where $advantage_1\ \mathcal{A}\ \eta = spmf\ (suf\text{-}cma_1\ \mathcal{A}\ \eta)\ True$

lemma $advantage_1$-$nonneg$: $advantage_1\ \mathcal{A}\ \eta \geq 0$ **by**($simp$ add: $advantage_1$-def $pmf\text{-}nonneg$)

abbreviation $secure\text{-}for_1$ :: ($'vkey$, $'message$, $'signature$) $adversary_1$ ⇒ $bool$
where $secure\text{-}for_1\ \mathcal{A} \equiv negligible\ (advantage_1\ \mathcal{A})$

definition $ibounded\text{-}by_1'$ :: ($'vkey$, $'message$, $'signature$) $adversary_1'$ ⇒ nat ⇒ $bool$
where $ibounded\text{-}by_1'\ \mathcal{A}\ q = (interaction\text{-}any\text{-}bounded\text{-}by\ \mathcal{A}\ q)$

abbreviation $ibounded\text{-}by_1$ :: ($'vkey$, $'message$, $'signature$) $adversary_1$ ⇒ ($security$ ⇒ nat) ⇒ $bool$
where $ibounded\text{-}by_1 \equiv rel\text{-}envir\ ibounded\text{-}by_1'$

definition $lossless_1'$ :: ($'vkey$, $'message$, $'signature$) $adversary_1'$ ⇒ $bool$
where $lossless_1'\ \mathcal{A} = (lossless\text{-}gpv\ \mathcal{I}\text{-}full\ \mathcal{A})$

abbreviation $lossless_1$ :: ($'vkey$, $'message$, $'signature$) $adversary_1$ ⇒ $bool$
where $lossless_1 \equiv pred\text{-}envir\ lossless_1'$

1.8.2 Multi-user setting

definition $oracle_n$ :: $security$
$\Rightarrow$ ($'i \Rightarrow$ ($'vkey$, $'sigkey$, $'message$, $'signature$) $state\text{-}oracle$, $'i \times 'message$ $call_1$, ($'vkey$, $'signature$) ret_1) $oracle'$
where $oracle_n$ η = $family\text{-}oracle$ (λ-. $oracle_1$ η)

lemma $oracle_n$-$apply$ [$simp$]:
$oracle_n$ η s (i, x) = $map\text{-}spmf$ ($apsnd$ ($fun\text{-}upd$ s i)) ($oracle_1$ η (s i) x)
by($simp$ add: $oracle_n$-def)

type-synonym ($'i$, $'vkey'$, $'message'$, $'signature'$) $adversary_n'$ =
(($'i \times 'message' \times 'signature'$), $'i \times 'message'$ $call_1$, ($'vkey'$, $'signature'$) ret_1) gpv
type-synonym ($'i$, $'vkey'$, $'message'$, $'signature'$) $adversary_n$ =
$security \Rightarrow$ ($'i$, $'vkey'$, $'message'$, $'signature'$) $adversary_n'$

definition $suf\text{-}cma_n$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n \Rightarrow security \Rightarrow$ $bool$ $spmf$
where
$\bigwedge log$. $suf\text{-}cma_n$ $\mathcal{A}$ η = do {
((i, m, sig), σ) $\leftarrow$ $exec\text{-}gpv$ ($oracle_n$ η) ($\mathcal{A}$ η) (λ-. $None$);
$return\text{-}spmf$ (
$case$ σ i of $None \Rightarrow False$
| $Some$ ($vkey$, $skey$, log) $\Rightarrow$ $verify$ η $vkey$ m sig $\wedge$ (m, sig) $\notin$ set log)
}

definition $advantage_n$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n \Rightarrow advantage$
where $advantage_n$ $\mathcal{A}$ η = $spmf$ ($suf\text{-}cma_n$ $\mathcal{A}$ η) $True$

lemma $advantage_n$-$nonneg$: $advantage_n$ $\mathcal{A}$ $\eta \geq 0$ **by**($simp$ add: $advantage_n$-def $pmf\text{-}nonneg$)

abbreviation $secure\text{-}for_n$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n \Rightarrow bool$
where $secure\text{-}for_n$ $\mathcal{A}$ $\equiv$ $negligible$ ($advantage_n$ $\mathcal{A}$)

definition $ibounded\text{-}by_n'$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n' \Rightarrow nat \Rightarrow$ $bool$
where $ibounded\text{-}by_n'$ $\mathcal{A}$ q = ($interaction\text{-}any\text{-}bounded\text{-}by$ $\mathcal{A}$ q)

abbreviation $ibounded\text{-}by_n$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n \Rightarrow$ ($security$ $\Rightarrow nat$) $\Rightarrow bool$
where $ibounded\text{-}by_n$ $\equiv$ $rel\text{-}envir$ $ibounded\text{-}by_n'$

definition $lossless_n'$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n' \Rightarrow bool$
where $lossless_n'$ $\mathcal{A}$ = ($lossless\text{-}gpv$ $\mathcal{I}\text{-}full$ $\mathcal{A}$)

abbreviation $lossless_n$ :: ($'i$, $'vkey$, $'message$, $'signature$) $adversary_n \Rightarrow bool$
where $lossless_n$ $\equiv$ $pred\text{-}envir$ $lossless_n'$

end

end

theory *Pseudo-Random-Function* **imports**
CryptHOL.Computational-Model
begin

1.9 Pseudo-random function

locale *random-function* =
fixes p :: $'a$ *spmf*
begin

type-synonym $('b,'a')$ *dict* = $'b \rightharpoonup 'a'$

definition *random-oracle* :: $('b, 'a)$ *dict* $\Rightarrow$ $'b \Rightarrow ('a \times ('b, 'a)$ *dict*$)$ *spmf*
where
random-oracle σ x =
(*case* σ x *of Some* $y \Rightarrow$ *return-spmf* (y, σ)
| *None* $\Rightarrow p \gg\!= (\lambda y.$ *return-spmf* $(y, \sigma(x \mapsto y))))$

definition *forgetful-random-oracle* :: *unit* $\Rightarrow 'b \Rightarrow ('a \times$ *unit*$)$ *spmf*
where
forgetful-random-oracle σ $x = p \gg\!= (\lambda y.$ *return-spmf* $(y, ()))$

lemma *weight-random-oracle* [*simp*]:
weight-spmf $p = 1 \Longrightarrow$ *weight-spmf* (*random-oracle* σ x) = 1
by(*simp add*: *random-oracle-def weight-bind-spmf o-def split*: *option.split*)

lemma *lossless-random-oracle* [*simp*]:
lossless-spmf $p \Longrightarrow$ *lossless-spmf* (*random-oracle* σ x)
by(*simp add*: *lossless-spmf-def*)

sublocale *finite*: *callee-invariant-on random-oracle* $\lambda\sigma.$ *finite* (*dom* σ) $\mathcal{I}$*-full*
by(*unfold-locales*)(*auto simp add*: *random-oracle-def split*: *option.splits*)

lemma *card-dom-random-oracle*:
assumes *interaction-any-bounded-by* $\mathcal{A}$ q
and $(y, \sigma') \in$ *set-spmf* (*exec-gpv random-oracle* $\mathcal{A}$ σ)
and *fin*: *finite* (*dom* σ)
shows *card* (*dom* σ') $\leq q +$ *card* (*dom* σ)
by(*rule finite.interaction-bounded-by'-exec-gpv-count*[*OF assms*(*1*−*2*)])
(*auto simp add*: *random-oracle-def fin card-insert-if simp del*: *fun-upd-apply split*: *option.split-asm*)

end

1.10 Pseudo-random function

locale *prf* =
  fixes *key-gen* :: *'key spmf*
  and *prf* :: *'key* ⇒ *'domain* ⇒ *'range*
  and *rand* :: *'range spmf*
begin

sublocale *random-function rand* **.**

definition *prf-oracle* :: *'key* ⇒ *unit* ⇒ *'domain* ⇒ (*'range* × *unit*) *spmf*
where *prf-oracle key σ x* = *return-spmf* (*prf key x*, ())

type-synonym (*'domain'*, *'range'*) *adversary* = (*bool*, *'domain'*, *'range'*) *gpv*

definition *game-0* :: (*'domain*, *'range*) *adversary* ⇒ *bool spmf*
where
  game-0 $\mathcal{A}$ = *do* {
    key ← *key-gen*;
    (*b*, -) ← *exec-gpv* (*prf-oracle key*) $\mathcal{A}$ ();
    return-spmf b
  }

definition *game-1* :: (*'domain*, *'range*) *adversary* ⇒ *bool spmf*
where
  game-1 $\mathcal{A}$ = *do* {
    (*b*, -) ← *exec-gpv random-oracle* $\mathcal{A}$ *empty*;
    return-spmf b
  }

definition *advantage* :: (*'domain*, *'range*) *adversary* ⇒ *real*
where *advantage* $\mathcal{A}$ = |*spmf* (*game-0* $\mathcal{A}$) *True* − *spmf* (*game-1* $\mathcal{A}$) *True*|

lemma *advantage-nonneg*: *advantage* $\mathcal{A}$ ≥ *0*
by(*simp add*: *advantage-def*)

abbreviation *lossless* :: (*'domain*, *'range*) *adversary* ⇒ *bool*
where *lossless* ≡ *lossless-gpv* $\mathcal{I}$*-full*

abbreviation (*input*) *ibounded-by* :: (*'domain*, *'range*) *adversary* ⇒ *enat* ⇒ *bool*
where *ibounded-by* ≡ *interaction-any-bounded-by*

end

end

1.11 Random permutation

theory *Pseudo-Random-Permutation* **imports**
  CryptHOL.Computational-Model

begin

locale *random-permutation* =
  fixes A :: $'b$ *set*
begin

definition *random-permutation* :: $('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow ('b \times ('a \rightharpoonup 'b))$ *spmf*
where
  random-permutation σ x =
  (*case* σ x *of Some* $y \Rightarrow$ *return-spmf* (y, σ)
   | *None* $\Rightarrow$ *spmf-of-set* $(A - ran\ \sigma) \ggg (\lambda y.$ *return-spmf* $(y, \sigma(x \mapsto y))))$

lemma *weight-random-oracle* [*simp*]:
  $[\![$ *finite* A; $A - ran\ \sigma \neq \{\}$ $]\!] \Longrightarrow$ *weight-spmf* (*random-permutation* σ x) = *1*
by(*simp add*: *random-permutation-def weight-bind-spmf o-def split*: *option.split*)

lemma *lossless-random-oracle* [*simp*]:
  $[\![$ *finite* A; $A - ran\ \sigma \neq \{\}$ $]\!] \Longrightarrow$ *lossless-spmf* (*random-permutation* σ x)
by(*simp add*: *lossless-spmf-def*)

sublocale *finite*: *callee-invariant-on random-permutation* $\lambda\sigma.$ *finite* (*dom* σ) $\mathcal{I}$*-full*
by(*unfold-locales*)(*auto simp add*: *random-permutation-def split*: *option.splits*)

lemma *card-dom-random-oracle*:
  assumes *interaction-any-bounded-by* $\mathcal{A}$ q
  and $(y, \sigma') \in$ *set-spmf* (*exec-gpv random-permutation* $\mathcal{A}$ σ)
  and *fin*: *finite* (*dom* σ)
  shows *card* (*dom* σ') $\leq q +$ *card* (*dom* σ)
by(*rule finite.interaction-bounded-by'-exec-gpv-count*[*OF assms*(*1*−*2*)])
 (*auto simp add*: *random-permutation-def fin card-insert-if simp del*: *fun-upd-apply split*: *option.split-asm*)

end

end

1.12 Reducing games with many adversary guesses to games with single guesses

theory *Guessing-Many-One* **imports**
  CryptHOL.Computational-Model
  CryptHOL.GPV-Bisim
begin

locale *guessing-many-one* =
  fixes *init* :: $('c\text{-}o \times 'c\text{-}a \times 's)$ *spmf*
  and *oracle* :: $'c\text{-}o \Rightarrow 's \Rightarrow 'call \Rightarrow ('ret \times 's)$ *spmf*
  and *eval* :: $'c\text{-}o \Rightarrow 'c\text{-}a \Rightarrow 's \Rightarrow 'guess \Rightarrow$ *bool spmf*
begin

type-synonym ('*c-a*', '*guess*', '*call*', '*ret*') *adversary-single* = '*c-a*' ⇒ ('*guess*', '*call*', '*ret*') *gpv*

definition *game-single* :: ('*c-a*, '*guess*, '*call*, '*ret*) *adversary-single* ⇒ *bool spmf*
where
  game-single $\mathcal{A}$ = *do* {
    (*c-o*, *c-a*, *s*) ← *init*;
    (*guess*, *s*') ← *exec-gpv* (*oracle c-o*) ($\mathcal{A}$ *c-a*) *s*;
    eval c-o c-a s' guess
  }

definition *advantage-single* :: ('*c-a*, '*guess*, '*call*, '*ret*) *adversary-single* ⇒ *real*
where *advantage-single* $\mathcal{A}$ = *spmf* (*game-single* $\mathcal{A}$) *True*

type-synonym ('*c-a*', '*guess*', '*call*', '*ret*') *adversary-many* = '*c-a*' ⇒ (*unit*, '*call*' + '*guess*', '*ret*' + *unit*) *gpv*

definition *eval-oracle* :: '*c-o* ⇒ '*c-a* ⇒ *bool* × '*s* ⇒ '*guess* ⇒ (*unit* × (*bool* × '*s*)) *spmf*
where
  eval-oracle c-o c-a = (λ(*b*, *s*') *guess*. *map-spmf* (λ*b*'. ((), (*b* ∨ *b*', *s*'))) (*eval c-o c-a s' guess*))

definition *game-multi* :: ('*c-a*, '*guess*, '*call*, '*ret*) *adversary-many* ⇒ *bool spmf*
where
  game-multi $\mathcal{A}$ = *do* {
    (*c-o*, *c-a*, *s*) ← *init*;
    (-, (*b*, -)) ← *exec-gpv*
      (†(*oracle c-o*) $\oplus_O$ *eval-oracle c-o c-a*)
      ($\mathcal{A}$ *c-a*)
      (*False*, *s*);
    return-spmf b
  }

definition *advantage-multi* :: ('*c-a*, '*guess*, '*call*, '*ret*) *adversary-many* ⇒ *real*
where *advantage-multi* $\mathcal{A}$ = *spmf* (*game-multi* $\mathcal{A}$) *True*

type-synonym '*guess*' *reduction-state* = '*guess*' + *nat*

primrec *process-call* :: '*guess reduction-state* ⇒ '*call* ⇒ ('*ret option* × '*guess reduction-state*, '*call*, '*ret*) *gpv*
where
  process-call (*Inr j*) *x* = *do* {
    ret ← *Pause x Done*;
    Done (*Some ret*, *Inr j*)
  }

| *process-call* (*Inl guess*) *x* = *Done* (*None*, *Inl guess*)

primrec *process-guess* :: *'guess reduction-state* ⇒ *'guess* ⇒ (*unit option* × *'guess reduction-state*, *'call*, *'ret*) *gpv*
where
process-guess (*Inr j*) *guess* = *Done* (*if j* > *0 then* (*Some* (), *Inr* (*j* − *1*)) *else* (*None*, *Inl guess*))
| *process-guess* (*Inl guess*) - = *Done* (*None*, *Inl guess*)

abbreviation *reduction-oracle* :: *'guess* + *nat* ⇒ *'call* + *'guess* ⇒ ((*'ret* + *unit*) *option* × (*'guess* + *nat*), *'call*, *'ret*) *gpv*
where *reduction-oracle* ≡ *plus-intercept-stop process-call process-guess*

definition *reduction* :: *nat* ⇒ (*'c-a*, *'guess*, *'call*, *'ret*) *adversary-many* ⇒ (*'c-a*, *'guess*, *'call*, *'ret*) *adversary-single*
where
reduction q $\mathcal{A}$ *c-a* = *do* {
j-star ← *lift-spmf* (*spmf-of-set* {..<*q*});
(-, *s*) ← *inline-stop reduction-oracle* ($\mathcal{A}$ *c-a*) (*Inr j-star*);
Done (*projl s*)
}

lemma *many-single-reduction*:
assumes *bound*: ⋀*c-a c-o s*. (*c-o*, *c-a*, *s*) ∈ *set-spmf init* ⟹ *interaction-bounded-by* (*Not* ∘ *isl*) ($\mathcal{A}$ *c-a*) *q*
and *lossless-oracle*: ⋀*c-a c-o s s' x*. (*c-o*, *c-a*, *s*) ∈ *set-spmf init* ⟹ *lossless-spmf* (*oracle c-o s' x*)
and *lossless-eval*: ⋀*c-a c-o s s' guess*. (*c-o*, *c-a*, *s*) ∈ *set-spmf init* ⟹ *lossless-spmf* (*eval c-o c-a s' guess*)
shows *advantage-multi* $\mathcal{A}$ ≤ *advantage-single* (*reduction q* $\mathcal{A}$) ∗ *q*
including *lifting-syntax*
proof −
def *eval-oracle'* ≡ λ*c-o c-a* ((*id*, *occ* :: *nat option*), *s'*) *guess*.
map-spmf (λ*b'*. *case occ of Some* j_0 ⇒ ((), (*Suc id*, *Some* j_0), *s'*)
| *None* ⇒ ((), (*Suc id*, (*if b' then Some id else None*)), *s'*))
(*eval c-o c-a s' guess*)
let *?multi'-body* = λ*c-o c-a s*. *exec-gpv* (†(*oracle c-o*) $\oplus_O$ *eval-oracle' c-o c-a*) ($\mathcal{A}$ *c-a*) ((*0*, *None*), *s*)
def *game-multi'* ≡ λ*c-o c-a s*. *do* {
(-, ((*id*, j_0), *s'* :: *'s*)) ← *?multi'-body c-o c-a s*;
return-spmf (j_0 ≠ *None*) }

define *initialize* :: (*'c-o* ⇒ *'c-a* ⇒ *'s* ⇒ *nat* ⇒ *bool spmf*) ⇒ *bool spmf* **where**
initialize body = *do* {
(*c-o*, *c-a*, *s*) ← *init*;
j_s ← *spmf-of-set* {..<*q*};
body c-o c-a s j_s } **for** *body*
define *body2* **where** *body2 c-o c-a s* j_s = *do* {

(-, (*id*, j_0), *s'*) ← *?multi'-body c-o c-a s*;
return-spmf (j_0 = *Some* j_s) } **for** *c-o c-a s* j_s
let *?game2* = *initialize body2*

def *stop-oracle* ≡ λ*c-o*.
(λ(*idgs*, *s*) *x*. *case idgs of Inr* - ⇒ *map-spmf* (λ(*y*, *s*). (*Some y*, (*idgs*, *s*))) (*oracle c-o s x*) | *Inl* - ⇒ *return-spmf* (*None*, (*idgs*, *s*)))
$\oplus_O{}^S$
(λ(*idgs*, *s*) *guess* :: *'guess*. *return-spmf* (*case idgs of Inr 0* ⇒ (*None*, *Inl* (*guess*, *s*), *s*) | *Inr* (*Suc i*) ⇒ (*Some* (), *Inr i*, *s*) | *Inl* - ⇒ (*None*, *idgs*, *s*)))
define *body3* **where** *body3 c-o c-a s* j_s = *do* {
(- :: *unit option*, *idgs*, -) ← *exec-gpv-stop* (*stop-oracle c-o*) ($\mathcal{A}$ *c-a*) (*Inr* j_s, *s*);
(*b'* :: *bool*) ← *case idgs of Inr* - ⇒ *return-spmf False* | *Inl* (*g*, *s'*) ⇒ *eval c-o c-a s' g*;
return-spmf b' } **for** *c-o c-a s* j_s
let *?game3* = *initialize body3*

{ define *S* :: *bool* ⇒ *nat* × *nat option* ⇒ *bool* **where** *S* ≡ λ*b'* (*id*, *occ*). *b'* ⟷ (∃j_0. *occ* = *Some* j_0)
let *?S* = *rel-prod S op* =

define *initial* :: *nat* × *nat option* **where** *initial* = (*0*, *None*)
define *result* :: *nat* × *nat option* ⇒ *bool* **where** *result p* = (*snd p* ≠ *None*) **for** *p*
have [*transfer-rule*]: (*S* ===> *op* =) (λ*b*. *b*) *result* **by**(*simp add*: *rel-fun-def result-def S-def*)
have [*transfer-rule*]: *S False initial* **by** (*simp add*: *S-def initial-def*)

have *eval-oracle'*[*transfer-rule*]:
(*op* = ===> *op* = ===> *?S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *?S*))
eval-oracle eval-oracle'
unfolding *eval-oracle-def*[*abs-def*] *eval-oracle'-def*[*abs-def*]
by (*auto simp add*: *rel-fun-def S-def map-spmf-conv-bind-spmf intro*!: *rel-spmf-bind-reflI split*: *option.split*)

have *game-multi'*: *game-multi* $\mathcal{A}$ = *bind-spmf init* (λ(*c-o*, *c-a*, *s*). *game-multi' c-o c-a s*)
unfolding *game-multi-def game-multi'-def initial-def*[*symmetric*]
by (*rewrite* **in** *case-prod* ¤ **in** *bind-spmf* - (*case-prod* ¤) **in** - = *bind-spmf* - ¤ *split-def*)
(*fold result-def*; *transfer-prover*) **}**
moreover
have *spmf* (*game-multi' c-o c-a s*) *True* = *spmf* (*bind-spmf* (*spmf-of-set* {..<*q*}) (*body2 c-o c-a s*)) *True* ∗ *q*
if (*c-o*, *c-a*, *s*) ∈ *set-spmf init* **for** *c-o c-a s*
proof −
have *bnd*: *interaction-bounded-by* (*Not* ∘ *isl*) ($\mathcal{A}$ *c-a*) *q* **using** *bound that* **by** *blast*

have *bound-occ*: $j_s < q$ **if** *that*: $((), (id, Some\ j_s), s') \in set\text{-}spmf\ (?multi'\text{-}body\ c\text{-}o\ c\text{-}a\ s)$
for $s'\ id\ j_s$
proof −
have $id \leq q$
by(*rule oi-True.interaction-bounded-by′-exec-gpv-count*[*OF bnd that*, **where** *count=fst* ∘ *fst*, *simplified*])
(*auto simp add*: *eval-oracle′-def split*: *plus-oracle-split-asm option.split-asm*)
moreover let $?I = \lambda((id, occ), s').\ case\ occ\ of\ None \Rightarrow True \mid Some\ j_s \Rightarrow j_s < id$
have *callee-invariant* (†(*oracle c-o*) $\oplus_O$ *eval-oracle′ c-o c-a*) *?I*
by(*clarsimp simp add*: *split-def intro*!: *conjI*[*OF callee-invariant-extend-state-oracle-const′*])
(*unfold-locales*; *auto simp add*: *eval-oracle′-def split*: *option.split-asm*)
from *callee-invariant-on.exec-gpv-invariant*[*OF this that*] **have** $j_s < id$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

let *?M* = *measure* (*measure-spmf* (*?multi′-body c-o c-a s*))
have *spmf* (*game-multi′ c-o c-a s*) *True* = *?M* $\{(u, (id, j_0), s').\ j_0 \neq None\}$
by(*auto simp add*: *game-multi′-def map-spmf-conv-bind-spmf*[*symmetric*] *split-def spmf-conv-measure-spmf measure-map-spmf vimage-def*)
also have $\{(u, (id, j_0), s').\ j_0 \neq None\} =$
$\{((), (id, Some\ j_s), s') \mid j_s\ s'\ id.\ j_s < q\} \cup \{((), (id, Some\ j_s), s') \mid j_s\ s'\ id.\ j_s \geq q\}$
(**is** - = *?A* ∪ -) **by** *auto*
also have *?M* ... = *?M ?A*
by (*rule measure-spmf.measure-zero-union*)(*auto simp add*: *measure-spmf-zero-iff dest*: *bound-occ*)
also have ... = *measure* (*measure-spmf* (*pair-spmf* (*spmf-of-set* {..< *q*}) (*?multi′-body c-o c-a s*)))
$\{(j_s, (), (id, j_0), s') \mid j_s\ j_0\ s'\ id.\ j_0 = Some\ j_s\} * q$
(**is** - = *measure ?M′ ?B* ∗ -)
proof −
have $?B = \{(j_s, (), (id, j_0), s') \mid j_s\ j_0\ s'\ id.\ j_0 = Some\ j_s \wedge j_s < q\} \cup$
$\{(j_s, (), (id, j_0), s') \mid j_s\ j_0\ s'\ id.\ j_0 = Some\ j_s \wedge j_s \geq q\}$ (**is** - = *?Set1* ∪ *?Set2*)
by *auto*
then have *measure ?M′ ?B* = *measure ?M′* (*?Set1* ∪ *?Set2*) **by** *simp*
also have ... = *measure ?M′ ?Set1*
by (*rule measure-spmf.measure-zero-union*) (*auto simp add*: *measure-spmf-zero-iff*)
also have ... = $(\sum j \in \{0..<q\}.\ measure\ ?M'\ (\{j\} \times \{((), (id, Some\ j), s') \mid s'\ id.\ True\}))$
by(*subst measure-spmf.finite-measure-finite-Union*[*symmetric*])
(*auto intro*!: *arg-cong2*[**where** *f=measure*] *simp add*: *disjoint-family-on-def*)
also have ... = $(\sum j \in \{0..<q\}.\ 1\ /\ q * measure\ (measure\text{-}spmf\ (?multi'\text{-}body\ c\text{-}o\ c\text{-}a\ s))\ \{((), (id, Some\ j), s') \mid s'\ id.\ True\})$
by(*simp add*: *measure-pair-spmf-times spmf-conv-measure-spmf*[*symmetric*]

spmf-of-set)
also have ... = *1* / *q* ∗ *measure* (*measure-spmf* (*?multi′-body c-o c-a s*)) {((), (*id*, *Some* j_s), *s′*)|j_s *s′ id*. j_s < *q*}
unfolding *sum-distrib-left*[*symmetric*]
by(*subst measure-spmf.finite-measure-finite-Union*[*symmetric*])
(*auto intro*!: *arg-cong2*[**where** *f*=*measure*] *simp add*: *disjoint-family-on-def*)
finally show *?thesis* **by** *simp*
qed
also have *?B* = (λ(j_s, -, (-, j_0), -). j_0 = *Some* j_s) −‘ {*True*}
by (*auto simp add*: *vimage-def*)
also have *rw2*: *measure ?M′* ... = *spmf* (*bind-spmf* (*spmf-of-set* {..<*q*}) (*body2 c-o c-a s*)) *True*
by (*simp add*: *body2-def*[*abs-def*] *measure-map-spmf*[*symmetric*] *map-spmf-conv-bind-spmf split-def pair-spmf-alt-def spmf-conv-measure-spmf*[*symmetric*])
finally show *?thesis* **.**
qed
hence *spmf* (*bind-spmf init* (λ(*c-a*, *c-o*, *s*). *game-multi′ c-a c-o s*)) *True* = *spmf ?game2 True* ∗ *q*
unfolding *initialize-def spmf-bind*[**where** *p*=*init*]
by (*auto intro*!: *integral-cong-AE simp del*: *integral-mult-left-zero simp add*: *integral-mult-left-zero*[*symmetric*])

moreover
have *ord-spmf op* ⟶ (*body2 c-o c-a s* j_s) (*body3 c-o c-a s* j_s)
if *init*: (*c-o*, *c-a*, *s*) ∈ *set-spmf init* **and** j_s: j_s < *Suc q* **for** *c-o c-a s* j_s
proof −
define *oracle2′* **where** *oracle2′* ≡ λ(*b*, (*id*, *gs*), *s*) *guess*. *if id* = j_s *then do* {
b′ :: *bool* ← *eval c-o c-a s guess*;
return-spmf ((), (*Some b′*, (*Suc id*, *Some* (*guess*, *s*)), *s*))
} *else return-spmf* ((), (*b*, (*Suc id*, *gs*), *s*))

let *?R* = λ((*id1*, j_0), *s1*) (*b′*, (*id2*, *gs*), *s2*). *s1* = *s2* ∧ *id1* = *id2* ∧ (j_0 = *Some* j_s ⟶ *b′* = *Some True*) ∧ (*id2* ≤ j_s ⟶ *b′* = *None*)
from *init* **have** *rel-spmf* (*rel-prod op* = *?R*)
(*exec-gpv* (*extend-state-oracle* (*oracle c-o*) $\oplus_O$ *eval-oracle′ c-o c-a*) ($\mathcal{A}$ *c-a*) ((*0*, *None*), *s*))
(*exec-gpv* (*extend-state-oracle* (*extend-state-oracle* (*oracle c-o*)) $\oplus_O$ *oracle2′*) ($\mathcal{A}$ *c-a*) (*None*, (*0*, *None*), *s*))
by(*intro exec-gpv-oracle-bisim*[**where** *X*=*?R*])(*auto simp add*: *oracle2′-def eval-oracle′-def spmf-rel-map map-spmf-conv-bind-spmf*[*symmetric*] *rel-spmf-return-spmf2 lossless-eval o-def intro*!: *rel-spmf-reflI split*: *option.split-asm plus-oracle-split if-split-asm*)
then have *rel-spmf* (*op* ⟶) (*body2 c-o c-a s* j_s)
(*do* {
(-, *b′*, -, -) ← *exec-gpv* (††(*oracle c-o*) $\oplus_O$ *oracle2′*) ($\mathcal{A}$ *c-a*) (*None*, (*0*, *None*), *s*);
return-spmf (*b′* = *Some True*) })
(**is** *rel-spmf - - ?body2′*)
— We do not get equality here because the right hand side may return *True* even when the bad event has happened before the j_s-th iteration.

unfolding *body2-def* **by**(*rule rel-spmf-bindI*) *clarsimp*
also
let *?guess-oracle* = λ((*id*, *gs*), *s*) *guess*. *return-spmf* ((), (*Suc id*, *if id* = j_s *then Some* (*guess*, *s*) *else gs*), *s*)
let *?I* = λ(*idgs*, *s*). *case idgs of* (-, *None*) ⇒ *False* | (*i*, *Some* -) ⇒ j_s < *i*
interpret *I*: *callee-invariant-on* †(*oracle c-o*) $\oplus_O$ *?guess-oracle ?I* $\mathcal{I}$*-full*
by(*simp*)(*unfold-locales*; *auto split*: *option.split*)

let *?f* = λ*s*. *case snd* (*fst s*) *of None* ⇒ *return-spmf False* | *Some a* ⇒ *eval c-o c-a* (*snd a*) (*fst a*)
let *?X* = λj_s (*b1*, (*id1*, *gs1*), *s1*) (*b2*, (*id2*, *gs2*), *s2*). *b1* = *b2* ∧ *id1* = *id2* ∧ *gs1* = *gs2* ∧ *s1* = *s2* ∧ (*b2* = *None* ⟷ *gs2* = *None*) ∧ (*id2* ≤ j_s ⟶ *b2* = *None*)
have *?body2′* = *do* {
(*a*, *r*, *s*) ← *exec-gpv* (λ(*r*, *s*) *x*. *do* {
(*y*, *s′*) ← (†(*oracle c-o*) $\oplus_O$ *?guess-oracle*) *s x*;
if ?I s′ ∧ *r* = *None then map-spmf* (λ*r*. (*y*, *Some r*, *s′*)) (*?f s′*) *else return-spmf* (*y*, *r*, *s′*)
})
($\mathcal{A}$ *c-a*) (*None*, (*0*, *None*), *s*);
case r of None ⇒ *?f s* ⤜ *return-spmf* | *Some r′* ⇒ *return-spmf r′* }
unfolding *oracle2′-def spmf-rel-eq*[*symmetric*]
by(*rule rel-spmf-bindI*[*OF exec-gpv-oracle-bisim′*[**where** *X*=*?X* j_s]])
(*auto simp add*: *bind-map-spmf o-def spmf.map-comp split-beta conj-comms map-spmf-conv-bind-spmf*[*symmetric*] *spmf-rel-map rel-spmf-reflI cong*: *conj-cong split*: *plus-oracle-split*)
also have ... = *do* {
us′ ← *exec-gpv* (†(*oracle c-o*) $\oplus_O$ *?guess-oracle*) ($\mathcal{A}$ *c-a*) ((*0*, *None*), *s*);
(*b′* :: *bool*) ← *?f* (*snd us′*);
return-spmf b′ }
(**is** - = *?body2″*)
by(*rule I.exec-gpv-bind-materialize*[*symmetric*])(*auto split*: *plus-oracle-split-asm option.split-asm*)
also have ... = *do* {
us′ ← *exec-gpv-stop* (*lift-stop-oracle* (†(*oracle c-o*) $\oplus_O$ *?guess-oracle*)) ($\mathcal{A}$ *c-a*) ((*0*, *None*), *s*);
(*b′* :: *bool*) ← *?f* (*snd us′*);
return-spmf b′ }
supply *lift-stop-oracle-transfer*[*transfer-rule*] *gpv-stop-transfer*[*transfer-rule*] *exec-gpv-parametric′*[*transfer-rule*]
by *transfer simp*
also let *?S* = λ((*id1*, *gs1*), *s1*) ((*id2*, *gs2*), *s2*). *gs1* = *gs2* ∧ (*gs2* = *None* ⟶ *s1* = *s2* ∧ *id1* = *id2*) ∧ (*gs1* = *None* ⟷ *id1* ≤ j_s)
have *ord-spmf op* ⟶ ... (*exec-gpv-stop* ((λ((*id*, *gs*), *s*) *x*. *case gs of None* ⇒ *lift-stop-oracle* (†(*oracle c-o*)) ((*id*, *gs*), *s*) *x* | *Some* - ⇒ *return-spmf* (*None*, ((*id*, *gs*), *s*))) $\oplus_O{}^S$
(λ((*id*, *gs*), *s*) *guess*. *return-spmf* (*if id* ≥ j_s *then None else Some* (), (*Suc id*, *if id* = j_s *then Some* (*guess*, *s*) *else gs*), *s*)))
($\mathcal{A}$ *c-a*) ((*0*, *None*), *s*) ⤜

(λ*us'*. *case snd* (*fst* (*snd us'*)) *of None* $\Rightarrow$ *return-spmf False* | *Some a* $\Rightarrow$ *eval c-o c-a* (*snd a*) (*fst a*)))
unfolding *body3-def stop-oracle-def*
by(*rule ord-spmf-exec-gpv-stop*[**where** *stop* = λ((*id*, *guess*), -). *guess* $\neq$ *None* **and** *S*=*?S*, *THEN ord-spmf-bindI*])
(*auto split*: *prod.split-asm plus-oracle-split-asm split*!: *plus-oracle-stop-split simp del*: *not-None-eq simp add*: *spmf.map-comp o-def apfst-compose ord-spmf-map-spmf1 ord-spmf-map-spmf2 split-beta ord-spmf-return-spmf2 intro*!: *ord-spmf-reflI*)
also let *?X* = λ((*id*, *gs*), *s1*) (*idgs*, *s2*). *s1* = *s2* $\wedge$ (*case* (*gs*, *idgs*) *of* (*None*, *Inr id'*) $\Rightarrow$ *id'* = j_s − *id* $\wedge$ *id* $\leq$ j_s | (*Some gs*, *Inl gs'*) $\Rightarrow$ *gs* = *gs'* $\wedge$ *id* > j_s | - $\Rightarrow$ *False*)
have ... = *body3 c-o c-a s* j_s **unfolding** *body3-def spmf-rel-eq*[*symmetric*] *stop-oracle-def*
by(*rule exec-gpv-oracle-bisim'*[**where** *X*=*?X*, *THEN rel-spmf-bindI*])
(*auto split*: *option.split-asm plus-oracle-stop-split nat.splits split*!: *sum.split simp add*: *spmf-rel-map intro*!: *rel-spmf-reflI*)
finally show *?thesis* **by**(*rule pmf.rel-mono-strong*)(*auto elim*!: *option.rel-cases ord-option.cases*)
qed
{ then have *ord-spmf* (*op* $\longrightarrow$) *?game2 ?game3*
by(*clarsimp simp add*: *initialize-def intro*!: *ord-spmf-bind-reflI*)
also
let *?X* = λ(*gsid*, *s*) (*gid*, *s'*). *s* = *s'* $\wedge$ *rel-sum* (λ(*g*, *s1*) *g'*. *g* = *g'* $\wedge$ *s1* = *s'*) *op* = *gsid gid*
have *rel-spmf* (*op* $\longrightarrow$) *?game3* (*game-single* (*reduction q* $\mathcal{A}$))
unfolding *body3-def stop-oracle-def game-single-def reduction-def split-def initialize-def*
apply(*clarsimp simp add*: *bind-map-spmf exec-gpv-bind exec-gpv-inline intro*!: *rel-spmf-bind-reflI*)
apply(*rule rel-spmf-bindI*[*OF exec-gpv-oracle-bisim'*[**where** *X*=*?X*]])
apply(*auto split*: *plus-oracle-stop-split elim*!: *rel-sum.cases simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *split-def spmf-rel-map rel-spmf-reflI rel-spmf-return-spmf1 lossless-eval split*: *nat.split*)
done
finally have *ord-spmf op* $\longrightarrow$ *?game2* (*game-single* (*reduction q* $\mathcal{A}$))
by(*rule pmf.rel-mono-strong*)(*auto elim*!: *option.rel-cases ord-option.cases*)
from *this*[*THEN ord-spmf-measureD*, *of* {*True*}]
have *spmf ?game2 True* $\leq$ *spmf* (*game-single* (*reduction q* $\mathcal{A}$)) *True* **unfolding** *spmf-conv-measure-spmf*
by(*rule ord-le-eq-trans*)(*auto intro*: *arg-cong2*[**where** *f*=*measure*]) **}**
ultimately show *?thesis* **unfolding** *advantage-multi-def advantage-single-def*
by(*simp add*: *mult-right-mono*)
qed

end

end

1.13 Unpredictable function

theory *Unpredictable-Function* **imports**
  Guessing-Many-One
begin

locale *upf* =
  fixes *key-gen* :: *'key spmf*
  and *hash* :: *'key* ⇒ *'x* ⇒ *'hash*
begin

type-synonym (*'x'*, *'hash'*) *adversary* = (*unit*, *'x'* + (*'x'* × *'hash'*), *'hash'* + *unit*) *gpv*

definition *oracle-hash* :: *'key* ⇒ (*'x*, *'hash*, *'x set*) *callee*
where
  oracle-hash k = (λ*L y*. *do* {
    let t = *hash k y*;
    let L = *insert y L*;
    return-spmf (*t*, *L*)
  })

definition *oracle-flag* :: *'key* ⇒ (*'x* × *'hash*, *unit*, *bool* × *'x set*) *callee*
where
  oracle-flag = (λ*key* (*flg*, *L*) (*y*, *t*).
    return-spmf ((), (*flg* ∨ (*t* = (*hash key y*) ∧ *y* ∉ *L*), *L*)))

abbreviation *oracle* :: *'key* ⇒ (*'x* + *'x* × *'hash*, *'hash* + *unit*, *bool* × *'x set*) *callee*
where *oracle key* ≡ †(*oracle-hash key*) $\oplus_O$ *oracle-flag key*

definition *game* :: (*'x*, *'hash*) *adversary* ⇒ *bool spmf*
where
  game 𝒜 = *do* {
    key ← *key-gen*;
    (-, (*flag'*, *L'*)) ← *exec-gpv* (*oracle key*) 𝒜 (*False*, {});
    return-spmf flag'
  }

definition *advantage* :: (*'x*, *'hash*) *adversary* ⇒ *real*
where *advantage* 𝒜 = *spmf* (*game* 𝒜) *True*

type-synonym (*'x'*, *'hash'*) *adversary1* = (*'x'* × *'hash'*, *'x'*, *'hash'*) *gpv*

definition *game1* :: (*'x*, *'hash*) *adversary1* ⇒ *bool spmf*
where
  game1 𝒜 = *do* {
    key ← *key-gen*;
    ((*m*, *h*), *L*) ← *exec-gpv* (*oracle-hash key*) 𝒜 {};
    return-spmf (*h* = *hash key m* ∧ *m* ∉ *L*)
  }

```
definition advantage1 :: ('x, 'hash) adversary1 ⇒ real
  where advantage1 𝒜 = spmf (game1 𝒜) True

lemma advantage-advantage1:
  assumes bound: interaction-bounded-by (Not ∘ isl) 𝒜 q
  shows advantage 𝒜 ≤ advantage1 (guessing-many-one.reduction q (λ- :: unit.
𝒜) ()) ∗ q
proof −
  let ?init = map-spmf (λkey. (key, (), {})) key-gen
  let ?oracle = λkey . oracle-hash key
  let ?eval = λkey (- :: unit) L (x, h). return-spmf (h = hash key x ∧ x ∉ L)

  interpret guessing-many-one ?init ?oracle ?eval .

  have [simp]: oracle-flag key = eval-oracle key () for key
    by(simp add: oracle-flag-def eval-oracle-def fun-eq-iff)
  have game 𝒜 = game-multi (λ-. 𝒜)
   by(auto simp add: game-multi-def game-def bind-map-spmf intro!: bind-spmf-cong[OF
refl])
   hence advantage 𝒜 = advantage-multi (λ-. 𝒜) by(simp add: advantage-def
advantage-multi-def)
  also have ... ≤ advantage-single (reduction q (λ-. 𝒜)) ∗ q using bound
    by(rule many-single-reduction)(auto simp add: oracle-hash-def)
  also have advantage-single (reduction q (λ-. 𝒜)) = advantage1 (reduction q (λ-.
𝒜) ()) for 𝒜
    unfolding advantage1-def advantage-single-def
   by(auto simp add: game1-def game-single-def bind-map-spmf o-def intro!: bind-spmf-cong[OF
refl] arg-cong2[where f=spmf])
  finally show ?thesis .
qed

end

end


theory Security-Spec imports
  Diffie-Hellman
  IND-CCA2
  IND-CCA2-sym
  IND-CPA
  IND-CPA-PK
  IND-CPA-PK-Single
  SUF-CMA
  Pseudo-Random-Function
  Pseudo-Random-Permutation
  Unpredictable-Function
begin
```

end

2 Cryptographic constructions and their security

theory *Elgamal* **imports**
  CryptHOL.Cyclic-Group-SPMF
  CryptHOL.Computational-Model
  Diffie-Hellman
  IND-CPA-PK-Single
  CryptHOL.Negligible
begin

2.1 Elgamal encryption scheme

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

type-synonym *'grp' pub-key* = *'grp'*
type-synonym *'grp' priv-key* = *nat*
type-synonym *'grp' plain* = *'grp'*
type-synonym *'grp' cipher* = *'grp'* × *'grp'*

definition *key-gen* :: (*'grp pub-key* × *'grp priv-key*) *spmf*
where
  key-gen = *do* {
    $x \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
    return-spmf (**g** (ˆ) *x*, *x*)
  }

lemma *key-gen-alt*:
  key-gen = *map-spmf* (λx. (**g** (ˆ) *x*, *x*)) (*sample-uniform* (*order* $\mathcal{G}$))
by(*simp add*: *map-spmf-conv-bind-spmf key-gen-def*)

definition *aencrypt* :: *'grp pub-key* ⇒ *'grp* ⇒ *'grp cipher spmf*
where
  aencrypt α *msg* = *do* {
    $y \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
    return-spmf (**g** (ˆ) *y*, (α (ˆ) *y*) ⊗ *msg*)
  }

lemma *aencrypt-alt*:
  aencrypt α *msg* = *map-spmf* (λy. (**g** (ˆ) *y*, (α (ˆ) *y*) ⊗ *msg*)) (*sample-uniform* (*order* $\mathcal{G}$))
by(*simp add*: *map-spmf-conv-bind-spmf aencrypt-def*)

definition *adecrypt* :: *'grp priv-key* ⇒ *'grp cipher* ⇒ *'grp option*
where

adecrypt x = ($\lambda(\beta, \zeta)$. *Some* ($\zeta \otimes$ (*inv* (β (ˆ) *x*))))

abbreviation *valid-plains* :: *'grp* ⇒ *'grp* ⇒ *bool*
where *valid-plains msg1 msg2* ≡ *msg1* ∈ *carrier* $\mathcal{G}$ ∧ *msg2* ∈ *carrier* $\mathcal{G}$

sublocale *ind-cpa*: *ind-cpa key-gen aencrypt adecrypt valid-plains* **.**
sublocale *ddh*: *ddh* $\mathcal{G}$ **.**

fun *elgamal-adversary* :: (*'grp pub-key*, *'grp plain*, *'grp cipher*, *'state*) *ind-cpa.adversary* ⇒ *'grp ddh.adversary*
where
  elgamal-adversary ($\mathcal{A}1$, $\mathcal{A}2$) α β γ = *TRY do* {
    b ← *coin-spmf*;
    ((*msg1*, *msg2*), σ) ← $\mathcal{A}1$ α;
    (∗ *have to check that the attacker actually sends two elements from the group*; *otherwise flip a coin* ∗)
    - :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
    guess ← $\mathcal{A}2$ (β, $\gamma \otimes$ (*if b then msg1 else msg2*)) σ;
    return-spmf (*guess* = *b*)
  } *ELSE coin-spmf*

end

locale *elgamal* = *elgamal-base* + *cyclic-group* $\mathcal{G}$ +
  assumes *finite-group*: *finite* (*carrier* $\mathcal{G}$)
begin

theorem *advantage-elgamal*: *ind-cpa.advantage* $\mathcal{A}$ = *ddh.advantage* (*elgamal-adversary* $\mathcal{A}$)
  including *monad-normalisation*
proof −
  obtain $\mathcal{A}1$ **and** $\mathcal{A}2$ **where** $\mathcal{A}$ = ($\mathcal{A}1$, $\mathcal{A}2$) **by**(*cases* $\mathcal{A}$)
   note [*simp*] = *this order-gt-0-iff-finite finite-group try-spmf-bind-out split-def o-def spmf-of-set bind-map-spmf weight-spmf-le-1 scale-bind-spmf bind-spmf-const*
    and [*cong*] = *bind-spmf-cong-simp*
  have *ddh.ddh-1* (*elgamal-adversary* $\mathcal{A}$) = *TRY do* {
      x ← *sample-uniform* (*order* $\mathcal{G}$);
      y ← *sample-uniform* (*order* $\mathcal{G}$);
      ((*msg1*, *msg2*), σ) ← $\mathcal{A}1$ (**g** (ˆ) *x*);
      - :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
      b ← *coin-spmf*;
      z ← *map-spmf* (λz. **g** (ˆ) *z* ⊗ (*if b then msg1 else msg2*)) (*sample-uniform* (*order* $\mathcal{G}$));
      guess ← $\mathcal{A}2$ (**g** (ˆ) *y*, *z*) σ;
      return-spmf (*guess* ⟷ *b*)
    } *ELSE coin-spmf*
    by(*simp add*: *ddh.ddh-1-def*)
  also have ... = *TRY do* {
      x ← *sample-uniform* (*order* $\mathcal{G}$);

$y \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
((*msg1*, *msg2*), σ) $\leftarrow \mathcal{A}1$ (**g** (^) x);
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains msg1 msg2*);
$z \leftarrow$ *map-spmf* (λz. **g** (^) z) (*sample-uniform* (*order* $\mathcal{G}$));
guess $\leftarrow \mathcal{A}2$ (**g** (^) y, z) σ;
map-spmf (*op* = *guess*) *coin-spmf*
} *ELSE coin-spmf*
by(*simp add*: *sample-uniform-one-time-pad map-spmf-conv-bind-spmf*[**where** *p*=*coin-spmf*])
also have ... = *coin-spmf*
by(*simp add*: *map-eq-const-coin-spmf try-bind-spmf-lossless2'*)
also have *ddh.ddh-0* (*elgamal-adversary* $\mathcal{A}$) = *ind-cpa.ind-cpa* $\mathcal{A}$
by(*simp add*: *ddh.ddh-0-def IND-CPA-PK-Single.ind-cpa.ind-cpa-def key-gen-def aencrypt-def nat-pow-pow eq-commute*)
ultimately show *?thesis* **by**(*simp add*: *ddh.advantage-def ind-cpa.advantage-def*)
qed

end

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

sublocale *elgamal* $\mathcal{G}$ η **for** η **by**(*simp add*: *elgamal*)

theorem *elgamal-secure*:
negligible ($\lambda\eta$. *ind-cpa.advantage* η ($\mathcal{A}$ η)) **if** *negligible* ($\lambda\eta$. *ddh.advantage* η (*elgamal-adversary* η ($\mathcal{A}$ η)))
by(*simp add*: *advantage-elgamal that*)

end

context *elgamal-base* **begin**

lemma *lossless-key-gen* [*simp*]: *lossless-spmf* (*key-gen*) $\longleftrightarrow$ *0* < *order* $\mathcal{G}$
by(*simp add*: *key-gen-def Let-def*)

lemma *lossless-aencrypt* [*simp*]:
lossless-spmf (*aencrypt key plain*) $\longleftrightarrow$ *0* < *order* $\mathcal{G}$
by(*simp add*: *aencrypt-def Let-def*)

lemma *lossless-elgamal-adversary*:
$[\![$ *ind-cpa.lossless* $\mathcal{A}$; *0* < *order* $\mathcal{G}$ $]\!]$
$\Longrightarrow$ *ddh.lossless* (*elgamal-adversary* $\mathcal{A}$)
by(*cases* $\mathcal{A}$)(*simp add*: *ddh.lossless-def ind-cpa.lossless-def Let-def split-def*)

end

end

2.2 Hashed Elgamal in the Random Oracle Model

theory *Hashed-Elgamal* **imports**
CryptHOL.GPV-Bisim
CryptHOL.Cyclic-Group-SPMF
CryptHOL.List-Bits
IND-CPA-PK
Diffie-Hellman
begin

type-synonym *bitstring* = *bool list*

locale *hash-oracle* = **fixes** *len* :: *nat* **begin**

type-synonym *'a state* = *'a* ⇀ *bitstring*

definition *oracle* :: *'a state* ⇒ *'a* ⇒ (*bitstring* × *'a state*) *spmf*
where
oracle σ x =
(*case σ x of None* ⇒ *do* {
bs ← *spmf-of-set* (*nlists UNIV len*);
return-spmf (*bs*, *σ*(*x* ↦ *bs*))
} | *Some bs* ⇒ *return-spmf* (*bs*, *σ*))

abbreviation (*input*) *initial* :: *'a state* **where** *initial* ≡ *empty*

inductive *invariant* :: *'a state* ⇒ *bool*
where
invariant: ⟦ *finite* (*dom σ*); *length* ‘ *ran σ* ⊆ {*len*} ⟧ ⟹ *invariant σ*

lemma *invariant-initial* [*simp*]: *invariant initial*
by(*rule invariant.intros*) *auto*

lemma *invariant-update* [*simp*]: ⟦ *invariant σ*; *length bs* = *len* ⟧ ⟹ *invariant* (*σ*(*x* ↦ *bs*))
by(*auto simp add*: *invariant.simps ran-def*)

lemma *invariant* [*intro*!, *simp*]: *callee-invariant oracle invariant*
by *unfold-locales*(*simp-all add*: *oracle-def in-nlists-UNIV split*: *option.split-asm*)

lemma *invariant-in-dom* [*simp*]: *callee-invariant oracle* (*λσ. x* ∈ *dom σ*)
by *unfold-locales*(*simp-all add*: *oracle-def split*: *option.split-asm*)

lemma *lossless-oracle* [*simp*]: *lossless-spmf* (*oracle σ x*)
by(*simp add*: *oracle-def split*: *option.split*)

lemma *card-dom-state*:

assumes σ': $(x, \sigma') \in$ *set-spmf* (*exec-gpv oracle gpv* σ)
and *ibound*: *interaction-any-bounded-by gpv n*
shows *card* (*dom* σ') $\leq n +$ *card* (*dom* σ)
proof(*cases finite* (*dom* σ))
case *True*
interpret *callee-invariant-on oracle* $\lambda\sigma$. *finite* (*dom* σ) $\mathcal{I}$*-full*
by *unfold-locales*(*auto simp add*: *oracle-def split*: *option.split-asm*)
from *ibound* σ' - - - *True* **show** *?thesis*
by(*rule interaction-bounded-by'-exec-gpv-count*)(*auto simp add*: *oracle-def card-insert-if simp del*: *fun-upd-apply split*: *option.split-asm*)
next
case *False*
interpret *callee-invariant-on oracle* $\lambda\sigma'$. *dom* $\sigma \subseteq$ *dom* σ' $\mathcal{I}$*-full*
by *unfold-locales*(*auto simp add*: *oracle-def split*: *option.split-asm*)
from σ' **have** *dom* $\sigma \subseteq$ *dom* σ' **by**(*rule exec-gpv-invariant*) *simp-all*
with *False* **have** *infinite* (*dom* σ') **by**(*auto intro*: *finite-subset*)
with *False* **show** *?thesis* **by** *simp*
qed

end

locale *elgamal-base* =
fixes $\mathcal{G}$ :: *'grp cyclic-group* (**structure**)
and *len-plain* :: *nat*
begin

sublocale *hash*: *hash-oracle len-plain* **.**
abbreviation *hash* :: *'grp* $\Rightarrow$ (*bitstring*, *'grp*, *bitstring*) *gpv*
where *hash x* $\equiv$ *Pause x Done*

type-synonym *'grp' pub-key* = *'grp'*
type-synonym *'grp' priv-key* = *nat*
type-synonym *plain* = *bitstring*
type-synonym *'grp' cipher* = *'grp'* $\times$ *bitstring*

definition *key-gen* :: (*'grp pub-key* $\times$ *'grp priv-key*) *spmf*
where
key-gen = *do* {
$x \leftarrow$ *sample-uniform* (*order* $\mathcal{G}$);
return-spmf (**g** (ˆ) *x*, *x*)
}

definition *aencrypt* :: *'grp pub-key* $\Rightarrow$ *plain* $\Rightarrow$ (*'grp cipher*, *'grp*, *bitstring*) *gpv*
where
aencrypt α *msg* = *do* {
$y \leftarrow$ *lift-spmf* (*sample-uniform* (*order* $\mathcal{G}$));
$h \leftarrow$ *hash* (α (ˆ) *y*);
Done (**g** (ˆ) *y*, *h* [$\oplus$] *msg*)
}

definition *adecrypt* :: *'grp priv-key* ⇒ *'grp cipher* ⇒ (*plain*, *'grp*, *bitstring*) *gpv*
where
  adecrypt x = (λ(β, ζ). *do* {
    h ← *hash* (β (^) *x*);
    Done (ζ [⊕] *h*)
  })

definition *valid-plains* :: *plain* ⇒ *plain* ⇒ *bool*
where *valid-plains msg1 msg2* ⟷ *length msg1* = *len-plain* ∧ *length msg2* = *len-plain*

lemma *lossless-aencrypt* [*simp*]: *lossless-gpv* 𝓘 (*aencrypt* α *msg*) ⟷ *0* < *order* 𝒢
by(*simp add*: *aencrypt-def Let-def*)

lemma *interaction-bounded-by-aencrypt* [*interaction-bound*, *simp*]:
  interaction-bounded-by (λ-. *True*) (*aencrypt* α *msg*) *1*
unfolding *aencrypt-def* **by** *interaction-bound*(*simp add*: *one-enat-def SUP-le-iff*)

sublocale *ind-cpa*: *ind-cpa-pk lift-spmf key-gen aencrypt adecrypt valid-plains* **.**
sublocale *lcdh*: *lcdh* 𝒢 **.**

fun *elgamal-adversary*
  :: (*'grp pub-key*, *plain*, *'grp cipher*, *'grp*, *bitstring*, *'state*) *ind-cpa.adversary*
  ⇒ *'grp lcdh.adversary*
where
  elgamal-adversary (𝒜*1*, 𝒜*2*) α β = *do* {
    (((*msg1*, *msg2*), σ), *s*) ← *exec-gpv hash.oracle* (𝒜*1* α) *hash.initial*;
    (∗ *have to check that the attacker actually sends an element from the group*; *otherwise stop early* ∗)
    TRY do {
      - :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
      h' ← *spmf-of-set* (*nlists UNIV len-plain*);
      (*guess*, *s'*) ← *exec-gpv hash.oracle* (𝒜*2* (β, *h'*) σ) *s*;
      return-spmf (*dom s'*)
    } *ELSE return-spmf* (*dom s*)
  }

end

locale *elgamal* = *elgamal-base* +
  assumes *cyclic-group*: *cyclic-group* 𝒢
begin

interpretation *cyclic-group* 𝒢 **by**(*fact cyclic-group*)

lemma *advantage-elgamal*:
  includes *lifting-syntax*

assumes *finite-group*: *finite* (*carrier* $\mathcal{G}$)
and *lossless*: *ind-cpa.lossless* $\mathcal{A}$
shows *ind-cpa.advantage hash.oracle hash.initial* $\mathcal{A} \leq$ *lcdh.advantage* (*elgamal-adversary* $\mathcal{A}$)
proof −
note [*cong del*] = *if-weak-cong* **and** [*split del*] = *if-split*
and [*simp*] = *map-lift-spmf gpv.map-id lossless-weight-spmfD map-spmf-bind-spmf bind-spmf-const*
obtain $\mathcal{A}1$ $\mathcal{A}2$ **where** $\mathcal{A}$ [*simp*]: $\mathcal{A}$ = ($\mathcal{A}1$, $\mathcal{A}2$) **by**(*cases* $\mathcal{A}$)

interpret *cyclic-group*: *cyclic-group* $\mathcal{G}$ **by**(*rule cyclic-group*)
from *finite-group* **have** [*simp*]: *order* $\mathcal{G}$ > *0* **using** *order-gt-0-iff-finite* **by**(*simp*)

from *lossless* **have** *lossless1* [*simp*]: $\bigwedge$*pk*. *lossless-gpv* $\mathcal{I}$*-full* ($\mathcal{A}1$ *pk*)
and *lossless2* [*simp*]: $\bigwedge\sigma$ *cipher*. *lossless-gpv* $\mathcal{I}$*-full* ($\mathcal{A}2$ σ *cipher*)
by(*auto simp add*: *ind-cpa.lossless-def*)

We change the adversary's oracle to record the queries made by the adversary

def *hash-oracle′* ≡ λσ *x*. *do* {
h ← *hash x*;
Done (*h*, *insert x* σ)
}
have [*simp*]: *lossless-gpv* $\mathcal{I}$*-full* (*hash-oracle′* σ *x*) **for** σ *x* **by**(*simp add*: *hash-oracle′-def*)
have [*simp*]: *lossless-gpv* $\mathcal{I}$*-full* (*inline hash-oracle′* ($\mathcal{A}1$ α) *s*) **for** α *s*
by(*rule lossless-inline*[**where** $\mathcal{I}$=$\mathcal{I}$*-full*]) *simp-all*
def *game0* ≡ *TRY do* {
(*pk*, *-*) ← *lift-spmf key-gen*;
b ← *lift-spmf coin-spmf*;
(((*msg1*, *msg2*), σ), *s*) ← *inline hash-oracle′* ($\mathcal{A}1$ *pk*) {};
assert-gpv (*valid-plains msg1 msg2*);
cipher ← *aencrypt pk* (*if b then msg1 else msg2*);
(*guess*, *s′*) ← *inline hash-oracle′* ($\mathcal{A}2$ *cipher* σ) *s*;
Done (*guess* = *b*)
} *ELSE lift-spmf coin-spmf*
{ def *cr* ≡ λ*-* :: *unit*. λ*-* :: *′a set*. *True*
have [*transfer-rule*]: *cr* () {} **by**(*simp add*: *cr-def*)
have [*transfer-rule*]: (*op* = ===> *cr* ===> *cr*) (λ*-* σ. σ) *insert* **by**(*simp add*: *rel-fun-def cr-def*)
have [*transfer-rule*]: (*cr* ===> *op* = ===> *rel-gpv* (*rel-prod op* = *cr*) *op* =) *id-oracle hash-oracle′*
unfolding *hash-oracle′-def id-oracle-def*[*abs-def*] *bind-gpv-Pause bind-rpv-Done*
by *transfer-prover*
have *ind-cpa.ind-cpa* $\mathcal{A}$ = *game0* **unfolding** *game0-def* $\mathcal{A}$ *ind-cpa-pk.ind-cpa.simps*
by(*transfer fixing*: $\mathcal{G}$ *len-plain* $\mathcal{A}1$ $\mathcal{A}2$)(*simp add*: *bind-map-gpv o-def ind-cpa-pk.ind-cpa.simps split-def*) **}**
note *game0* = *this*
have *game0-alt-def*: *game0* = *do* {
x ← *lift-spmf* (*sample-uniform* (*order* $\mathcal{G}$));

```
    b ← lift-spmf coin-spmf;
    (((msg1, msg2), σ), s) ← inline hash-oracle′ (𝒜1 (g (^) x)) {};
    TRY do {
      - :: unit ← assert-gpv (valid-plains msg1 msg2);
      cipher ← aencrypt (g (^) x) (if b then msg1 else msg2);
      (guess, s′) ← inline hash-oracle′ (𝒜2 cipher σ) s;
      Done (guess = b)
    } ELSE lift-spmf coin-spmf
  }
  by(simp add: split-def game0-def key-gen-def lift-spmf-bind-spmf bind-gpv-assoc
try-gpv-bind-lossless[symmetric])

 def hash-oracle″ ≡ λ(s, σ) (x :: ′a). do {
    (h, σ′) ← case σ x of
       None ⇒ bind-spmf (spmf-of-set (nlists UNIV len-plain)) (λbs. return-spmf
(bs, σ(x ↦ bs)))
      | Some (bs :: bitstring) ⇒ return-spmf (bs, σ);
    return-spmf (h, insert x s, σ′)
  }
 have *: exec-gpv hash.oracle (inline hash-oracle′ 𝒜 s) σ =
   map-spmf (λ(a, b, c). ((a, b), c)) (exec-gpv hash-oracle″ 𝒜 (s, σ)) for 𝒜 σ s
  by(simp add: hash-oracle′-def hash-oracle″-def hash.oracle-def Let-def exec-gpv-inline
exec-gpv-bind o-def split-def cong del: option.case-cong-weak)
 have [simp]: lossless-spmf (hash-oracle″ s plain) for s plain
   by(simp add: hash-oracle″-def Let-def split: prod.split option.split)
 have [simp]: lossless-spmf (exec-gpv hash-oracle″ (𝒜1 α) s) for s α
   by(rule lossless-exec-gpv[where ℐ=ℐ-full]) simp-all
 have [simp]: lossless-spmf (exec-gpv hash-oracle″ (𝒜2 σ cipher) s) for σ cipher
s
   by(rule lossless-exec-gpv[where ℐ=ℐ-full]) simp-all

 let ?sample = λf. bind-spmf (sample-uniform (order 𝒢)) (λx. bind-spmf (sample-uniform
(order 𝒢)) (f x))
 def game1 ≡ λ(x :: nat) (y :: nat). do {
    b ← coin-spmf;
    (((msg1, msg2), σ), (s, s-h)) ← exec-gpv hash-oracle″ (𝒜1 (g (^) x)) ({},
hash.initial);
    TRY do {
      - :: unit ← assert-spmf (valid-plains msg1 msg2);
      (h, s-h′) ← hash.oracle s-h (g (^) (x * y));
      let cipher = (g (^) y, h [⊕] (if b then msg1 else msg2));
      (guess, (s′, s-h″)) ← exec-gpv hash-oracle″ (𝒜2 cipher σ) (s, s-h′);
      return-spmf (guess = b, g (^) (x * y) ∈ s′)
    } ELSE do {
      b ← coin-spmf;
      return-spmf (b, g (^) (x * y) ∈ s)
    }
  }
 have game01: run-gpv hash.oracle game0 hash.initial = map-spmf fst (?sample
```

game1)
apply(*simp add*: *exec-gpv-bind split-def bind-gpv-assoc aencrypt-def game0-alt-def game1-def o-def bind-map-spmf if-distribs* $*$ *try-bind-assert-gpv try-bind-assert-spmf lossless-inline*[**where** $\mathcal{I}$=$\mathcal{I}$*-full*] *bind-rpv-def nat-pow-pow del*: *bind-spmf-const*)
including *monad-normalisation* **by**(*simp add*: *bind-rpv-def nat-pow-pow*)

def *game2* $\equiv$ λ(*x* :: *nat*) (*y* :: *nat*). *do* {
b $\leftarrow$ *coin-spmf*;
(((*msg1*, *msg2*), σ), (*s*, *s-h*)) $\leftarrow$ *exec-gpv hash-oracle''* ($\mathcal{A}$*1* (**g** (ˆ) *x*)) ({}, *hash.initial*);
TRY do {
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains msg1 msg2*);
h $\leftarrow$ *spmf-of-set* (*nlists UNIV len-plain*);
($*$ *We do not do the lookup in s-h here, so the rest differs only if the adversary guessed y* $*$)
let cipher = (**g** (ˆ) *y*, *h* [$\oplus$] (*if b then msg1 else msg2*));
(*guess*, (*s'*, *s-h'*)) $\leftarrow$ *exec-gpv hash-oracle''* ($\mathcal{A}$*2 cipher* σ) (*s*, *s-h*);
return-spmf (*guess* = *b*, **g** (ˆ) (*x* $*$ *y*) $\in$ *s'*)
} *ELSE do* {
b $\leftarrow$ *coin-spmf*;
return-spmf (*b*, **g** (ˆ) (*x* $*$ *y*) $\in$ *s*)
}
}
interpret *inv''*: *callee-invariant-on hash-oracle''* λ(*s*, *s-h*). *s* = *dom s-h* $\mathcal{I}$*-full*
by *unfold-locales*(*auto simp add*: *hash-oracle''-def split*: *option.split-asm if-split*)
have *in-encrypt-oracle*: *callee-invariant hash-oracle''* (λ(*s*, -). *x* $\in$ *s*) **for** *x*
by *unfold-locales*(*auto simp add*: *hash-oracle''-def*)

{ fix *x y* :: *nat*
let *?bad* = λ(*s*, *s-h*). **g** (ˆ) (*x* $*$ *y*) $\in$ *s*
let *?X* = λ(*s*, *s-h*) (*s'*, *s-h'*). *s* = *dom s-h* $\wedge$ *s'* = *s* $\wedge$ *s-h* = *s-h'*(**g** (ˆ) (*x* $*$ *y*) := *None*)
have *bisim*:
rel-spmf (λ(*a*, *s1'*) (*b*, *s2'*). *?bad s1'* = *?bad s2'* $\wedge$ ($\neg$ *?bad s2'* $\longrightarrow$ *a* = *b* $\wedge$ *?X s1' s2'*))
(*hash-oracle'' s1 plain*) (*hash-oracle'' s2 plain*)
if *?X s1 s2* **for** *s1 s2 plain* **using** *that*
by(*auto split*: *prod.splits intro*!: *rel-spmf-bind-reflI simp add*: *hash-oracle''-def rel-spmf-return-spmf2 fun-upd-twist split*: *option.split dest*!: *fun-upd-eqD*)
have *inv*: *callee-invariant hash-oracle'' ?bad*
by(*unfold-locales*)(*auto simp add*: *hash-oracle''-def split*: *option.split-asm*)
have *rel-spmf* (λ(*win*, *bad*) (*win'*, *bad'*). *bad* = *bad'* $\wedge$ ($\neg$ *bad'* $\longrightarrow$ *win* = *win'*)) (*game2 x y*) (*game1 x y*)
unfolding *game1-def game2-def*
apply(*clarsimp simp add*: *split-def o-def hash.oracle-def rel-spmf-bind-reflI if-distribs intro*!: *rel-spmf-bind-reflI simp del*: *bind-spmf-const*)
apply(*rule rel-spmf-try-spmf*)
subgoal for *b msg1 msg2* σ *s s-h*
apply(*rule rel-spmf-bind-reflI*)

apply(*drule inv''.exec-gpv-invariant*; *clarsimp*)
apply(*cases s-h* (**g** (ˆ) (*x* ∗ *y*)))
subgoal — case *None*
apply(*clarsimp intro*!: *rel-spmf-bind-reflI*)
apply(*rule rel-spmf-bindI*)
apply(*rule exec-gpv-oracle-bisim-bad-full*[*OF* - - *bisim inv inv*, **where** *R*=λ(*x*, *s1*) (*y*, *s2*). *?bad s1* = *?bad s2* ∧ (¬ *?bad s2* ⟶ *x* = *y*)]; *clarsimp simp add*: *fun-upd-idem*; *fail*)
apply *clarsimp*
done
subgoal by(*auto intro*!: *rel-spmf-bindI1 rel-spmf-bindI2 lossless-exec-gpv*[**where** ℐ=ℐ*-full*] *dest*!: *callee-invariant-on.exec-gpv-invariant*[*OF in-encrypt-oracle*])
done
subgoal by(*rule rel-spmf-reflI*) *simp*
done }
hence *rel-spmf* (λ(*win*, *bad*) (*win'*, *bad'*). (*bad* ⟷ *bad'*) ∧ (¬ *bad'* ⟶ *win* ⟷ *win'*)) (*?sample game2*) (*?sample game1*)
by(*intro rel-spmf-bind-reflI*)
hence |*measure* (*measure-spmf* (*?sample game2*)) {(*x*, -). *x*} − *measure* (*measure-spmf* (*?sample game1*)) {(*y*, -). *y*}|
≤ *measure* (*measure-spmf* (*?sample game2*)) {(-, *bad*). *bad*}
unfolding *split-def* **by**(*rule fundamental-lemma*)
moreover have *measure* (*measure-spmf* (*?sample game2*)) {(*x*, -). *x*} = *spmf* (*map-spmf fst* (*?sample game2*)) *True*
and *measure* (*measure-spmf* (*?sample game1*)) {(*y*, -). *y*} = *spmf* (*map-spmf fst* (*?sample game1*)) *True*
and *measure* (*measure-spmf* (*?sample game2*)) {(-, *bad*). *bad*} = *spmf* (*map-spmf snd* (*?sample game2*)) *True*
unfolding *spmf-conv-measure-spmf measure-map-spmf* **by**(*rule arg-cong2*[**where** *f*=*measure*]; *fastforce*)+
ultimately have *hop23*: |*spmf* (*map-spmf fst* (*?sample game2*)) *True* − *spmf* (*map-spmf fst* (*?sample game1*)) *True*| ≤ *spmf* (*map-spmf snd* (*?sample game2*)) *True* **by** *simp*

def *game3* ≡ λ*f* :: - ⇒ - ⇒ - ⇒ *bitstring spmf* ⇒ - *spmf*. λ(*x* :: *nat*) (*y* :: *nat*). *do* {
b ← *coin-spmf*;
(((*msg1*, *msg2*), σ), (*s*, *s-h*)) ← *exec-gpv hash-oracle''* (𝒜*1* (**g** (ˆ) *x*)) ({}, *hash.initial*);
TRY do {
- :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
h' ← *f b msg1 msg2* (*spmf-of-set* (*nlists UNIV len-plain*));
let cipher = (**g** (ˆ) *y*, *h'*);
(*guess*, (*s'*, *s-h'*)) ← *exec-gpv hash-oracle''* (𝒜*2 cipher* σ) (*s*, *s-h*);
return-spmf (*guess* = *b*, **g** (ˆ) (*x* ∗ *y*) ∈ *s'*)
} *ELSE do* {
b ← *coin-spmf*;
return-spmf (*b*, **g** (ˆ) (*x* ∗ *y*) ∈ *s*)
}

```
  }
  let ?f = λb msg1 msg2. map-spmf (λh. (if b then msg1 else msg2) [⊕] h)
  have game2 x y = game3 ?f x y for x y
   unfolding game2-def game3-def by(simp add: Let-def bind-map-spmf xor-list-commute
o-def nat-pow-pow)
  also have game3 ?f x y = game3 (λ- - - x. x) x y for x y
    unfolding game3-def
   by(auto intro!: try-spmf-cong bind-spmf-cong[OF refl] if-cong[OF refl] simp add:
split-def one-time-pad valid-plains-def simp del: map-spmf-of-set-inj-on bind-spmf-const
split: if-split)
  finally have game23: game2 x y = game3 (λ- - - x. x) x y for x y .

  def hash-oracle''' ≡ λ(σ :: 'a ⇒ -). hash.oracle σ
  { def bisim ≡ λσ (s :: 'a set, σ' :: 'a ⇀ bitstring). s = dom σ ∧ σ = σ'
   have [transfer-rule]: bisim Map-empty ({}, Map-empty) by(simp add: bisim-def)
   have [transfer-rule]: (bisim ===> op = ===> rel-spmf (rel-prod op = bisim))
hash-oracle''' hash-oracle''
      by(auto simp add: hash-oracle''-def split-def hash-oracle'''-def spmf-rel-map
hash.oracle-def rel-fun-def bisim-def split: option.split intro!: rel-spmf-bind-reflI)
    have * [transfer-rule]: (bisim ===> op =) dom fst by(simp add: bisim-def
rel-fun-def)
   have * [transfer-rule]: (bisim ===> op =) (λx. x) snd by(simp add: rel-fun-def
bisim-def)
    have game3 (λ- - - x. x) x y = do {
       b ← coin-spmf;
      (((msg1, msg2), σ), s) ← exec-gpv hash-oracle''' (𝒜1 (g (^) x)) hash.initial;
       TRY do {
         - :: unit ← assert-spmf (valid-plains msg1 msg2);
         h' ← spmf-of-set (nlists UNIV len-plain);
         let cipher = (g (^) y, h');
         (guess, s') ← exec-gpv hash-oracle''' (𝒜2 cipher σ) s;
         return-spmf (guess = b, g (^) (x * y) ∈ dom s')
       } ELSE do {
         b ← coin-spmf;
         return-spmf (b, g (^) (x * y) ∈ dom s)
       }
     } for x y
      unfolding game3-def Map-empty-def[symmetric] split-def fst-conv snd-conv
prod.collapse
     by(transfer fixing: 𝒜1 𝒢 len-plain x y 𝒜2) simp
    moreover have map-spmf snd (... x y) = do {
       zs ← elgamal-adversary 𝒜 (g (^) x) (g (^) y);
       return-spmf (g (^) (x * y) ∈ zs)
     } for x y
     by(simp add: o-def split-def hash-oracle'''-def map-try-spmf map-scale-spmf)
     (simp add: o-def map-try-spmf map-scale-spmf map-spmf-conv-bind-spmf[symmetric]
spmf.map-comp map-const-spmf-of-set)
    ultimately have map-spmf snd (?sample (game3 (λ- - - x. x))) = lcdh.lcdh
(elgamal-adversary 𝒜)
```

by(*simp add*: *o-def lcdh.lcdh-def Let-def nat-pow-pow*) **}**
then have *game2-snd*: *map-spmf snd* (*?sample game2*) = *lcdh.lcdh* (*elgamal-adversary* $\mathcal{A}$)
using *game23* **by**(*simp add*: *o-def*)

have *map-spmf fst* (*game3* (λ- - - *x*. *x*) *x y*) = *do* {
(((*msg1*, *msg2*), σ), (*s*, *s-h*)) $\leftarrow$ *exec-gpv hash-oracle''* ($\mathcal{A}$*1* (**g** (^) *x*)) ({}, *hash.initial*);
TRY do {
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains msg1 msg2*);
h' $\leftarrow$ *spmf-of-set* (*nlists UNIV len-plain*);
(*guess*, (*s'*, *s-h'*)) $\leftarrow$ *exec-gpv hash-oracle''* ($\mathcal{A}$*2* (**g** (^) *y*, *h'*) σ) (*s*, *s-h*);
map-spmf (*op* = *guess*) *coin-spmf*
} *ELSE coin-spmf*
} **for** *x y*
including *monad-normalisation*
by(*simp add*: *game3-def o-def split-def map-spmf-conv-bind-spmf try-spmf-bind-out weight-spmf-le-1 scale-bind-spmf try-spmf-bind-out1 bind-scale-spmf*)
then have *game3-fst*: *map-spmf fst* (*game3* (λ- - - *x*. *x*) *x y*) = *coin-spmf* **for** *x y*
by(*simp add*: *o-def if-distribs spmf.map-comp map-eq-const-coin-spmf split-def*)

have *ind-cpa.advantage hash.oracle hash.initial* $\mathcal{A}$ = |*spmf* (*map-spmf fst* (*?sample game1*)) *True* − *1* / *2*|
using *game0* **by**(*simp add*: *ind-cpa-pk.advantage-def game01 o-def*)
also have ... = |*1* / *2* − *spmf* (*map-spmf fst* (*?sample game1*)) *True*|
by(*simp add*: *abs-minus-commute*)
also have *1* / *2* = *spmf* (*map-spmf fst* (*?sample game2*)) *True*
by(*simp add*: *game23 o-def game3-fst spmf-of-set*)
also note *hop23* **also note** *game2-snd*
finally show *?thesis* **by**(*simp add*: *lcdh.advantage-def*)
qed

end

context *elgamal-base* **begin**

lemma *lossless-key-gen* [*simp*]: *lossless-spmf key-gen* $\longleftrightarrow$ *0* < *order* $\mathcal{G}$
by(*simp add*: *key-gen-def Let-def*)

lemma *lossless-elgamal-adversary*:
$[\![$ *ind-cpa.lossless* $\mathcal{A}$; $\bigwedge\eta$. *0* < *order* $\mathcal{G}$ $]\!]$
$\Longrightarrow$ *lcdh.lossless* (*elgamal-adversary* $\mathcal{A}$)
by(*cases* $\mathcal{A}$)(*auto simp add*: *lcdh.lossless-def ind-cpa.lossless-def split-def Let-def intro*!: *lossless-exec-gpv*[**where** $\mathcal{I}$=$\mathcal{I}$-*full*] *lossless-inline*)

end

end

2.3 The random-permutation random-function switching lemma

theory *RP-RF* **imports**
Pseudo-Random-Function
Pseudo-Random-Permutation
CryptHOL.GPV-Bisim
begin

lemma *rp-resample*:
assumes $B \subseteq A \cup C$ $A \cap C = \{\}$ $C \subseteq B$ **and** *finB*: *finite B*
shows *bind-spmf* (*spmf-of-set B*) (λ*x*. *if x* ∈ *A then spmf-of-set C else return-spmf x*) = *spmf-of-set C*
proof(*cases* $C = \{\} \vee A \cap B = \{\}$)
case *False*
define A' **where** $A' \equiv A \cap B$
from *False* **have** *C*: $C \neq \{\}$ **and** A': $A' \neq \{\}$ **by**(*auto simp add*: A'*-def*)
have *B*: $B = A' \cup C$ **using** *assms* **by**(*auto simp add*: A'*-def*)
with *finB* **have** *finA*: *finite* A' **and** *finC*: *finite C* **by** *simp-all*
from *assms* **have** $A'C$: $A' \cap C = \{\}$ **by**(*auto simp add*: A'*-def*)
have *bind-spmf* (*spmf-of-set B*) (λ*x*. *if x* ∈ *A then spmf-of-set C else return-spmf x*) =
bind-spmf (*spmf-of-set B*) (λ*x*. *if x* ∈ A' *then spmf-of-set C else return-spmf x*)
by(*rule bind-spmf-cong*[*OF refl*])(*simp add*: *set-spmf-of-set finB* A'*-def*)
also have ... = *spmf-of-set C* (**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *i*
have ($\sum x \in C$. *spmf* (*if x* ∈ A' *then spmf-of-set C else return-spmf x*) *i*) = *indicator C i* **using** *finA finC*
by(*simp add*: *disjoint-notin1*[*OF* $A'C$] *indicator-single-Some sum-mult-indicator*[*of C* λ-. *1* :: *real* λ-. - λ*x*. *x*, *simplified*] *split*: *split-indicator cong*: *conj-cong sum.cong*)
then show *spmf ?lhs i* = *spmf ?rhs i* **using** *B finA finC* $A'C$ *C* A'
by(*simp add*: *spmf-bind integral-spmf-of-set sum-Un spmf-of-set field-simps*)(*simp add*: *field-simps card-Un-disjoint*)
qed
finally show *?thesis* **.**
qed(*use assms* **in** ‹*auto 4 3 cong*: *bind-spmf-cong-simp simp add*: *subsetD bind-spmf-const spmf-of-set-empty disjoint-notin1 intro*!: *arg-cong*[*where f*=*spmf-of-set*]›)

locale *rp-rf* =
rp: *random-permutation A* +
rf: *random-function spmf-of-set A*
for *A* :: *'a set*
\+
assumes *finite-A*: *finite A*
and *nonempty-A*: $A \neq \{\}$
begin

type-synonym *'a' adversary* = (*bool*, *'a'*, *'a'*) *gpv*

definition *game* :: *bool* ⇒ *'a adversary* ⇒ *bool spmf* **where**
game b $\mathcal{A}$ = *run-gpv* (*if b then rp.random-permutation else rf.random-oracle*) $\mathcal{A}$ *Map.empty*

abbreviation *prp-game* :: *'a adversary* ⇒ *bool spmf* **where** *prp-game* ≡ *game True*
abbreviation *prf-game* :: *'a adversary* ⇒ *bool spmf* **where** *prf-game* ≡ *game False*

definition *advantage* :: *'a adversary* ⇒ *real* **where**
advantage $\mathcal{A}$ = |*spmf* (*prp-game* $\mathcal{A}$) *True* − *spmf* (*prf-game* $\mathcal{A}$) *True*|

lemma *advantage-nonneg*: *0* ≤ *advantage* $\mathcal{A}$ **by**(*simp add*: *advantage-def*)

lemma *advantage-le-1*: *advantage* $\mathcal{A}$ ≤ *1*
by(*auto simp add*: *advantage-def intro*!: *abs-leI*)(*metis diff-0-right diff-left-mono order-trans pmf-le-1 pmf-nonneg*) +

context includes $\mathcal{I}$*.lifting* **begin**
lift-definition $\mathcal{I}$ :: (*'a*, *'a*) $\mathcal{I}$ **is** (λ*x*. *if x* ∈ *A then A else* {}) **.**
lemma *outs-*$\mathcal{I}$*-*$\mathcal{I}$ [*simp*]: *outs-*$\mathcal{I}$ $\mathcal{I}$ = *A* **by** *transfer auto*
lemma *responses-*$\mathcal{I}$*-*$\mathcal{I}$ [*simp*]: *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* = (*if x* ∈ *A then A else* {}) **by** *transfer simp*
lifting-update $\mathcal{I}$*.lifting*
lifting-forget $\mathcal{I}$*.lifting*
end

lemma *rp-rf*:
assumes *bound*: *interaction-any-bounded-by* $\mathcal{A}$ *q*
and *lossless*: *lossless-gpv* $\mathcal{I}$ $\mathcal{A}$
and *WT*: $\mathcal{I}$ ⊢*g* $\mathcal{A}$ √
shows *advantage* $\mathcal{A}$ ≤ *q* ∗ *q* / *card A*
including *lifting-syntax*
proof −
let *?run* = λ*b*. *exec-gpv* (*if b then rp.random-permutation else rf.random-oracle*) $\mathcal{A}$ *Map.empty*
define *rp-bad* :: *bool* × (*'a* ⇀ *'a*) ⇒ *'a* ⇒ (*'a* × (*bool* × (*'a* ⇀ *'a*))) *spmf*
where *rp-bad* = (λ(*bad*, σ) *x*. *case* σ *x of Some y* ⇒ *return-spmf* (*y*, (*bad*, σ))
| *None* ⇒ *bind-spmf* (*spmf-of-set A*) (λ*y*. *if y* ∈ *ran* σ *then map-spmf* (λ*y'*. (*y'*, (*True*, σ(*x* ↦ *y'*)))) (*spmf-of-set* (*A* − *ran* σ)) *else return-spmf* (*y*, (*bad*, (σ(*x* ↦ *y*))))))
have *rp-bad-simps*: *rp-bad* (*bad*, σ) *x* = (*case* σ *x of Some y* ⇒ *return-spmf* (*y*, (*bad*, σ))
| *None* ⇒ *bind-spmf* (*spmf-of-set A*) (λ*y*. *if y* ∈ *ran* σ *then map-spmf* (λ*y'*. (*y'*, (*True*, σ(*x* ↦ *y'*)))) (*spmf-of-set* (*A* − *ran* σ)) *else return-spmf* (*y*, (*bad*, (σ(*x* ↦ *y*))))))
for *bad* σ *x* **by**(*simp add*: *rp-bad-def*)

let *?S* = *rel-prod2 op* =

define *init* :: *bool* × ($'a \rightharpoonup 'a$) **where** *init* = (*False*, *Map.empty*)
have *rp*: *rp.random-permutation* = ($\lambda\sigma$ *x*. *case* σ *x* *of Some y* ⇒ *return-spmf* (*y*, σ)
| *None* ⇒ *bind-spmf* (*bind-spmf* (*spmf-of-set A*) (λ*y*. *if y* ∈ *ran* σ *then spmf-of-set* (*A* − *ran* σ) *else return-spmf y*)) (λ*y*. *return-spmf* (*y*, σ(*x* ↦ *y*))))
by(*subst rp-resample*)(*auto simp add*: *finite-A rp.random-permutation-def*[*abs-def*])
have [*transfer-rule*]: (*?S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *?S*)) *rp.random-permutation rp-bad*
unfolding *rp rp-bad-def*
by(*auto simp add*: *rel-fun-def map-spmf-conv-bind-spmf split*: *option.split intro*!: *rel-spmf-bind-reflI*)
have [*transfer-rule*]: *?S Map.empty init* **by**(*simp add*: *init-def*)
have *spmf* (*prp-game* $\mathcal{A}$) *True* = *spmf* (*run-gpv rp-bad* $\mathcal{A}$ *init*) *True*
unfolding *vimage-def game-def if-True* **by** *transfer-prover*
moreover {
define *collision* :: ($'a \rightharpoonup 'a$) ⇒ *bool* **where** *collision m* ⟷ ¬ *inj-on m* (*dom m*) **for** *m*
have [*simp*]: ¬ *collision Map.empty* **by**(*simp add*: *collision-def*)
have [*simp*]: ⟦ *collision m*; *m x* = *None* ⟧ ⟹ *collision* (*m*(*x* := *y*)) **for** *m x y*
by(*auto simp add*: *collision-def fun-upd-idem dom-minus fun-upd-image dest*: *inj-on-fun-updD*)
have *collision-map-updI*: ⟦ *m x* = *None*; *y* ∈ *ran m* ⟧ ⟹ *collision* (*m*(*x* ↦ *y*)) **for** *m x y*
by(*auto simp add*: *collision-def ran-def intro*: *rev-image-eqI*)
have *collision-map-upd-iff*: ¬ *collision m* ⟹ *collision* (*m*(*x* ↦ *y*)) ⟷ *y* ∈ *ran m* ∧ *m x* ≠ *Some y* **for** *m x y*
by(*auto simp add*: *collision-def ran-def fun-upd-idem intro*: *inj-on-fun-updI rev-image-eqI dest*: *inj-on-eq-iff*)

let *?bad1* = *collision* **and** *?bad2* = *fst*
and *?X* = $\lambda\sigma 1$ (*bad*, $\sigma 2$). $\sigma 1$ = $\sigma 2$ ∧ ¬ *collision* $\sigma 1$ ∧ ¬ *bad*
and *?I1* = $\lambda\sigma 1$. *dom* $\sigma 1$ ⊆ *A* ∧ *ran* $\sigma 1$ ⊆ *A*
and *?I2* = λ(*bad*, $\sigma 2$). *dom* $\sigma 2$ ⊆ *A* ∧ *ran* $\sigma 2$ ⊆ *A*
let *?X-bad* = $\lambda\sigma 1$ *s2*. *?I1* $\sigma 1$ ∧ *?I2 s2*
have [*simp*]: $\mathcal{I}$ ⊢*c rf.random-oracle s1* √ **if** *ran s1* ⊆ *A* **for** *s1* **using** *that*
by(*intro WT-calleeI*)(*auto simp add*: *rf.random-oracle-def*[*abs-def*] *finite-A nonempty-A ran-def split*: *option.split-asm*)
have [*simp*]: *callee-invariant-on rf.random-oracle ?I1* $\mathcal{I}$
by(*unfold-locales*)(*auto simp add*: *rf.random-oracle-def finite-A split*: *option.split-asm*)
then interpret *rf*: *callee-invariant-on rf.random-oracle ?I1* $\mathcal{I}$ **.**
have [*simp*]: $\mathcal{I}$ ⊢*c rp-bad s2* √ **if** *ran* (*snd s2*) ⊆ *A* **for** *s2* **using** *that*
by(*intro WT-calleeI*)(*auto simp add*: *rp-bad-def finite-A split*: *prod.split-asm option.split-asm if-split-asm intro*: *ranI*)
have [*simp*]: *callee-invariant-on rf.random-oracle* ($\lambda\sigma 1$. *?bad1* $\sigma 1$ ∧ *?I1* $\sigma 1$) $\mathcal{I}$
by(*unfold-locales*)(*clarsimp simp add*: *rf.random-oracle-def finite-A split*: *option.split-asm*)+
have [*simp*]: *callee-invariant-on rp-bad* (λ*s2*. *?I2 s2*) $\mathcal{I}$

by(*unfold-locales*)(*auto 4 3 simp add*: *rp-bad-simps finite-A split*: *option.splits if-split-asm iff del*: *domIff*)
have [*simp*]: *callee-invariant-on rp-bad* ($\lambda s2$. *?bad2 s2* $\wedge$ *?I2 s2*) $\mathcal{I}$
by(*unfold-locales*)(*auto 4 3 simp add*: *rp-bad-simps finite-A split*: *option.splits if-split-asm iff del*: *domIff*)
have [*simp*]: $\mathcal{I}$ $\vdash$*c rp-bad* (*bad*, σ*2*) $\surd$ **if** *ran* σ*2* $\subseteq$ *A* **for** *bad* σ*2* **using** *that*
by(*intro WT-calleeI*)(*auto simp add*: *rp-bad-def finite-A nonempty-A ran-def split*: *option.split-asm if-split-asm*)
have [*simp*]: *lossless-spmf* (*rp-bad* (*b*, σ*2*) *x*) **if** *x* $\in$ *A dom* σ*2* $\subseteq$ *A ran* σ*2* $\subseteq$ *A* **for** *b* σ*2 x*
using *finite-A that* **unfolding** *rp-bad-def*
by(*clarsimp simp add*: *nonempty-A dom-subset-ran-iff eq-None-iff-not-dom split*: *option.split*)
have *rel-spmf* (λ(*b1*, σ*1*) (*b2*, *state2*). (*?bad1* σ*1* $\longleftrightarrow$ *?bad2 state2*) $\wedge$ (*if ?bad2 state2 then ?X-bad* σ*1 state2 else b1* = *b2* $\wedge$ *?X* σ*1 state2*))
((*if False then rp.random-permutation else rf.random-oracle*) *s1 x*) (*rp-bad s2 x*)
if *?X s1 s2 x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ *?I1 s1 ?I2 s2* **for** *s1 s2 x* **using** *that finite-A*
by(*auto split*!: *option.split simp add*: *rf.random-oracle-def rp-bad-def rel-spmf-return-spmf1 collision-map-updI dom-subset-ran-iff eq-None-iff-not-dom collision-map-upd-iff intro*!: *rel-spmf-bind-reflI*)
with - - **have** *rel-spmf*
(λ(*b1*, σ*1*) (*b2*, *state2*). (*?bad1* σ*1* $\longleftrightarrow$ *?bad2 state2*) $\wedge$ (*if ?bad2 state2 then ?X-bad* σ*1 state2 else b1* = *b2* $\wedge$ *?X* σ*1 state2*))
(*?run False*) (*exec-gpv rp-bad* $\mathcal{A}$ *init*)
by(*rule exec-gpv-oracle-bisim-bad-invariant*[**where** $\mathcal{I}$ = $\mathcal{I}$ **and** *?I1.0* = *?I1* **and** *?I2.0*=*?I2*])(*auto simp add*: *init-def WT lossless finite-A nonempty-A*)
then have |*spmf* (*map-spmf fst* (*?run False*)) *True* $-$ *spmf* (*run-gpv rp-bad* $\mathcal{A}$ *init*) *True*| $\leq$ *spmf* (*map-spmf* (*?bad1* $\circ$ *snd*) (*?run False*)) *True*
unfolding *spmf-conv-measure-spmf measure-map-spmf vimage-def*
by(*intro fundamental-lemma*[**where** *?bad2.0*=λ(-, *s2*). *?bad2 s2*])(*auto simp add*: *split-def elim*: *rel-spmf-mono*)
also have *ennreal* ... $\leq$ *ennreal* (*q* / *card A*) $*$ (*enat q*) **unfolding** *if-False* **using** *bound* - - - - - - - *WT*
by(*rule rf.interaction-bounded-by-exec-gpv-bad-count*[**where** *count*=λ*s*. *card* (*dom s*)])
(*auto simp add*: *rf.random-oracle-def finite-A nonempty-A card-insert-if finite-subset*[*OF - finite-A*] *map-spmf-conv-bind-spmf*[*symmetric*] *spmf.map-comp o-def collision-map-upd-iff map-mem-spmf-of-set card-gt-0-iff card-mono field-simps Int-absorb2 intro*: *card-ran-le-dom*[*OF finite-subset*, *OF - finite-A*, *THEN order-trans*] *split*: *option.splits*)
hence *spmf* (*map-spmf* (*?bad1* $\circ$ *snd*) (*?run False*)) *True* $\leq$ *q* $*$ *q* / *card A*
by(*simp add*: *ennreal-of-nat-eq-real-of-nat ennreal-times-divide ennreal-mult*$''$[*symmetric*])
finally have |*spmf* (*run-gpv rp-bad* $\mathcal{A}$ *init*) *True* $-$ *spmf* (*run-gpv rf.random-oracle* $\mathcal{A}$ *Map.empty*) *True*| $\leq$ *q* $*$ *q* / *card A*
by *simp* **}**
ultimately show *?thesis* **by**(*simp add*: *advantage-def game-def*)
qed

end

end

2.4 Extending the input length of a PRF using a universal hash function

This example is taken from [4, §4.2].

theory *PRF-UHF* **imports**
  CryptHOL.GPV-Bisim
  Pseudo-Random-Function
begin

locale *hash* =
  fixes *seed-gen* :: *'seed spmf*
  and *hash* :: *'seed* ⇒ *'domain* ⇒ *'range*
begin

definition *game-hash* :: *'domain* ⇒ *'domain* ⇒ *bool spmf*
where
 game-hash w w' = *do* {
   seed ← *seed-gen*;
   return-spmf (*hash seed w* = *hash seed w'* ∧ *w* ≠ *w'*)
  }

definition *game-hash-set* :: *'domain set* ⇒ *bool spmf*
where
  game-hash-set W = *do* {
    seed ← *seed-gen*;
    return-spmf (¬ *inj-on* (*hash seed*) *W*)
   }

definition *ε-uh* :: *real*
where *ε-uh* = (*SUP w w'*. *spmf* (*game-hash w w'*) *True*)

lemma *ε-uh-nonneg* : *ε-uh* ≥ *0*
by(*auto 4 3 intro*!: *cSUP-upper2 bdd-aboveI2*[**where** *M=1*] *cSUP-least pmf-le-1 pmf-nonneg simp add*: *ε-uh-def*)

lemma *hash-ineq-card*:
  assumes *finite W*
  shows *spmf* (*game-hash-set W*) *True* ≤ *ε-uh* ∗ *card W* ∗ *card W*
proof −
  let *?M* = *measure* (*measure-spmf seed-gen*)
  have *bound*: *?M* {*x*. *hash x w* = *hash x w'* ∧ *w* ≠ *w'*} ≤ *ε-uh* **for** *w w'*
  proof −
   have *?M* {*x*. *hash x w* = *hash x w'* ∧ *w* ≠ *w'*} = *spmf* (*game-hash w w'*) *True*
    by(*simp add*: *game-hash-def spmf-conv-measure-spmf map-spmf-conv-bind-spmf*[*symmetric*] *measure-map-spmf vimage-def*)

also have ... $\leq$ *ε-uh* **unfolding** *ε-uh-def*
by(*auto intro*!: *cSUP-upper2 bdd-aboveI*[**where** *M=1*] *cSUP-least simp add*: *pmf-le-1*)
finally show *?thesis* **.**
qed

have *spmf* (*game-hash-set W*) *True* = *?M* $\{x.\ \exists xa \in W.\ \exists y \in W.\ hash\ x\ xa = hash\ x\ y \wedge xa \neq y\}$
by(*auto simp add*: *game-hash-set-def inj-on-def map-spmf-conv-bind-spmf*[*symmetric*] *spmf-conv-measure-spmf measure-map-spmf vimage-def*)
also have $\{x.\ \exists xa \in W.\ \exists y \in W.\ hash\ x\ xa = hash\ x\ y \wedge xa \neq y\} = (\bigcup (w,\ w') \in W \times W.\ \{x.\ hash\ x\ w = hash\ x\ w' \wedge w \neq w'\})$
by(*auto*)
also have *?M* ... $\leq (\sum (w,\ w') \in W \times W.$ *?M* $\{x.\ hash\ x\ w = hash\ x\ w' \wedge w \neq w'\})$
by(*auto intro*!: *measure-spmf.finite-measure-subadditive-finite simp add*: *split-def assms*)
also have ... $\leq (\sum (w,\ w') \in W \times W.$ *ε-uh*) **by**(*rule sum-mono*)(*clarsimp simp add*: *bound*)
also have ... = *ε-uh* $*$ *card*(*W*) $*$ *card*(*W*) **by**(*simp add*: *card-cartesian-product*)
finally show *?thesis* **.**
qed

end

locale *prf-hash* =
fixes *f* :: *'key* ⇒ *'α* ⇒ *'γ*
and *h* :: *'seed* ⇒ *'β* ⇒ *'α*
and *key-gen* :: *'key spmf*
and *seed-gen* :: *'seed spmf*
and *range-f* :: *'γ set*
assumes *lossless-seed-gen*: *lossless-spmf seed-gen*
and *range-f-finite*: *finite range-f*
and *range-f-nonempty*: *range-f* ≠ {}
begin

definition *rand* :: *'γ spmf*
where *rand* = *spmf-of-set range-f*

lemma *lossless-rand* [*simp*]: *lossless-spmf rand*
by(*simp add*: *rand-def range-f-finite range-f-nonempty*)

definition *key-seed-gen* :: (*'key* $*$ *'seed*) *spmf*
where
key-seed-gen = *do* {
k ← *key-gen*;
s :: *'seed* ← *seed-gen*;
return-spmf (*k*, *s*)
}

```
interpretation prf: prf key-gen f rand .
interpretation hash: hash seed-gen h.

fun f' :: 'key × 'seed ⇒ 'β ⇒ 'γ
where f' (key, seed) x = f key (h seed x)

interpretation prf': prf key-seed-gen f' rand .

definition reduction-oracle :: 'seed ⇒ unit ⇒ 'β ⇒ ('γ × unit, 'α, 'γ) gpv
where reduction-oracle seed x b = Pause (h seed b) (λx. Done (x, ()))

definition prf'-reduction :: ('β, 'γ) prf'.adversary ⇒ ('α, 'γ) prf.adversary
where
  prf'-reduction 𝒜 = do {
      seed ← lift-spmf seed-gen;
      (b, σ) ← inline (reduction-oracle seed) 𝒜 ();
      Done b
  }

theorem prf-prf'-advantage:
  assumes prf'.lossless 𝒜
  and bounded: prf'.ibounded-by 𝒜 q
  shows prf'.advantage 𝒜 ≤ prf.advantage (prf'-reduction 𝒜) + hash.ε-uh ∗ q ∗
q
  including lifting-syntax
proof −
  let ?𝒜 = prf'-reduction 𝒜

  { def cr ≡ λ- :: unit × unit. λ- :: unit. True
    have [transfer-rule]: cr ((), ()) () by(simp add: cr-def)
    have prf.game-0 ?𝒜 = prf'.game-0 𝒜
         unfolding prf'.game-0-def prf.game-0-def prf'-reduction-def unfolding
key-seed-gen-def
     by(simp add: exec-gpv-bind split-def exec-gpv-inline reduction-oracle-def bind-map-spmf
prf.prf-oracle-def prf'.prf-oracle-def[abs-def])
        (transfer-prover) }
  note hop1 = this[symmetric]

  def semi-forgetful-RO ≡ λseed :: 'seed. λ(σ :: 'α ⇀ 'β × 'γ, b :: bool). λx.
    case σ (h seed x) of Some (a, y) ⇒ return-spmf (y, (σ, a ≠ x ∨ b))
    | None ⇒ bind-spmf rand (λy. return-spmf (y, (σ(h seed x ↦ (x, y)), b)))

  def game-semi-forgetful ≡ do {
    seed :: 'seed ← seed-gen;
    (b, rep) ← exec-gpv (semi-forgetful-RO seed) 𝒜 (Map.empty, False);
    return-spmf (b, rep)
   }
```

have *bad-semi-forgetful* [*simp*]: *callee-invariant* (*semi-forgetful-RO seed*) *snd* **for** *seed*
by(*unfold-locales*)(*auto simp add*: *semi-forgetful-RO-def split*: *option.split-asm*)
have *lossless-semi-forgetful* [*simp*]: *lossless-spmf* (*semi-forgetful-RO seed s1 x*) **for** *seed s1 x*
by(*simp add*: *semi-forgetful-RO-def split-def split*: *option.split*)

{ def *cr* ≡ λ(- :: *unit*, σ) (σ′ :: ′α ⇒ (′β × ′γ) *option*, - :: *bool*). σ = *map-option snd* ∘ σ′
def *initial* ≡ (*Map.empty* :: ′α ⇒ (′β × ′γ) *option*, *False*)
have [*transfer-rule*]: *cr* ((), *Map.empty*) *initial* **by**(*simp add*: *cr-def initial-def fun-eq-iff*)
have [*transfer-rule*]: (*op* = ===> *cr* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *cr*))
(λ*y p ya*. *do* {*y* ← *prf.random-oracle* (*snd p*) (*h y ya*); *return-spmf* (*fst y*, (), *snd y*) })
semi-forgetful-RO
by(*auto simp add*: *semi-forgetful-RO-def cr-def prf.random-oracle-def rel-fun-def fun-eq-iff split*: *option.split intro*!: *rel-spmf-bind-reflI*)
have *prf.game-1 ?*𝒜 = *map-spmf fst game-semi-forgetful*
unfolding *prf.game-1-def prf′-reduction-def game-semi-forgetful-def*
by(*simp add*: *exec-gpv-bind exec-gpv-inline split-def bind-map-spmf map-spmf-bind-spmf o-def map-spmf-conv-bind-spmf reduction-oracle-def initial-def* [*symmetric*])
(*transfer-prover*) **}**
note *hop2* = *this*

def *game-semi-forgetful-bad* ≡ *do* {
seed :: ′*seed* ← *seed-gen*;
x ← *exec-gpv* (*semi-forgetful-RO seed*) 𝒜 (*Map.empty*, *False*);
return-spmf (*snd x*)
}
have *game-semi-forgetful-bad* : *map-spmf snd game-semi-forgetful* = *game-semi-forgetful-bad*
unfolding *game-semi-forgetful-bad-def game-semi-forgetful-def*
by(*simp add*: *map-spmf-bind-spmf o-def*)

have *bad-random-oracle-A* [*simp*]: *callee-invariant prf.random-oracle* (λσ. ¬ *inj-on* (*h seed*) (*dom* σ)) **for** *seed*
by *unfold-locales*(*auto simp add*: *prf.random-oracle-def split*: *option.split-asm*)

def *invar* ≡ λ*seed* (σ*1*, *b*) (σ*2* :: ′β ⇒ ′γ *option*). ¬ *b* ∧ *dom* σ*1* = *h seed* ‘ *dom* σ*2* ∧
(∀ *x* ∈ *dom* σ*2*. σ*1* (*h seed x*) = *map-option* (*Pair x*) (σ*2 x*))

have *rel-spmf-oracle-adv*:
rel-spmf (λ(*x*, *s1*) (*y*, *s2*). *snd s1* ≠ *inj-on* (*h seed*) (*dom s2*) ∧ (*inj-on* (*h seed*) (*dom s2*) ⟶ *x* = *y* ∧ *invar seed s1 s2*))
(*exec-gpv* (*semi-forgetful-RO seed*) 𝒜 (*Map.empty*, *False*))
(*exec-gpv prf.random-oracle* 𝒜 *Map.empty*)
if *seed*: *seed* ∈ *set-spmf seed-gen* **for** *seed*

proof −
have *invar-initial* [*simp*]: *invar seed* (*Map.empty*, *False*) *Map.empty* **by**(*simp add*: *invar-def*)
have *invarD-inj*: *inj-on* (*h seed*) (*dom s2*) **if** *invar seed bs1 s2* **for** *bs1 s2*
using *that* **by**(*auto intro*!: *inj-onI simp add*: *invar-def*)(*metis domI domIff option.map-sel prod.inject*)

let *?R* = λ(*a*, *s1*) (*b*, *s2* :: ′β ⇒ ′γ *option*).
snd s1 = (¬ *inj-on* (*h seed*) (*dom s2*)) ∧
(¬ ¬ *inj-on* (*h seed*) (*dom s2*) ⟶ *a* = *b* ∧ *invar seed s1 s2*)

have *step*: *rel-spmf ?R* (*semi-forgetful-RO seed σ1b x*) (*prf.random-oracle s2 x*)
if *X*: *invar seed σ1b s2* **for** *s2 σ1b x*
proof −
obtain *σ1 b* **where** [*simp*]: *σ1b* = (*σ1*, *b*) **by**(*cases σ1b*)
from *X* **have** *not-b*: ¬ *b*
and *dom*: *dom σ1* = *h seed* ‘ *dom s2*
and *eq*: ∀ *x*∈*dom s2*. *σ1* (*h seed x*) = *map-option* (*Pair x*) (*s2 x*)
by(*simp-all add*: *invar-def*)
from *X* **have** *inj*: *inj-on* (*h seed*) (*dom s2*) **by**(*rule invarD-inj*)

have *not-in-image*: *h seed x* ∉ *h seed* ‘ (*dom s2* − {*x*}) **if** *σ1* (*h seed x*) = *None*
proof (*rule notI*)
assume *h seed x* ∈ *h seed* ‘ (*dom s2* − {*x*})
then obtain *y* **where** *y* ∈ *dom s2* **and** *hx-hy*: *h seed x* = *h seed y* **by** (*auto*)
then have *σ1* (*h seed y*) = *None* **using** *that* **by** (*auto*)
then have *h seed y* ∉ *h seed* ‘ *dom s2* **using** *dom* **by** (*auto*)
then have *y* ∉ *dom s2* **by** (*auto*)
then show *False* **using** ‹*y* ∈ *dom s2*› **by**(*auto*)
qed

show *?thesis*
proof(*cases σ1* (*h seed x*))
case *σ1*: *None*
hence *s2*: *s2 x* = *None* **using** *dom* **by**(*auto*)
have *insert* (*h seed x*) (*dom σ1*) = *insert* (*h seed x*) (*h seed* ‘ *dom s2*) **by**(*simp add*: *dom*)
then have *invar-update*: *invar seed* (*σ1*(*h seed x* ↦ (*x*, *bs*)), *False*) (*s2*(*x* ↦ *bs*)) **for** *bs*
using *inj not-b not-in-image σ1 dom*
by(*auto simp add*: *invar-def domIff eq*) (*metis domI domIff imageI*)
with *σ1 s2* **show** *?thesis* **using** *inj not-b not-in-image*
by(*auto simp add*: *semi-forgetful-RO-def prf.random-oracle-def intro*: *rel-spmf-bind-reflI*)
next
case *σ1*: (*Some by*)

show *?thesis*
proof(*cases s2 x*)
case *s2*: (*Some z*)
with *eq σ1* **have** *by* = (*x*, *z*) **by**(*auto simp add*: *domIff*)
thus *?thesis* **using** *σ1 inj not-b s2 X*
by(*simp add*: *semi-forgetful-RO-def prf.random-oracle-def split-beta*)
next
case *s2*: *None*
from *σ1 dom* **obtain** *y* **where** *y*: *y* ∈ *dom s2* **and** ∗: *h seed x* = *h seed y*
by(*metis domIff imageE option.distinct*(*1*))
from *y* **obtain** *z* **where** *z*: *s2 y* = *Some z* **by** *auto*
from *eq z σ1* **have** *by*: *by* = (*y*, *z*) **by**(*auto simp add*: ∗ *domIff*)
from *y s2* **have** *xny*: *x* ≠ *y* **by** *auto*
with *y* ∗ **have** *h seed x* ∈ *h seed* ` (*dom s2* − {*x*}) **by** *auto*
then show *?thesis* **using** *σ1 s2 not-b by xny inj*
by(*simp add*: *semi-forgetful-RO-def prf.random-oracle-def split-beta*)(*rule rel-spmf-bindI2*; *simp*)
qed
qed
qed
from *invar-initial - step* **show** *?thesis*
by(*rule exec-gpv-oracle-bisim-bad-full*[**where** *?bad1.0* = *snd* **and** *?bad2.0* = *λσ.* ¬ *inj-on* (*h seed*) (*dom σ*)])
(*simp-all add*: *assms*)
qed

def *game-A* ≡ *do* {
seed :: *'seed* ← *seed-gen*;
(*b*, *σ*) ← *exec-gpv prf.random-oracle 𝒜 Map.empty*;
return-spmf (*b*, ¬ *inj-on* (*h seed*) (*dom σ*))
}

let *?bad1* = *λx. snd* (*snd x*) **and** *?bad2* = *snd*
have *hop3*: *rel-spmf* (*λx xa.* (*?bad1 x* ⟷ *?bad2 xa*) ∧ (¬ *?bad2 xa* ⟶ *fst x* ⟷ *fst xa*)) *game-semi-forgetful game-A*
unfolding *game-semi-forgetful-def game-A-def*
by(*clarsimp simp add*: *restrict-bind-spmf split-def map-spmf-bind-spmf restrict-return-spmf o-def intro*!: *rel-spmf-bind-reflI simp del*: *bind-return-spmf*)
(*rule rel-spmf-bindI*[*OF rel-spmf-oracle-adv*]; *auto*)
have *bad1-bad2*: *spmf* (*map-spmf* (*snd* ∘ *snd*) *game-semi-forgetful*) *True* = *spmf* (*map-spmf snd game-A*) *True*
using *fundamental-lemma-bad*[*OF hop3*] **by**(*simp add*: *measure-map-spmf spmf-conv-measure-spmf vimage-def*)
have *bound-bad1-event*: |*spmf* (*map-spmf fst game-semi-forgetful*) *True* − *spmf* (*map-spmf fst game-A*) *True*| ≤ *spmf* (*map-spmf* (*snd* ∘ *snd*) *game-semi-forgetful*) *True*
using *fundamental-lemma*[*OF hop3*] **by**(*simp add*: *measure-map-spmf spmf-conv-measure-spmf vimage-def*)

```
  then have bound-bad2-event : |spmf (map-spmf fst game-semi-forgetful) True −
spmf (map-spmf fst game-A) True| ≤ spmf (map-spmf snd game-A) True
    using bad1-bad2 by (simp)

  def game-B ≡ do {
      (b, σ) ← exec-gpv prf.random-oracle 𝒜 Map.empty;
      hash.game-hash-set (dom σ)
    }

  have game-A-game-B: map-spmf snd game-A = game-B
   unfolding game-B-def game-A-def hash.game-hash-set-def including monad-normalisation
    by(simp add: map-spmf-bind-spmf o-def split-def)

 have game-B-bound : spmf game-B True ≤ hash.ε-uh ∗ q ∗ q unfolding game-B-def
  proof(rule spmf-bind-leI, clarify)
    fix b σ
    assume ∗: (b, σ) ∈ set-spmf (exec-gpv prf.random-oracle 𝒜 Map.empty)
    have finite (dom σ) by(rule prf.finite.exec-gpv-invariant[OF ∗]) simp-all
   then have spmf (hash.game-hash-set (dom σ)) True ≤ hash.ε-uh ∗ (card (dom
σ) ∗ card (dom σ))
      using hash.hash-ineq-card[of dom σ] by simp
   also have p1: card (dom σ) ≤ q + card (dom (Map.empty :: 'β ⇒ 'γ option))

      by(rule prf.card-dom-random-oracle[OF bounded ∗]) simp
   then have card (dom σ) ∗ card (dom σ) ≤ q ∗ q using mult-le-mono by auto
    finally show spmf (hash.game-hash-set (dom σ)) True ≤ hash.ε-uh ∗ q ∗ q
      by(simp add: hash.ε-uh-nonneg mult-left-mono)
  qed(simp add: hash.ε-uh-nonneg)

  have hop4: prf'.game-1 𝒜 = map-spmf fst game-A
     by(simp add: game-A-def prf'.game-1-def map-spmf-bind-spmf o-def split-def
bind-spmf-const lossless-seed-gen lossless-weight-spmfD)

  have prf'.advantage 𝒜 ≤ |spmf (prf.game-0 ?𝒜) True − spmf (prf'.game-1 𝒜)
True|
    using hop1 by(simp add: prf'.advantage-def)
   also have ... ≤ prf.advantage ?𝒜 + |spmf (prf.game-1 ?𝒜) True − spmf
(prf'.game-1 𝒜) True|
    by(simp add: prf.advantage-def)
  also have |spmf (prf.game-1 ?𝒜) True − spmf (prf'.game-1 𝒜) True| ≤
    |spmf (map-spmf fst game-semi-forgetful) True − spmf (prf'.game-1 𝒜) True|
    using hop2 by simp
  also have ... ≤ hash.ε-uh ∗ q ∗ q
    using game-A-game-B game-B-bound bound-bad2-event hop4 by(simp)
  finally show ?thesis by(simp add: add-left-mono)
qed

end
```

end

2.5 IND-CPA from PRF

theory *PRF-IND-CPA* **imports**
CryptHOL.GPV-Bisim
CryptHOL.List-Bits
Pseudo-Random-Function
IND-CPA
begin

Formalises the construction from [3].

declare [[*simproc del*: *let-simp*]]

type-synonym *key* = *bool list*
type-synonym *plain* = *bool list*
type-synonym *cipher* = *bool list* ∗ *bool list*

locale *otp* =
fixes *f* :: *key* ⇒ *bool list* ⇒ *bool list*
and *len* :: *nat*
assumes *length-f*: $\bigwedge$*xs ys*. ⟦ *length xs* = *len*; *length ys* = *len* ⟧ ⟹ *length* (*f xs ys*) = *len*
begin

definition *key-gen* :: *bool list spmf*
where *key-gen* = *spmf-of-set* (*nlists UNIV len*)

definition *valid-plain* :: *plain* ⇒ *bool*
where *valid-plain plain* ⟷ *length plain* = *len*

definition *encrypt* :: *key* ⇒ *plain* ⇒ *cipher spmf*
where
encrypt key plain = *do* {
r ← *spmf-of-set* (*nlists UNIV len*);
return-spmf (*r*, *xor-list plain* (*f key r*))
}

fun *decrypt* :: *key* ⇒ *cipher* ⇒ *plain option*
where *decrypt key* (*r*, *c*) = *Some* (*xor-list* (*f key r*) *c*)

lemma *encrypt-decrypt-correct*:
⟦ *length key* = *len*; *length plain* = *len* ⟧
⟹ *encrypt key plain* ⤜ (λ*cipher*. *return-spmf* (*decrypt key cipher*)) = *return-spmf* (*Some plain*)
by(*simp add*: *encrypt-def zip-map2 o-def split-def bind-eq-return-spmf length-f in-nlists-UNIV xor-list-left-commute*)

interpretation *ind-cpa*: *ind-cpa key-gen encrypt decrypt valid-plain* **.**

interpretation *prf*: *prf key-gen f spmf-of-set* (*nlists UNIV len*) **.**

definition *prf-encrypt-oracle* :: *unit* ⇒ *plain* ⇒ (*cipher* × *unit*, *plain*, *plain*) *gpv*
where
  prf-encrypt-oracle x plain = *do* {
    r ← *lift-spmf* (*spmf-of-set* (*nlists UNIV len*));
    Pause r (λ*pad*. *Done* ((*r*, *xor-list plain pad*), ()))
  }

lemma *interaction-bounded-by-prf-encrypt-oracle* [*interaction-bound*]:
  interaction-any-bounded-by (*prf-encrypt-oracle* σ *plain*) *1*
unfolding *prf-encrypt-oracle-def* **by** *simp*

lemma *lossless-prf-encyrpt-oracle* [*simp*]: *lossless-gpv* *I-top* (*prf-encrypt-oracle s x*)
by(*simp add*: *prf-encrypt-oracle-def*)

definition *prf-adversary* :: (*plain*, *cipher*, ′*state*) *ind-cpa.adversary* ⇒ (*plain*, *plain*) *prf.adversary*
where
  prf-adversary 𝒜 = *do* {
    let (𝒜*1*, 𝒜*2*) = 𝒜;
    (((*p1*, *p2*), σ), *n*) ← *inline prf-encrypt-oracle* 𝒜*1* ();
    if valid-plain p1 ∧ *valid-plain p2 then do* {
      b ← *lift-spmf coin-spmf*;
      let pb = (*if b then p1 else p2*);
      r ← *lift-spmf* (*spmf-of-set* (*nlists UNIV len*));
      pad ← *Pause r Done*;
      let c = (*r*, *xor-list pb pad*);
      (*b*′, -) ← *inline prf-encrypt-oracle* (𝒜*2 c* σ) *n*;
      Done (*b*′ = *b*)
    } *else lift-spmf coin-spmf*
  }

theorem *prf-encrypt-advantage*:
  assumes *ind-cpa.ibounded-by* 𝒜 *q*
  and *lossless-gpv* *I-full* (*fst* 𝒜)
  and ⋀*cipher* σ. *lossless-gpv* *I-full* (*snd* 𝒜 *cipher* σ)
  shows *ind-cpa.advantage* 𝒜 ≤ *prf.advantage* (*prf-adversary* 𝒜) + *q* / *2* ^ *len*
proof −
  note [*split del*] = *if-split*
    and [*cong del*] = *if-weak-cong*
    and [*simp*] =
      bind-spmf-const map-spmf-bind-spmf bind-map-spmf
      exec-gpv-bind exec-gpv-inline
      rel-spmf-bind-reflI rel-spmf-reflI
  obtain 𝒜*1* 𝒜*2* **where** 𝒜: 𝒜 = (𝒜*1*, 𝒜*2*) **by**(*cases* 𝒜)
  from ‹*ind-cpa.ibounded-by* - -›
  obtain *q1 q2* :: *nat*

where *q1*: *interaction-any-bounded-by* $\mathcal{A}1$ *q1*
and *q2*: $\bigwedge$*cipher* σ. *interaction-any-bounded-by* ($\mathcal{A}2$ *cipher* σ) *q2*
and *q1* + *q2* $\leq$ *q*
unfolding $\mathcal{A}$ **by**(*rule ind-cpa.ibounded-byE*)(*auto simp add*: *iadd-le-enat-iff*)
from $\mathcal{A}$ *assms* **have** *lossless1*: *lossless-gpv* $\mathcal{I}$*-full* $\mathcal{A}1$
and *lossless2*: $\bigwedge$*cipher* σ. *lossless-gpv* $\mathcal{I}$*-full* ($\mathcal{A}2$ *cipher* σ) **by** *simp-all*
have *weight1*: $\bigwedge$*oracle s*. ($\bigwedge$*s x*. *lossless-spmf* (*oracle s x*))
$\Longrightarrow$ *weight-spmf* (*exec-gpv oracle* $\mathcal{A}1$ *s*) = *1*
by(*rule lossless-weight-spmfD*)(*rule lossless-exec-gpv*[*OF lossless1*], *simp-all*)
have *weight2*: $\bigwedge$*oracle s cipher* σ. ($\bigwedge$*s x*. *lossless-spmf* (*oracle s x*))
$\Longrightarrow$ *weight-spmf* (*exec-gpv oracle* ($\mathcal{A}2$ *cipher* σ) *s*) = *1*
by(*rule lossless-weight-spmfD*)(*rule lossless-exec-gpv*[*OF lossless2*], *simp-all*)

let *?oracle1* = λ*key* (*s'*, *s*) *y*. *map-spmf* (λ((*x*, *s'*), *s*). (*x*, (), ())) (*exec-gpv* (*prf.prf-oracle key*) (*prf-encrypt-oracle* () *y*) ())
have *bisim1*: $\bigwedge$*key*. *rel-spmf* (λ(*x*, -) (*y*, -). *x* = *y*)
(*exec-gpv* (*ind-cpa.encrypt-oracle key*) $\mathcal{A}1$ ())
(*exec-gpv* (*?oracle1 key*) $\mathcal{A}1$ ((), ()))
using *TrueI*
by(*rule exec-gpv-oracle-bisim*)(*auto simp add*: *encrypt-def prf-encrypt-oracle-def ind-cpa.encrypt-oracle-def prf.prf-oracle-def o-def*)
have *bisim2*: $\bigwedge$*key cipher* σ. *rel-spmf* (λ(*x*, -) (*y*, -). *x* = *y*)
(*exec-gpv* (*ind-cpa.encrypt-oracle key*) ($\mathcal{A}2$ *cipher* σ) ())
(*exec-gpv* (*?oracle1 key*) ($\mathcal{A}2$ *cipher* σ) ((), ()))
using *TrueI*
by(*rule exec-gpv-oracle-bisim*)(*auto simp add*: *encrypt-def prf-encrypt-oracle-def ind-cpa.encrypt-oracle-def prf.prf-oracle-def o-def*)

have *ind-cpa-0*: *rel-spmf op* = (*ind-cpa.ind-cpa* $\mathcal{A}$) (*prf.game-0* (*prf-adversary* $\mathcal{A}$))
unfolding *IND-CPA.ind-cpa.ind-cpa-def* $\mathcal{A}$ *key-gen-def Let-def prf-adversary-def Pseudo-Random-Function.prf.game-0-def*
apply(*simp*)
apply(*rewrite* **in** *bind-spmf* - ⌑ *bind-commute-spmf*)
apply(*rule rel-spmf-bind-reflI*)
apply(*rule rel-spmf-bindI*[*OF bisim1*])
apply(*clarsimp simp add*: *if-distribs bind-coin-spmf-eq-const'*)
apply(*auto intro*: *rel-spmf-bindI*[*OF bisim2*] *intro*!: *rel-spmf-bind-reflI simp add*: *encrypt-def prf.prf-oracle-def cong del*: *if-cong*)
done

def *rf-encrypt* $\equiv$ λ*s plain*. *bind-spmf* (*spmf-of-set* (*nlists UNIV len*)) (λ*r* :: *bool list*.
bind-spmf (*prf.random-oracle s r*) (λ(*pad*, *s'*).
return-spmf ((*r*, *xor-list plain pad*), *s'*))
)
interpret *rf-finite*: *callee-invariant-on rf-encrypt* λ*s*. *finite* (*dom s*) $\mathcal{I}$*-full*
by *unfold-locales*(*auto simp add*: *rf-encrypt-def dest*: *prf.finite.callee-invariant*)
have *lossless-rf-encrypt* [*simp*]: $\bigwedge$*s plain*. *lossless-spmf* (*rf-encrypt s plain*)

by(*auto simp add*: *rf-encrypt-def*)

def *game2* ≡ *do* {
  (((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* 𝒜*1 Map.empty*;
  if valid-plain p0 ∧ *valid-plain p1 then do* {
    b ← *coin-spmf*;
    let pb = (*if b then p0 else p1*);
    (*cipher*, *s2*) ← *rf-encrypt s1 pb*;
    (*b*′, *s3*) ← *exec-gpv rf-encrypt* (𝒜*2 cipher* σ) *s2*;
    return-spmf (*b*′ = *b*)
  } *else coin-spmf*
}

let *?oracle2* = λ(*s*′, *s*) *y*. *map-spmf* (λ((*x*, *s*′), *s*). (*x*, (), *s*)) (*exec-gpv prf.random-oracle* (*prf-encrypt-oracle* () *y*) *s*)
let *?I* = λ(*x*, -, *s*) (*y*, *s*′). *x* = *y* ∧ *s* = *s*′
have *bisim1*: *rel-spmf ?I* (*exec-gpv ?oracle2* 𝒜*1* ((), *Map.empty*)) (*exec-gpv rf-encrypt* 𝒜*1 Map.empty*)
  by(*rule exec-gpv-oracle-bisim*[**where** *X*=λ(-, *s*) *s*′. *s* = *s*′])
  (*auto simp add*: *rf-encrypt-def prf-encrypt-oracle-def intro*!: *rel-spmf-bind-reflI*)
have *bisim2*: ⋀*cipher* σ *s*. *rel-spmf ?I* (*exec-gpv ?oracle2* (𝒜*2 cipher* σ) ((), *s*)) (*exec-gpv rf-encrypt* (𝒜*2 cipher* σ) *s*)
  by(*rule exec-gpv-oracle-bisim*[**where** *X*=λ(-, *s*) *s*′. *s* = *s*′])
  (*auto simp add*: *prf-encrypt-oracle-def rf-encrypt-def intro*!: *rel-spmf-bind-reflI*)
have *game1-2* [*unfolded spmf-rel-eq*]: *rel-spmf op* = (*prf.game-1* (*prf-adversary* 𝒜)) *game2*
  unfolding *prf.game-1-def game2-def prf-adversary-def*
  by(*rewrite* **in** *if* - *then* ⌑ *else* - *rf-encrypt-def*)
  (*auto simp add*: *Let-def* 𝒜 *if-distribs intro*!: *rel-spmf-bindI*[*OF bisim2*] *rel-spmf-bind-reflI* *rel-spmf-bindI*[*OF bisim1*])

def *game2-a* ≡ *do* {
  r ← *spmf-of-set* (*nlists UNIV len*);
  (((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* 𝒜*1 Map.empty*;
  let bad = *r* ∈ *dom s1*;
  if valid-plain p0 ∧ *valid-plain p1 then do* {
    b ← *coin-spmf*;
    let pb = (*if b then p0 else p1*);
    (*pad*, *s2*) ← *prf.random-oracle s1 r*;
    let cipher = (*r*, *xor-list pb pad*);
    (*b*′, *s3*) ← *exec-gpv rf-encrypt* (𝒜*2 cipher* σ) *s2*;
    return-spmf (*b*′ = *b*, *bad*)
  } *else coin-spmf* ⤜ (λ*b*. *return-spmf* (*b*, *bad*))
}
def *game2-b* ≡ *do* {
  r ← *spmf-of-set* (*nlists UNIV len*);
  (((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* 𝒜*1 Map.empty*;
  let bad = *r* ∈ *dom s1*;
  if valid-plain p0 ∧ *valid-plain p1 then do* {

```
    b ← coin-spmf;
    let pb = (if b then p0 else p1);
    pad ← spmf-of-set (nlists UNIV len);
    let cipher = (r, xor-list pb pad);
    (b', s3) ← exec-gpv rf-encrypt (𝒜2 cipher σ) (s1(r ↦ pad));
    return-spmf (b' = b, bad)
  } else coin-spmf ≫= (λb. return-spmf (b, bad))
}

have game2 = do {
    r ← spmf-of-set (nlists UNIV len);
    (((p0, p1), σ), s1) ← exec-gpv rf-encrypt 𝒜1 Map.empty;
    if valid-plain p0 ∧ valid-plain p1 then do {
      b ← coin-spmf;
      let pb = (if b then p0 else p1);
      (pad, s2) ← prf.random-oracle s1 r;
      let cipher = (r, xor-list pb pad);
      (b', s3) ← exec-gpv rf-encrypt (𝒜2 cipher σ) s2;
      return-spmf (b' = b)
    } else coin-spmf
  }
  including monad-normalisation by(simp add: game2-def split-def rf-encrypt-def
Let-def)
also have ... = map-spmf fst game2-a unfolding game2-a-def
  by(clarsimp simp add: map-spmf-conv-bind-spmf Let-def cond-application-beta
if-distrib split-def cong: if-cong)
finally have game2-2a: game2 = ... .

have map-spmf snd game2-a = map-spmf snd game2-b unfolding game2-a-def
game2-b-def
  by(auto simp add: o-def Let-def split-def if-distribs weight2 split: option.split
intro: bind-spmf-cong[OF refl])
moreover
have rel-spmf op = (map-spmf fst (game2-a ↿ (snd -` {False}))) (map-spmf fst
(game2-b ↿ (snd -` {False})))
  unfolding game2-a-def game2-b-def
  by(clarsimp simp add: restrict-bind-spmf o-def Let-def if-distribs split-def restrict-return-spmf
prf.random-oracle-def intro!: rel-spmf-bind-reflI split: option.splits)
hence spmf game2-a (True, False) = spmf game2-b (True, False)
  unfolding spmf-rel-eq by(subst (1 2) spmf-map-restrict[symmetric]) simp
ultimately
have game2a-2b: |spmf (map-spmf fst game2-a) True - spmf (map-spmf fst
game2-b) True| ≤ spmf (map-spmf snd game2-a) True
  by(subst (1 2) spmf-conv-measure-spmf)(rule identical-until-bad; simp add:
spmf.map-id[unfolded id-def] spmf-conv-measure-spmf)

def game2-a-bad ≡ do {
    r ← spmf-of-set (nlists UNIV len);
    (((p0, p1), σ), s1) ← exec-gpv rf-encrypt 𝒜1 Map.empty;
```

return-spmf $(r \in dom\ s1)$
}
have *game2a-bad*: *map-spmf snd game2-a* = *game2-a-bad*
unfolding *game2-a-def game2-a-bad-def*
by(*auto intro*!: *bind-spmf-cong*[*OF refl*] *simp add*: *o-def weight2 Let-def split-def split*: *if-split*)
have *card*: $\bigwedge B$ :: *bool list set*. *card* $(B \cap$ *nlists UNIV len*$) \leq$ *card* (*nlists UNIV len* :: *bool list set*)
by(*rule card-mono*) *simp-all*
then have *spmf game2-a-bad True* = $\int^+$ *x*. *card* (*dom* (*snd x*) $\cap$ *nlists UNIV len*) / 2 ^ *len* ∂*measure-spmf* (*exec-gpv rf-encrypt* $\mathcal{A}1$ *Map.empty*)
unfolding *game2-a-bad-def*
by(*rewrite bind-commute-spmf*)(*simp add*: *ennreal-spmf-bind split-def map-mem-spmf-of-set*[*unfolded map-spmf-conv-bind-spmf*] *card-nlists*)
also { **fix** *x s*
assume $*$: (*x*, *s*) $\in$ *set-spmf* (*exec-gpv rf-encrypt* $\mathcal{A}1$ *Map.empty*)
hence *finite* (*dom s*) **by**(*rule rf-finite.exec-gpv-invariant*) *simp-all*
hence *1*: *card* (*dom s* $\cap$ *nlists UNIV len*) $\leq$ *card* (*dom s*) **by**(*intro card-mono*) *simp-all*
moreover from *q1* $*$
have *card* (*dom s*) $\leq$ *q1* + *card* (*dom* (*Map.empty* :: (*plain*, *plain*) *prf.dict*))
by(*rule rf-finite.interaction-bounded-by'-exec-gpv-count*)
(*auto simp add*: *rf-encrypt-def eSuc-enat prf.random-oracle-def card-insert-if split*: *option.split-asm if-split*)
ultimately have *card* (*dom s* $\cap$ *nlists UNIV len*) $\leq$ *q1* **by**(*simp*) }
then have ... $\leq$ $\int^+$ *x*. *q1* / 2 ^ *len* ∂*measure-spmf* (*exec-gpv rf-encrypt* $\mathcal{A}1$ *Map.empty*)
by(*intro nn-integral-mono-AE*)(*clarsimp simp add*: *field-simps*)
also have ... $\leq$ *q1* / 2 ^ *len*
by(*simp add*: *measure-spmf.emeasure-eq-measure field-simps mult-left-le weight1*)
finally have *game2a-bad-bound*: *spmf game2-a-bad True* $\leq$ *q1* / 2 ^ *len* **by** *simp*

def *rf-encrypt-bad* $\equiv$ λ*secret* (*s* :: (*plain*, *plain*) *prf.dict*, *bad*) *plain*. *bind-spmf*
(*spmf-of-set* (*nlists UNIV len*)) (λ*r*.
bind-spmf (*prf.random-oracle s r*) (λ(*pad*, *s'*).
return-spmf ((*r*, *xor-list plain pad*), (*s'*, *bad* $\vee$ *r* = *secret*))))
have *rf-encrypt-bad-sticky* [*simp*]: $\bigwedge$*s*. *callee-invariant* (*rf-encrypt-bad s*) *snd*
by(*unfold-locales*)(*auto simp add*: *rf-encrypt-bad-def*)
have *lossless-rf-encrypt* [*simp*]: $\bigwedge$*challenge s plain*. *lossless-spmf* (*rf-encrypt-bad challenge s plain*)
by(*clarsimp simp add*: *rf-encrypt-bad-def prf.random-oracle-def split*: *option.split*)

def *game2-c* $\equiv$ *do* {
r $\leftarrow$ *spmf-of-set* (*nlists UNIV len*);
(((*p0*, *p1*), σ), *s1*) $\leftarrow$ *exec-gpv rf-encrypt* $\mathcal{A}1$ *Map.empty*;
if valid-plain p0 $\wedge$ *valid-plain p1 then do* {
b $\leftarrow$ *coin-spmf*;
let pb = (*if b then p0 else p1*);
pad $\leftarrow$ *spmf-of-set* (*nlists UNIV len*);

let cipher = (*r*, *xor-list pb pad*);
(*b'*, (*s2*, *bad*)) ← *exec-gpv* (*rf-encrypt-bad r*) ($\mathcal{A}$*2 cipher* σ) (*s1*(*r* ↦ *pad*), *False*);
return-spmf (*b'* = *b*, *bad*)
} *else coin-spmf* ⋙ (λ*b*. *return-spmf* (*b*, *False*))
}

have *bisim2c-bad*: ⋀*cipher* σ *s x r*. *rel-spmf* (λ(*x*, -) (*y*, -). *x* = *y*)
(*exec-gpv rf-encrypt* ($\mathcal{A}$*2 cipher* σ) (*s*(*x* ↦ *r*)))
(*exec-gpv* (*rf-encrypt-bad x*) ($\mathcal{A}$*2 cipher* σ) (*s*(*x* ↦ *r*), *False*))
by(*rule exec-gpv-oracle-bisim*[**where** *X*=λ*s* (*s'*, -). *s* = *s'*])
(*auto simp add*: *rf-encrypt-bad-def rf-encrypt-def intro*!: *rel-spmf-bind-reflI*)

have *game2b-c* [*unfolded spmf-rel-eq*]: *rel-spmf op* = (*map-spmf fst game2-b*) (*map-spmf fst game2-c*)
by(*auto simp add*: *game2-b-def game2-c-def o-def split-def Let-def if-distribs intro*!: *rel-spmf-bind-reflI rel-spmf-bindI*[*OF bisim2c-bad*])

def *game2-d* ≡ *do* {
r ← *spmf-of-set* (*nlists UNIV len*);
(((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* $\mathcal{A}$*1 Map.empty*;
if valid-plain p0 ∧ *valid-plain p1 then do* {
b ← *coin-spmf*;
let pb = (*if b then p0 else p1*);
pad ← *spmf-of-set* (*nlists UNIV len*);
let cipher = (*r*, *xor-list pb pad*);
(*b'*, (*s2*, *bad*)) ← *exec-gpv* (*rf-encrypt-bad r*) ($\mathcal{A}$*2 cipher* σ) (*s1*, *False*);
return-spmf (*b'* = *b*, *bad*)
} *else coin-spmf* ⋙ (λ*b*. *return-spmf* (*b*, *False*))
}

{ fix *cipher* σ **and** *x* :: *plain* **and** *s r*
let *?I* = (λ(*x*, *s*, *bad*) (*y*, *s'*, *bad'*). (*bad* ⟷ *bad'*) ∧ (¬ *bad'* ⟶ *x* ⟷ *y*))
let *?X* = λ(*s*, *bad*) (*s'*, *bad'*). *bad* = *bad'* ∧ (∀ *z*. *z* ≠ *x* ⟶ *s z* = *s' z*)
have ⋀*s1 s2 x'*. *?X s1 s2* ⟹ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *snd s1'* = *snd s2'* ∧ (¬ *snd s2'* ⟶ *a* = *b* ∧ *?X s1' s2'*))
(*rf-encrypt-bad x s1 x'*) (*rf-encrypt-bad x s2 x'*)
by(*case-tac x* = *x'*)(*clarsimp simp add*: *rf-encrypt-bad-def prf.random-oracle-def rel-spmf-return-spmf1 rel-spmf-return-spmf2 Let-def split-def bind-UNION intro*!: *rel-spmf-bind-reflI split*: *option.split*)+
with - - **have** *rel-spmf ?I*
(*exec-gpv* (*rf-encrypt-bad x*) ($\mathcal{A}$*2 cipher* σ) (*s*(*x* ↦ *r*), *False*))
(*exec-gpv* (*rf-encrypt-bad x*) ($\mathcal{A}$*2 cipher* σ) (*s*, *False*))
by(*rule exec-gpv-oracle-bisim-bad-full*)(*auto simp add*: *lossless2*) **}**
note *bisim-bad* = *this*
have *game2c-2d-bad* [*unfolded spmf-rel-eq*]: *rel-spmf op* = (*map-spmf snd game2-c*) (*map-spmf snd game2-d*)
by(*auto simp add*: *game2-c-def game2-d-def o-def Let-def split-def if-distribs*

intro!: *rel-spmf-bind-reflI rel-spmf-bindI*[*OF bisim-bad*])
moreover
have *rel-spmf op* = (*map-spmf fst* (*game2-c* ↾ (*snd* −‘ {*False*}))) (*map-spmf fst* (*game2-d* ↾ (*snd* −‘ {*False*})))
unfolding *game2-c-def game2-d-def*
by(*clarsimp simp add*: *restrict-bind-spmf o-def Let-def if-distribs split-def restrict-return-spmf intro*!: *rel-spmf-bind-reflI rel-spmf-bindI*[*OF bisim-bad*])
hence *spmf game2-c* (*True*, *False*) = *spmf game2-d* (*True*, *False*)
unfolding *spmf-rel-eq* **by**(*subst* (*1 2*) *spmf-map-restrict*[*symmetric*]) *simp*
ultimately have *game2c-2d*: |*spmf* (*map-spmf fst game2-c*) *True* − *spmf* (*map-spmf fst game2-d*) *True*| ≤ *spmf* (*map-spmf snd game2-c*) *True*
apply(*subst* (*1 2*) *spmf-conv-measure-spmf*)
apply(*intro identical-until-bad*)
apply(*simp-all add*: *spmf.map-id*[*unfolded id-def*] *spmf-conv-measure-spmf*)
done
{ fix *cipher* σ **and** *challenge* :: *plain* **and** *s*
have *card* (*nlists UNIV len* ∩ (λ*x*. *x* = *challenge*) −‘ {*True*}) ≤ *card* {*challenge*}
by(*rule card-mono*) *auto*
then have *spmf* (*map-spmf* (*snd* ∘ *snd*) (*exec-gpv* (*rf-encrypt-bad challenge*) (𝒜*2 cipher* σ) (*s*, *False*))) *True* ≤ (*1* / *2* ˆ *len*) ∗ *q2*
by(*intro oi-True.interaction-bounded-by-exec-gpv-bad*[*OF q2*])(*simp-all add*: *rf-encrypt-bad-def o-def split-beta map-spmf-conv-bind-spmf*[*symmetric*] *spmf-map measure-spmf-of-set field-simps card-nlists*)
hence (∫⁺ *x*. *ennreal* (*indicator* {*True*} *x*) ∂*measure-spmf* (*map-spmf* (*snd* ∘ *snd*) (*exec-gpv* (*rf-encrypt-bad challenge*) (𝒜*2 cipher* σ) (*s*, *False*)))) ≤ (*1* / *2* ˆ *len*) ∗ *q2*
by(*simp only*: *ennreal-indicator nn-integral-indicator sets-measure-spmf sets-count-space Pow-UNIV UNIV-I emeasure-spmf-single*) *simp* **}**
then have *spmf* (*map-spmf snd game2-d*) *True* ≤
∫⁺ (*r* :: *plain*). ∫⁺ (((*p0*, *p1*), σ), *s*). (*if valid-plain p0* ∧ *valid-plain p1* *then*
∫⁺ *b* . ∫⁺ (*pad* :: *plain*). *q2* / *2* ˆ *len* ∂*measure-spmf* (*spmf-of-set* (*nlists UNIV len*)) ∂*measure-spmf coin-spmf*
else 0)
∂*measure-spmf* (*exec-gpv rf-encrypt* 𝒜*1 Map.empty*) ∂*measure-spmf* (*spmf-of-set* (*nlists UNIV len*))
unfolding *game2-d-def*
by(*simp add*: *ennreal-spmf-bind o-def split-def Let-def if-distribs if-distrib*[**where** *f*=λ*x*. *ennreal* (*spmf x* -)] *indicator-single-Some nn-integral-mono if-mono-cong del*: *nn-integral-const cong*: *if-cong*)
also have ... ≤ ∫⁺ (*r* :: *plain*). ∫⁺ (((*p0*, *p1*), σ), *s*). (*if valid-plain p0* ∧ *valid-plain p1 then ennreal* (*q2* / *2* ˆ *len*) *else q2* / *2* ˆ *len*)
∂*measure-spmf* (*exec-gpv rf-encrypt* 𝒜*1 Map.empty*) ∂*measure-spmf* (*spmf-of-set* (*nlists UNIV len*))
unfolding *split-def*
by(*intro nn-integral-mono if-mono-cong*)(*auto simp add*: *measure-spmf.emeasure-eq-measure*)
also have ... ≤ *q2* / *2* ˆ *len* **by**(*simp add*: *split-def weight1 measure-spmf.emeasure-eq-measure*)
finally have *game2-d-bad*: *spmf* (*map-spmf snd game2-d*) *True* ≤ *q2* / *2* ˆ *len*
by *simp*

def *game3* ≡ *do* {
  (((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* 𝒜*1 Map.empty*;
  if valid-plain p0 ∧ *valid-plain p1 then do* {
    b ← *coin-spmf*;
    let pb = (*if b then p0 else p1*);
    r ← *spmf-of-set* (*nlists UNIV len*);
    pad ← *spmf-of-set* (*nlists UNIV len*);
    let cipher = (*r*, *xor-list pb pad*);
    (*b'*, *s2*) ← *exec-gpv rf-encrypt* (𝒜*2 cipher* σ) *s1*;
    return-spmf (*b'* = *b*)
  } *else coin-spmf*
}
have *bisim2d-3*: ⋀*cipher* σ *s r*. *rel-spmf* (λ(*x*, -) (*y*, -). *x* = *y*)
  (*exec-gpv* (*rf-encrypt-bad r*) (𝒜*2 cipher* σ) (*s*, *False*))
  (*exec-gpv rf-encrypt* (𝒜*2 cipher* σ) *s*)
by(*rule exec-gpv-oracle-bisim*[**where** *X*=λ(*s1*, -) *s2*. *s1* = *s2*])(*auto simp add*: *rf-encrypt-bad-def rf-encrypt-def intro*!: *rel-spmf-bind-reflI*)
have *game2d-3*: *rel-spmf op* = (*map-spmf fst game2-d*) *game3*
unfolding *game2-d-def game3-def Let-def* **including** *monad-normalisation*
by(*clarsimp simp add*: *o-def split-def if-distribs cong*: *if-cong intro*!: *rel-spmf-bind-reflI rel-spmf-bindI*[*OF bisim2d-3*])

have |*spmf game2 True* − *1* / *2*| ≤
  |*spmf* (*map-spmf fst game2-a*) *True* − *spmf* (*map-spmf fst game2-b*) *True*| + |*spmf* (*map-spmf fst game2-b*) *True* − *1* / *2*|
unfolding *game2-2a* **by**(*rule abs-diff-triangle-ineq2*)
also have ... ≤ *q1* / *2* ˆ *len* + |*spmf* (*map-spmf fst game2-b*) *True* − *1* / *2*|
using *game2a-2b game2a-bad-bound* **unfolding** *game2a-bad* **by**(*intro add-right-mono*) *simp*
also have |*spmf* (*map-spmf fst game2-b*) *True* − *1* / *2*| ≤
  |*spmf* (*map-spmf fst game2-c*) *True* − *spmf* (*map-spmf fst game2-d*) *True*| + |*spmf* (*map-spmf fst game2-d*) *True* − *1* / *2*|
unfolding *game2b-c* **by**(*rule abs-diff-triangle-ineq2*)
also (*add-left-mono-trans*) **have** ... ≤ *q2* / *2* ˆ *len* + |*spmf* (*map-spmf fst game2-d*) *True* − *1* / *2*|
using *game2c-2d game2-d-bad* **unfolding** *game2c-2d-bad* **by**(*intro add-right-mono*) *simp*
finally (*add-left-mono-trans*)
have *game2*: |*spmf game2 True* − *1* / *2*| ≤ *q1* / *2* ˆ *len* + *q2* / *2* ˆ *len* + |*spmf game3 True* − *1* / *2*|
using *game2d-3* **by**(*simp add*: *field-simps spmf-rel-eq*)

have *game3* = *do* {
  (((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* 𝒜*1 Map.empty*;
  if valid-plain p0 ∧ *valid-plain p1 then do* {
    b ← *coin-spmf*;
    let pb = (*if b then p0 else p1*);
    r ← *spmf-of-set* (*nlists UNIV len*);

pad ← *map-spmf* (*xor-list pb*) (*spmf-of-set* (*nlists UNIV len*));
let cipher = (*r*, *xor-list pb pad*);
(*b'*, *s2*) ← *exec-gpv rf-encrypt* ($\mathcal{A}$*2 cipher* σ) *s1*;
return-spmf (*b'* = *b*)
} *else coin-spmf*
}
by(*simp add*: *valid-plain-def game3-def Let-def one-time-pad del*: *bind-map-spmf map-spmf-of-set-inj-on cong*: *bind-spmf-cong-simp if-cong split*: *if-split*)
also have ... = *do* {
(((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* $\mathcal{A}$*1 Map.empty*;
if valid-plain p0 ∧ *valid-plain p1 then do* {
b ← *coin-spmf*;
let pb = (*if b then p0 else p1*);
r ← *spmf-of-set* (*nlists UNIV len*);
pad ← *spmf-of-set* (*nlists UNIV len*);
let cipher = (*r*, *pad*);
(*b'*, -) ← *exec-gpv rf-encrypt* ($\mathcal{A}$*2 cipher* σ) *s1*;
return-spmf (*b'* = *b*)
} *else coin-spmf*
}
by(*simp add*: *game3-def Let-def valid-plain-def in-nlists-UNIV cong*: *bind-spmf-cong-simp if-cong split*: *if-split*)
also have ... = *do* {
(((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* $\mathcal{A}$*1 Map.empty*;
if valid-plain p0 ∧ *valid-plain p1 then do* {
r ← *spmf-of-set* (*nlists UNIV len*);
pad ← *spmf-of-set* (*nlists UNIV len*);
let cipher = (*r*, *pad*);
(*b'*, -) ← *exec-gpv rf-encrypt* ($\mathcal{A}$*2 cipher* σ) *s1*;
map-spmf (*op* = *b'*) *coin-spmf*
} *else coin-spmf*
}
including *monad-normalisation* **by**(*simp add*: *map-spmf-conv-bind-spmf split-def Let-def*)
also have ... = *coin-spmf*
by(*simp add*: *map-eq-const-coin-spmf Let-def split-def weight2 weight1*)
finally have *game3*: *game3* = *coin-spmf* .

have *ind-cpa.advantage* $\mathcal{A}$ ≤ *prf.advantage* (*prf-adversary* $\mathcal{A}$) + |*spmf* (*prf.game-1* (*prf-adversary* $\mathcal{A}$)) *True* − *1* / *2*|
unfolding *ind-cpa.advantage-def prf.advantage-def ind-cpa-0*[*unfolded spmf-rel-eq*]
by(*rule abs-diff-triangle-ineq2*)
also have |*spmf* (*prf.game-1* (*prf-adversary* $\mathcal{A}$)) *True* − *1* / *2*| ≤ *q1* / *2* ^ *len* + *q2* / *2* ^ *len*
using *game1-2 game2 game3* **by**(*simp add*: *spmf-of-set*)
also have ... = (*q1* + *q2*) / *2* ^ *len* **by**(*simp add*: *field-simps*)
also have ... ≤ *q* / *2* ^ *len* **using** ‹*q1* + *q2* ≤ *q*› **by**(*simp add*: *divide-right-mono*)
finally show *?thesis* **by**(*simp add*: *field-simps*)
qed

lemma *interaction-bounded-prf-adversary*:
fixes *q* :: *nat*
assumes *ind-cpa.ibounded-by* $\mathcal{A}$ *q*
shows *prf.ibounded-by* (*prf-adversary* $\mathcal{A}$) (*1* + *q*)
proof −
fix η
from *assms* **have** *ind-cpa.ibounded-by* $\mathcal{A}$ *q* **by** *blast*
then obtain *q1 q2* **where** *q*: *q1* + *q2* ≤ *q*
and [*interaction-bound*]: *interaction-any-bounded-by* (*fst* $\mathcal{A}$) *q1*
$\bigwedge$*x* σ. *interaction-any-bounded-by* (*snd* $\mathcal{A}$ *x* σ) *q2*
unfolding *ind-cpa.ibounded-by-def* **by**(*auto simp add*: *split-beta iadd-le-enat-iff*)
show *prf.ibounded-by* (*prf-adversary* $\mathcal{A}$) (*1* + *q*) **using** *q*
unfolding *prf-adversary-def Let-def split-def*
by −(*interaction-bound*, *auto simp add*: *iadd-SUP-le-iff SUP-le-iff add.assoc*[*symmetric*] *one-enat-def*)
qed

lemma *lossless-prf-adversary*: *ind-cpa.lossless* $\mathcal{A}$ ⟹ *prf.lossless* (*prf-adversary* $\mathcal{A}$)
by(*fastforce simp add*: *prf-adversary-def Let-def split-def ind-cpa.lossless-def intro*: *lossless-inline*)

end

locale *otp-*η =
fixes *f* :: *security* ⇒ *key* ⇒ *bool list* ⇒ *bool list*
and *len* :: *security* ⇒ *nat*
assumes *length-f*: $\bigwedge\eta$ *xs ys*. ⟦ *length xs* = *len* η; *length ys* = *len* η ⟧ ⟹ *length* (*f* η *xs ys*) = *len* η
and *negligible-len* [*negligible-intros*]: *negligible* ($\lambda\eta$. *1* / *2* ^ (*len* η))
begin

interpretation *otp f* η *len* η **for** η **by**(*unfold-locales*)(*rule length-f*)
interpretation *ind-cpa*: *ind-cpa key-gen* η *encrypt* η *decrypt* η *valid-plain* η **for** η **.**
interpretation *prf*: *prf key-gen* η *f* η *spmf-of-set* (*nlists UNIV* (*len* η)) **for** η **.**

lemma *prf-encrypt-secure-for*:
assumes [*negligible-intros*]: *negligible* ($\lambda\eta$. *prf.advantage* η (*prf-adversary* η ($\mathcal{A}$ η)))
and *q*: $\bigwedge\eta$. *ind-cpa.ibounded-by* ($\mathcal{A}$ η) (*q* η) **and** [*negligible-intros*]: *polynomial q*
and *lossless*: $\bigwedge\eta$. *ind-cpa.lossless* ($\mathcal{A}$ η)
shows *negligible* ($\lambda\eta$. *ind-cpa.advantage* η ($\mathcal{A}$ η))
proof(*rule negligible-mono*)
show *negligible* ($\lambda\eta$. *prf.advantage* η (*prf-adversary* η ($\mathcal{A}$ η)) + *q* η / *2* ^ *len* η)
by(*intro negligible-intros*)
{ fix η
from ‹*ind-cpa.ibounded-by* - -› **have** *ind-cpa.ibounded-by* ($\mathcal{A}$ η) (*q* η) **by** *blast*

moreover from *lossless* **have** *ind-cpa.lossless* ($\mathcal{A}$ η) **by** *blast*
hence *lossless-gpv* $\mathcal{I}$*-full* (*fst* ($\mathcal{A}$ η)) $\bigwedge$*cipher* σ. *lossless-gpv* $\mathcal{I}$*-full* (*snd* ($\mathcal{A}$ η) *cipher* σ)
by(*auto simp add*: *ind-cpa.lossless-def*)
ultimately have *ind-cpa.advantage* η ($\mathcal{A}$ η) $\leq$ *prf.advantage* η (*prf-adversary* η ($\mathcal{A}$ η)) + *q* η / *2* ^ *len* η
by(*rule prf-encrypt-advantage*) **}**
hence *eventually* ($\lambda\eta$. |*ind-cpa.advantage* η ($\mathcal{A}$ η)| $\leq$ *1* $*$ |*prf.advantage* η (*prf-adversary* η ($\mathcal{A}$ η)) + *q* η / *2* ^ *len* η|) *at-top*
by(*simp add*: *always-eventually ind-cpa.advantage-nonneg prf.advantage-nonneg*)
then show ($\lambda\eta$. *ind-cpa.advantage* η ($\mathcal{A}$ η)) $\in$ *O*($\lambda\eta$. *prf.advantage* η (*prf-adversary* η ($\mathcal{A}$ η)) + *q* η / *2* ^ *len* η)
by(*intro bigoI*[**where** *c*=*1*]) *simp*
qed

end

end

2.6 IND-CCA from a PRF and an unpredictable function

theory *PRF-UPF-IND-CCA*
imports
Pseudo-Random-Function
CryptHOL.List-Bits
Unpredictable-Function
IND-CCA2-sym
CryptHOL.Negligible
begin

Formalisation of Shoup's construction of an IND-CCA secure cipher from a PRF and an unpredictable function [4, §7].

type-synonym *bitstring* = *bool list*

locale *simple-cipher* =
PRF: *prf prf-key-gen prf-fun spmf-of-set* (*nlists UNIV prf-clen*) +
UPF: *upf upf-key-gen upf-fun*
for *prf-key-gen* :: *'prf-key spmf*
and *prf-fun* :: *'prf-key* $\Rightarrow$ *bitstring* $\Rightarrow$ *bitstring*
and *prf-domain* :: *bitstring set*
and *prf-range* :: *bitstring set*
and *prf-dlen* :: *nat*
and *prf-clen* :: *nat*
and *upf-key-gen* :: *'upf-key spmf*
and *upf-fun* :: *'upf-key* $\Rightarrow$ *bitstring* $\Rightarrow$ *'hash*
+
assumes *prf-domain-finite*: *finite prf-domain*
assumes *prf-domain-nonempty*: *prf-domain* $\neq$ {}
assumes *prf-domain-length*: *x* $\in$ *prf-domain* $\Longrightarrow$ *length x* = *prf-dlen*

assumes *prf-codomain-length*:
⟦ *key-prf* ∈ *set-spmf prf-key-gen*; *m* ∈ *prf-domain* ⟧ ⟹ *length* (*prf-fun key-prf m*) = *prf-clen*
assumes *prf-key-gen-lossless*: *lossless-spmf prf-key-gen*
assumes *upf-key-gen-lossless*: *lossless-spmf upf-key-gen*
begin

type-synonym *'hash' cipher-text* = *bitstring* × *bitstring* × *'hash'*

definition *key-gen* :: (*'prf-key* × *'upf-key*) *spmf* **where**
key-gen = *do* {
k-prf ← *prf-key-gen*;
k-upf :: *'upf-key* ← *upf-key-gen*;
return-spmf (*k-prf*, *k-upf*)
}

lemma *lossless-key-gen* [*simp*]: *lossless-spmf key-gen*
by(*simp add*: *key-gen-def prf-key-gen-lossless upf-key-gen-lossless*)

fun *encrypt* :: (*'prf-key* × *'upf-key*) ⇒ *bitstring* ⇒ *'hash cipher-text spmf*
where
encrypt (*k-prf*, *k-upf*) *m* = *do* {
x ← *spmf-of-set prf-domain*;
let c = *prf-fun k-prf x* [⊕] *m*;
let t = *upf-fun k-upf* (*x* @ *c*);
return-spmf ((*x*, *c*, *t*))
}

lemma *lossless-encrypt* [*simp*]: *lossless-spmf* (*encrypt k m*)
by (*cases k*) (*simp add*: *Let-def prf-domain-nonempty prf-domain-finite split*: *bool.split*)

fun *decrypt* :: (*'prf-key* × *'upf-key*) ⇒ *'hash cipher-text* ⇒ *bitstring option*
where
decrypt (*k-prf*, *k-upf*) (*x*, *c*, *t*) = (
if upf-fun k-upf (*x* @ *c*) = *t* ∧ *length x* = *prf-dlen then*
Some (*prf-fun k-prf x* [⊕] *c*)
else
None
)

lemma *cipher-correct*:
⟦ *k* ∈ *set-spmf key-gen*; *length m* = *prf-clen* ⟧
⟹ *encrypt k m* ⤜ (λ*c*. *return-spmf* (*decrypt k c*)) = *return-spmf* (*Some m*)
by (*cases k*) (*simp add*: *prf-domain-nonempty prf-domain-finite prf-domain-length prf-codomain-length key-gen-def bind-eq-return-spmf Let-def*)

declare *encrypt.simps*[*simp del*]

sublocale *ind-cca*: *ind-cca key-gen encrypt decrypt* $\lambda m.$ *length* m = *prf-clen* **.**
interpretation *ind-cca'*: *ind-cca key-gen encrypt* λ - -. *None* $\lambda m.$ *length* m = *prf-clen* **.**

definition *intercept-upf-enc*
:: *'prf-key* ⇒ *bool* ⇒ *'hash cipher-text set* × *'hash cipher-text set* ⇒ *bitstring* × *bitstring*
⇒ (*'hash cipher-text option* × (*'hash cipher-text set* × *'hash cipher-text set*),
bitstring + (*bitstring* × *'hash*), *'hash* + *unit*) *gpv*
where
intercept-upf-enc k b = (λ(*L*, *D*) (*m1*, *m0*).
(*case* (*length m1* = *prf-clen* ∧ *length m0* = *prf-clen*) *of*
False ⇒ *Done* (*None*, *L*, *D*)
| *True* ⇒ *do* {
x ← *lift-spmf* (*spmf-of-set prf-domain*);
let c = *prf-fun k x* [⊕] (*if b then m1 else m0*);
t ← *Pause* (*Inl* (*x* @ *c*)) *Done*;
Done ((*Some* (*x*, *c*, *projl t*)), (*insert* (*x*, *c*, *projl t*) *L*, *D*))
}))

definition *intercept-upf-dec*
:: *'hash cipher-text set* × *'hash cipher-text set* ⇒ *'hash cipher-text*
⇒ (*bitstring option* × (*'hash cipher-text set* × *'hash cipher-text set*),
bitstring + (*bitstring* × *'hash*), *'hash* + *unit*) *gpv*
where
intercept-upf-dec = (λ(*L*, *D*) (*x*, *c*, *t*).
if (*x*, *c*, *t*) ∈ *L* ∨ *length x* ≠ *prf-dlen then Done* (*None*, (*L*, *D*)) *else do* {
Pause (*Inr* (*x* @ *c*, *t*)) *Done*;
Done (*None*, (*L*, *insert* (*x*, *c*, *t*) *D*))
})

definition *intercept-upf* ::
'prf-key ⇒ *bool* ⇒ *'hash cipher-text set* × *'hash cipher-text set* ⇒ *bitstring* × *bitstring* + *'hash cipher-text*
⇒ ((*'hash cipher-text option* + *bitstring option*) × (*'hash cipher-text set* × *'hash cipher-text set*),
bitstring + (*bitstring* × *'hash*), *'hash* + *unit*) *gpv*
where
intercept-upf k b = *plus-intercept* (*intercept-upf-enc k b*) *intercept-upf-dec*

lemma *intercept-upf-simps* [*simp*]:
intercept-upf k b (*L*, *D*) (*Inr* (*x*, *c*, *t*)) =
(*if* (*x*, *c*, *t*) ∈ *L* ∨ *length x* ≠ *prf-dlen then Done* (*Inr None*, (*L*, *D*)) *else do* {
Pause (*Inr* (*x* @ *c*, *t*)) *Done*;
Done (*Inr None*, (*L*, *insert* (*x*, *c*, *t*) *D*))
})
intercept-upf k b (*L*, *D*) (*Inl* (*m1*, *m0*)) =
(*case* (*length m1* = *prf-clen* ∧ *length m0* = *prf-clen*) *of*
False ⇒ *Done* (*Inl None*, *L*, *D*)

| *True* ⇒ *do* {
x ← *lift-spmf* (*spmf-of-set prf-domain*);
let c = *prf-fun k x* [⊕] (*if b then m1 else m0*);
t ← *Pause* (*Inl* (*x* @ *c*)) *Done*;
Done (*Inl* (*Some* (*x*, *c*, *projl t*)), (*insert* (*x*, *c*, *projl t*) *L*, *D*))
})
by(*simp-all add*: *intercept-upf-def intercept-upf-dec-def intercept-upf-enc-def o-def map-gpv-bind-gpv gpv.map-id Let-def split*!: *bool.split*)

lemma *interaction-bounded-by-upf-enc-Inr* [*interaction-bound*]:
interaction-bounded-by (*Not* ∘ *isl*) (*intercept-upf-enc k b LD mm*) *0*
unfolding *intercept-upf-enc-def case-prod-app*
by(*interaction-bound*, *clarsimp simp add*: *SUP-constant bot-enat-def split*: *prod.split*)

lemma *interaction-bounded-by-upf-dec-Inr* [*interaction-bound*]:
interaction-bounded-by (*Not* ∘ *isl*) (*intercept-upf-dec LD c*) *1*
unfolding *intercept-upf-dec-def case-prod-app*
by(*interaction-bound*, *clarsimp simp add*: *SUP-constant split*: *prod.split*)

lemma *interaction-bounded-by-intercept-upf-Inr* [*interaction-bound*]:
interaction-bounded-by (*Not* ∘ *isl*) (*intercept-upf k b LD x*) *1*
unfolding *intercept-upf-def*
by *interaction-bound*(*simp add*: *split-def one-enat-def SUP-le-iff split*: *sum.split*)

lemma *interaction-bounded-by-intercept-upf-Inl* [*interaction-bound*]:
isl x ⟹ *interaction-bounded-by* (*Not* ∘ *isl*) (*intercept-upf k b LD x*) *0*
unfolding *intercept-upf-def case-prod-app*
by *interaction-bound*(*auto split*: *sum.split*)

lemma *lossless-intercept-upf-enc* [*simp*]: *lossless-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) (*intercept-upf-enc k b LD mm*)
by(*simp add*: *intercept-upf-enc-def split-beta prf-domain-finite prf-domain-nonempty Let-def split*: *bool.split*)

lemma *lossless-intercept-upf-dec* [*simp*]: *lossless-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) (*intercept-upf-dec LD mm*)
by(*simp add*: *intercept-upf-dec-def split-beta*)

lemma *lossless-intercept-upf* [*simp*]: *lossless-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) (*intercept-upf k b LD x*)
by(*cases x*)(*simp-all add*: *intercept-upf-def*)

lemma *results-gpv-intercept-upf* [*simp*]: *results-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) (*intercept-upf k b LD x*) ⊆ *responses-*$\mathcal{I}$ ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) *x* × *UNIV*
by(*cases x*)(*auto simp add*: *intercept-upf-def*)

definition *reduction-upf* :: (*bitstring*, *'hash cipher-text*) *ind-cca.adversary*
⇒ (*bitstring*, *'hash*) *UPF.adversary*

where *reduction-upf* $\mathcal{A}$ = *do* {
  $k \leftarrow$ *lift-spmf prf-key-gen*;
  $b \leftarrow$ *lift-spmf coin-spmf*;
  (-, (*L*, *D*)) $\leftarrow$ *inline* (*intercept-upf k b*) $\mathcal{A}$ ({}, {});
  Done () }

lemma *lossless-reduction-upf* [*simp*]:
lossless-gpv ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\mathcal{A} \Longrightarrow$ *lossless-gpv* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) (*reduction-upf* $\mathcal{A}$)
by(*auto simp add*: *reduction-upf-def prf-key-gen-lossless intro*: *lossless-inline del*: *subsetI*)

context includes *lifting-syntax* **begin**

lemma *round-1*:
assumes *lossless-gpv* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\mathcal{A}$
shows |*spmf* (*ind-cca.game* $\mathcal{A}$) *True* $-$ *spmf* (*ind-cca*$'$*.game* $\mathcal{A}$) *True*| $\leq$ *UPF.advantage* (*reduction-upf* $\mathcal{A}$)
proof $-$
define *oracle-decrypt0*$'$ **where** *oracle-decrypt0*$'$ $\equiv$ (λ*key* (*bad*, *L*) (x', c', t'). *return-spmf* (
  if (x', c', t') $\in L \vee$ *length* $x' \neq$ *prf-dlen then* (*None*, (*bad*, *L*))
  else (*decrypt key* (x', c', t'), (*bad* $\vee$ *upf-fun* (*snd key*) (x' @ c') = t', *L*))))
have *oracle-decrypt0*$'$*-simps*:
  oracle-decrypt0$'$ *key* (*bad*, *L*) (x', c', t') = *return-spmf* (
    if (x', c', t') $\in L \vee$ *length* $x' \neq$ *prf-dlen then* (*None*, (*bad*, *L*))
    else (*decrypt key* (x', c', t'), (*bad* $\vee$ *upf-fun* (*snd key*) (x' @ c') = t', *L*)))
  for *key L bad* x' c' t' **by**(*simp add*: *oracle-decrypt0*$'$*-def*)
have *lossless-oracle-decrypt0*$'$ [*simp*]: *lossless-spmf* (*oracle-decrypt0*$'$ *k Lbad c*) **for** *k Lbad c*
  by(*simp add*: *oracle-decrypt0*$'$*-def split-def*)
have *callee-invariant-oracle-decrypt0*$'$ [*simp*]: *callee-invariant* (*oracle-decrypt0*$'$ *k*) *fst* **for** *k*
  by (*unfold-locales*) (*auto simp add*: *oracle-decrypt0*$'$*-def split*: *if-split-asm*)

def *oracle-decrypt1*$'$ $\equiv$ λ(*key* :: $'$*prf-key* $\times$ $'$*upf-key*) (*bad*, *L*) (x', c', t').
  return-spmf (*None* :: *bitstring option*,
    (*bad* $\vee$ *upf-fun* (*snd key*) (x' @ c') = t' $\wedge$ (x', c', t') $\notin L \wedge$ *length* x' = *prf-dlen*), *L*)
have *oracle-decrypt1*$'$*-simps*:
  oracle-decrypt1$'$ *key* (*bad*, *L*) (x', c', t') =
  return-spmf (*None*,
    (*bad* $\vee$ *upf-fun* (*snd key*) (x' @ c') = t' $\wedge$ (x', c', t') $\notin L \wedge$ *length* x' = *prf-dlen*, *L*))
  for *key L bad* x' c' t' **by**(*simp add*: *oracle-decrypt1*$'$*-def*)
have *lossless-oracle-decrypt1*$'$ [*simp*]: *lossless-spmf* (*oracle-decrypt1*$'$ *k Lbad c*) **for** *k Lbad c*
  by(*simp add*: *oracle-decrypt1*$'$*-def split-def*)
have *callee-invariant-oracle-decrypt1*$'$ [*simp*]: *callee-invariant* (*oracle-decrypt1*$'$

k) *fst* **for** *k*
  by (*unfold-locales*) (*auto simp add*: *oracle-decrypt1'-def*)

 def *game01'* ≡ λ(*decrypt* :: *'prf-key* × *'upf-key* ⇒ (*bitstring* × *bitstring* × *'hash*, *bitstring option*, *bool* × (*bitstring* × *bitstring* × *'hash*) *set*) *callee*) 𝒜. *do* {
  key ← *key-gen*;
  b ← *coin-spmf*;
  (*b'*, (*bad'*, *L'*)) ← *exec-gpv* (†(*ind-cca.oracle-encrypt key b*) $\oplus_O$ *decrypt key*) 𝒜 (*False*, {});
  return-spmf (*b* = *b'*, *bad'*) }
 let *?game0'* = *game01' oracle-decrypt0'*
 let *?game1'* = *game01' oracle-decrypt1'*

 have *game0'-eq*: *ind-cca.game* 𝒜 = *map-spmf fst* (*?game0'* 𝒜) (**is** *?game0*)
  and *game1'-eq*: *ind-cca'.game* 𝒜 = *map-spmf fst* (*?game1'* 𝒜) (**is** *?game1*)
 proof −
  let *?S* = *rel-prod2 op* =
  def *initial* ≡ (*False*, {} :: *'hash cipher-text set*)
  have [*transfer-rule*]: *?S* {} *initial* **by**(*simp add*: *initial-def*)

  have [*transfer-rule*]:
   (*op* = ===> *?S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *?S*))
   ind-cca.oracle-decrypt oracle-decrypt0'
   unfolding *ind-cca.oracle-decrypt-def*[*abs-def*] *oracle-decrypt0'-def*[*abs-def*]
   by(*simp add*: *rel-spmf-return-spmf1 rel-fun-def*)

  have [*transfer-rule*]:
   (*op* = ===> *?S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *?S*))
   ind-cca'.oracle-decrypt oracle-decrypt1'
   unfolding *ind-cca'.oracle-decrypt-def*[*abs-def*] *oracle-decrypt1'-def*[*abs-def*]
   by (*simp add*: *rel-spmf-return-spmf1 rel-fun-def*)

  note [*transfer-rule*] = *extend-state-oracle-transfer*
  show *?game0 ?game1* **unfolding** *game01'-def ind-cca.game-def ind-cca'.game-def initial-def*[*symmetric*]
   by (*simp-all add*: *map-spmf-bind-spmf o-def split-def*) *transfer-prover+*
 qed

 have ∗: *rel-spmf* (λ(*b'1*, (*bad1*, *L1*)) (*b'2*, (*bad2*, *L2*)). *bad1* = *bad2* ∧ (¬ *bad2* ⟶ *b'1* = *b'2*))
    (*exec-gpv* (†(*ind-cca.oracle-encrypt k b*) $\oplus_O$ *oracle-decrypt1' k*) 𝒜 (*False*, {}))
    (*exec-gpv* (†(*ind-cca.oracle-encrypt k b*) $\oplus_O$ *oracle-decrypt0' k*) 𝒜 (*False*, {}))
  for *k b*
  by (*cases k*; *rule exec-gpv-oracle-bisim-bad*[**where** *X*=*op* = **and** *?bad1.0*=*fst* **and** *?bad2.0*=*fst* **and** ℐ = ℐ*-full* $\oplus_\mathcal{I}$ ℐ*-full*])
   (*auto intro*: *rel-spmf-reflI callee-invariant-extend-state-oracle-const' simp add*: *spmf-rel-map1 spmf-rel-map2 oracle-decrypt0'-simps oracle-decrypt1'-simps assms*

split: *plus-oracle-split*)

— We cannot get rid of the losslessness assumption on $\mathcal{A}$ in this step, because if it were not, then the bad event might still occur, but the adversary does not terminate in the case of *game01′ oracle-decrypt1′*. Thus, the reduction does not terminate either, but it cannot detect whether the bad event has happened. So the advantage in the UPF game could be lower than the probability of the bad event, if the adversary is not lossless.

have |*measure* (*measure-spmf* (*?game1′* $\mathcal{A}$)) {(*b*, *bad*). *b*} − *measure* (*measure-spmf* (*?game0′* $\mathcal{A}$)) {(*b*, *bad*). *b*}|
  ≤ *measure* (*measure-spmf* (*?game1′* $\mathcal{A}$)) {(*b*, *bad*). *bad*}
 by (*rule fundamental-lemma*[**where** *?bad2.0*=*snd*])(*auto intro*!: *rel-spmf-bind-reflI rel-spmf-bindI*[*OF* ∗] *simp add*: *game01′-def*)
 also have … = *spmf* (*map-spmf snd* (*?game1′* $\mathcal{A}$)) *True*
  by (*simp add*: *spmf-conv-measure-spmf measure-map-spmf split-def vimage-def*)
 also have *map-spmf snd* (*?game1′* $\mathcal{A}$) = *UPF.game* (*reduction-upf* $\mathcal{A}$)
 proof −
  note [*split del*] = *if-split*
  have *map-spmf* (λ*x*. *fst* (*snd x*)) (*exec-gpv* (†(*ind-cca.oracle-encrypt* (*k-prf*, *k-upf*) *b*) $\oplus_O$ *oracle-decrypt1′* (*k-prf*, *k-upf*)) $\mathcal{A}$ (*False*, {})) =
   map-spmf (λ*x*. *fst* (*snd x*)) (*exec-gpv* (*UPF.oracle k-upf*) (*inline* (*intercept-upf k-prf b*) $\mathcal{A}$ ({}, {})) (*False*, {}))
  (**is** *map-spmf ?fl ?lhs* = *map-spmf ?fr ?rhs* **is** *map-spmf* - (*exec-gpv ?oracle-normal* - *?init-normal*) = -)
   for *k-prf k-upf b*
  proof(*rule map-spmf-eq-map-spmfI*)
   def [*simp*]: *oracle-intercept* ≡ λ(*s′*, *s*) *y*. *map-spmf* (λ((*x*, *s′*), *s*). (*x*, *s′*, *s*))
    (*exec-gpv* (*UPF.oracle k-upf*) (*intercept-upf k-prf b s′ y*) *s*)
   let *?I* = (λ((*L*, *D*), (*flg*, *Li*)).
     (∀(*x*, *c*, *t*) ∈ *L*. *upf-fun k-upf* (*x* @ *c*) = *t* ∧ *length x* = *prf-dlen*) ∧
     (∀ *e*∈*Li*. ∃(*x*,*c*,-) ∈ *L*. *e* = *x* @ *c*) ∧
     ((∃(*x*, *c*, *t*) ∈ *D*. *upf-fun k-upf* (*x* @ *c*) = *t*) ⟷ *flg*))
   interpret *callee-invariant-on oracle-intercept ?I* $\mathcal{I}$*-full*
    apply(*unfold-locales*)
    subgoal for *s x y s′*
     apply(*cases s*; *cases s′*; *cases x*)
      apply(*clarsimp simp add*: *set-spmf-of-set-finite*[*OF prf-domain-finite*]
        UPF.oracle-hash-def prf-domain-length exec-gpv-bind Let-def split: *bool.splits*)
     apply(*force simp add*: *exec-gpv-bind UPF.oracle-flag-def split*: *if-split-asm*)
     done
    subgoal by *simp*
    done

   def *S* ≡ (λ(*bad*, *L1*) ((*L2*, *D*), -). *bad* = (∃(*x*, *c*, *t*)∈*D*. *upf-fun k-upf* (*x* @ *c*) = *t*) ∧ *L1* = *L2*) ↿ (λ-. *True*) ⊗ *?I*
    :: *bool* × *′hash cipher-text set* ⇒ (*′hash cipher-text set* × *′hash cipher-text set*) × *bool* × *bitstring set* ⇒ *bool*
   def *initial* ≡ (({}, {}), (*False*, {})) :: (*′hash cipher-text set* × *′hash cipher-text set*) × *bool* × *bitstring set*

have [*transfer-rule*]: *S ?init-normal initial* **by**(*simp add*: *S-def initial-def*)
have [*transfer-rule*]: (*S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *S*)) *?oracle-normal oracle-intercept*
unfolding *S-def*
by(*rule callee-invariant-restrict-relp*, *unfold-locales*)
(*auto simp add*: *rel-fun-def bind-spmf-of-set prf-domain-finite prf-domain-nonempty bind-spmf-pmf-assoc bind-assoc-pmf bind-return-pmf spmf-rel-map exec-gpv-bind Let-def ind-cca.oracle-encrypt-def oracle-decrypt1′-def encrypt.simps UPF.oracle-hash-def UPF.oracle-flag-def bind-map-spmf o-def split*: *plus-oracle-split bool.split if-split intro*!: *rel-spmf-bind-reflI rel-pmf-bind-reflI*)
have *rel-spmf* (*rel-prod op* = *S*) *?lhs* (*exec-gpv oracle-intercept* $\mathcal{A}$ *initial*)
by(*transfer-prover*)
then show *rel-spmf* (λ*x y*. *?fl x* = *?fr y*) *?lhs ?rhs*
by(*auto simp add*: *S-def exec-gpv-inline spmf-rel-map initial-def elim*: *rel-spmf-mono*)
qed
then show *?thesis* **including** *monad-normalisation*
by(*auto simp add*: *reduction-upf-def UPF.game-def game01′-def key-gen-def map-spmf-conv-bind-spmf split-def exec-gpv-bind intro*!: *bind-spmf-cong*[*OF refl*])
qed
finally show *?thesis* **using** *game0′-eq game1′-eq*
by (*auto simp add*: *spmf-conv-measure-spmf measure-map-spmf vimage-def fst-def UPF.advantage-def*)
qed

definition *oracle-encrypt2* ::
(′*prf-key* × ′*upf-key*) ⇒ *bool* ⇒ (*bitstring*, *bitstring*) *PRF.dict* ⇒ *bitstring* × *bitstring*
⇒ (′*hash cipher-text option* × (*bitstring*, *bitstring*) *PRF.dict*) *spmf*
where
oracle-encrypt2 = (λ(*k-prf*, *k-upf*) *b D* (*msg1*, *msg0*). (*case* (*length msg1* = *prf-clen* ∧ *length msg0* = *prf-clen*) *of*
False ⇒ *return-spmf* (*None*, *D*)
| *True* ⇒ *do* {
x ← *spmf-of-set prf-domain*;
P ← *spmf-of-set* (*nlists UNIV prf-clen*);
let p = (*case D x of Some r* ⇒ *r* | *None* ⇒ *P*);
let c = *p* [⊕] (*if b then msg1 else msg0*);
let t = *upf-fun k-upf* (*x* @ *c*);
return-spmf (*Some* (*x*, *c*, *t*), *D*(*x* ↦ *p*))
}))

definition *oracle-decrypt2*:: (′*prf-key* × ′*upf-key*) ⇒ (′*hash cipher-text*, *bitstring option*, ′*state*) *callee*
where *oracle-decrypt2* = (λ*key D cipher*. *return-spmf* (*None*, *D*))

lemma *lossless-oracle-decrypt2* [*simp*]: *lossless-spmf* (*oracle-decrypt2 k Dbad c*)
by(*simp add*: *oracle-decrypt2-def split-def*)

lemma *callee-invariant-oracle-decrypt2* [*simp*]: *callee-invariant* (*oracle-decrypt2 key*) *fst*
  by (*unfold-locales*) (*auto simp add*: *oracle-decrypt2-def split*: *if-split-asm*)

lemma *oracle-decrypt2-parametric* [*transfer-rule*]:
  (*rel-prod P U* ===> *S* ===> *rel-prod op* = (*rel-prod op* = *H*) ===> *rel-spmf* (*rel-prod op* = *S*))
    oracle-decrypt2 oracle-decrypt2
  unfolding *oracle-decrypt2-def split-def relator-eq*[*symmetric*] **by** *transfer-prover*

definition *game2* :: (*bitstring*, *'hash cipher-text*) *ind-cca.adversary* ⇒ *bool spmf*
where
  game2 $\mathcal{A}$ ≡ *do* {
    key ← *key-gen*;
    b ← *coin-spmf*;
    (*b'*, *D*) ← *exec-gpv*
      (*oracle-encrypt2 key b* $\oplus_O$ *oracle-decrypt2 key*) $\mathcal{A}$ *Map-empty*;
    return-spmf (*b* = *b'*)
  }

fun *intercept-prf* ::
  'upf-key ⇒ *bool* ⇒ *unit* ⇒ (*bitstring* × *bitstring*) + *'hash cipher-text*
  ⇒ ((*'hash cipher-text option* + *bitstring option*) × *unit*, *bitstring*, *bitstring*) *gpv*
where
  intercept-prf - - - (*Inr* -) = *Done* (*Inr None*, ())
| *intercept-prf k b* - (*Inl* (*m1*, *m0*)) = (*case* (*length m1*) = *prf-clen* ∧ (*length m0*) = *prf-clen of*
      False ⇒ *Done* (*Inl None*, ())
    | *True* ⇒ *do* {
        x ← *lift-spmf* (*spmf-of-set prf-domain*);
        p ← *Pause x Done*;
        let c = *p* [⊕] (*if b then m1 else m0*);
        let t = *upf-fun k* (*x* @ *c*);
        Done (*Inl* (*Some* (*x*, *c*, *t*)), ())
      })

definition *reduction-prf*
  :: (*bitstring*, *'hash cipher-text*) *ind-cca.adversary* ⇒ (*bitstring*, *bitstring*) *PRF.adversary*
where
  reduction-prf $\mathcal{A}$ = *do* {
    k ← *lift-spmf upf-key-gen*;
    b ← *lift-spmf coin-spmf*;
    (*b'*, -) ← *inline* (*intercept-prf k b*) $\mathcal{A}$ ();
    Done (*b'* = *b*)
  }

lemma *round-2*: |*spmf* (*ind-cca'.game* $\mathcal{A}$) *True* − *spmf* (*game2* $\mathcal{A}$) *True*| = *PRF.advantage* (*reduction-prf* $\mathcal{A}$)

```
proof −
  def oracle-encrypt1'' ≡ (λ(k-prf, k-upf) b (- :: unit) (msg1, msg0).
    case length msg1 = prf-clen ∧ length msg0 = prf-clen of
      False ⇒ return-spmf (None, ())
    | True ⇒ do {
        x ← spmf-of-set prf-domain;
        let p = prf-fun k-prf x;
        let c = p [⊕] (if b then msg1 else msg0);
        let t = upf-fun k-upf (x @ c);
        return-spmf (Some (x, c, t), ())})
  def game1''≡ do {
    key ← key-gen;
    b ← coin-spmf;
    (b', D) ← exec-gpv (oracle-encrypt1'' key b ⊕O oracle-decrypt2 key) 𝒜 ();
    return-spmf (b = b')}

  have ind-cca'.game 𝒜 = game1''
  proof −
    def S ≡ λ(L :: 'hash cipher-text set) (D :: unit). True
    have [transfer-rule]: S {} () by (simp add: S-def)
    have [transfer-rule]:
      (op = ===> op = ===> S ===> op = ===> rel-spmf (rel-prod op =
S))
       ind-cca'.oracle-encrypt oracle-encrypt1''
      unfolding ind-cca'.oracle-encrypt-def[abs-def] oracle-encrypt1''-def[abs-def]
       by (auto simp add: rel-fun-def Let-def S-def encrypt.simps prf-domain-finite
prf-domain-nonempty intro: rel-spmf-bind-reflI rel-pmf-bind-reflI split: bool.split)
    have [transfer-rule]:
      (op = ===> S ===> op = ===> rel-spmf (rel-prod op = S))
       ind-cca'.oracle-decrypt oracle-decrypt2
      unfolding ind-cca'.oracle-decrypt-def[abs-def] oracle-decrypt2-def[abs-def]
      by(auto simp add: rel-fun-def)
    show ?thesis unfolding ind-cca'.game-def game1''-def by transfer-prover
  qed

  also have ... = PRF.game-0 (reduction-prf 𝒜)
  proof −
    { fix k-prf k-upf b
       def oracle-normal ≡ oracle-encrypt1'' (k-prf, k-upf) b ⊕O oracle-decrypt2
(k-prf, k-upf)
      def oracle-intercept ≡ λ(s', s :: unit) y. map-spmf (λ((x, s'), s). (x, s', s))
(exec-gpv (PRF.prf-oracle k-prf) (intercept-prf k-upf b s' y) ())
      def initial ≡ ()
      def S ≡ λ(s2 :: unit, - :: unit) (s1 :: unit). True
      have [transfer-rule]: S ((), ()) initial by(simp add: S-def initial-def)
       have [transfer-rule]: (S ===> op = ===> rel-spmf (rel-prod op = S))
oracle-intercept oracle-normal
        unfolding oracle-normal-def oracle-intercept-def
     by(auto split: bool.split plus-oracle-split simp add: S-def rel-fun-def exec-gpv-bind
```

PRF.prf-oracle-def oracle-encrypt1''-def Let-def map-spmf-conv-bind-spmf oracle-decrypt2-def intro!: *rel-spmf-bind-reflI rel-spmf-reflI*)
have *map-spmf* (λx. *b* = *fst x*) (*exec-gpv oracle-normal* $\mathcal{A}$ *initial*) =
map-spmf (λx. *b* = *fst* (*fst x*)) (*exec-gpv* (*PRF.prf-oracle k-prf*) (*inline* (*intercept-prf k-upf b*) $\mathcal{A}$ ()) ())
by(*transfer fixing*: *b* $\mathcal{A}$ *prf-fun k-prf prf-domain prf-clen upf-fun k-upf*)
(*auto simp add*: *map-spmf-eq-map-spmf-iff exec-gpv-inline spmf-rel-map oracle-intercept-def split-def intro*: *rel-spmf-reflI*) **}**
then show *?thesis* **unfolding** *game1''-def PRF.game-0-def key-gen-def reduction-prf-def*
by (*auto simp add*: *exec-gpv-bind-lift-spmf exec-gpv-bind map-spmf-conv-bind-spmf split-def eq-commute intro*!: *bind-spmf-cong*[*OF refl*])
qed
also have *game2* $\mathcal{A}$ = *PRF.game-1* (*reduction-prf* $\mathcal{A}$)
proof −
note [*split del*] = *if-split*
{ fix *k-upf b k-prf*
def *oracle2* ≡ *oracle-encrypt2* (*k-prf*, *k-upf*) *b* $\oplus_O$ *oracle-decrypt2* (*k-prf*, *k-upf*)
def *oracle-intercept* ≡ (λ(*s'*, *s*) *y*. *map-spmf* (λ((*x*, *s'*), *s*). (*x*, *s'*, *s*)) (*exec-gpv PRF.random-oracle* (*intercept-prf k-upf b s' y*) *s*))
def *S* ≡ λ(*s2* :: *unit*, *s2'*) (*s1* :: (*bitstring*, *bitstring*) *PRF.dict*). *s2'* = *s1*

have [*transfer-rule*]: *S* ((), *Map-empty*) *Map-empty* **by**(*simp add*: *S-def*)
have [*transfer-rule*]: (*S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *S*)) *oracle-intercept oracle2*
unfolding *oracle2-def oracle-intercept-def*
by(*auto split*: *bool.split plus-oracle-split option.split simp add*: *S-def rel-fun-def exec-gpv-bind PRF.random-oracle-def oracle-encrypt2-def Let-def map-spmf-conv-bind-spmf oracle-decrypt2-def rel-spmf-return-spmf1 fun-upd-idem intro*!: *rel-spmf-bind-reflI rel-spmf-reflI*)

have [*symmetric*]: *map-spmf* (λx. *b* = *fst* (*fst x*)) (*exec-gpv* (*PRF.random-oracle*) (*inline* (*intercept-prf k-upf b*) $\mathcal{A}$ ()) *Map.empty*) =
map-spmf (λx. *b* = *fst x*) (*exec-gpv oracle2* $\mathcal{A}$ *Map-empty*)
by(*transfer fixing*: *b prf-clen prf-domain upf-fun k-upf* $\mathcal{A}$ *k-prf*)
(*simp add*: *exec-gpv-inline map-spmf-conv-bind-spmf*[*symmetric*] *spmf.map-comp o-def split-def oracle-intercept-def*) **}**
then show *?thesis*
unfolding *game2-def PRF.game-1-def key-gen-def reduction-prf-def*
by (*clarsimp simp add*: *exec-gpv-bind-lift-spmf exec-gpv-bind map-spmf-conv-bind-spmf split-def bind-spmf-const prf-key-gen-lossless lossless-weight-spmfD eq-commute*)
qed
ultimately show *?thesis* **by**(*simp add*: *PRF.advantage-def*)
qed

definition *oracle-encrypt3* ::
('*prf-key* × '*upf-key*) ⇒ *bool* ⇒ (*bool* × (*bitstring*, *bitstring*) *PRF.dict*) ⇒
bitstring × *bitstring* ⇒ ('*hash cipher-text option* × (*bool* × (*bitstring*, *bitstring*)

PRF.dict)) *spmf*
where
oracle-encrypt3 = (λ(*k-prf*, *k-upf*) *b* (*bad*, *D*) (*msg1*, *msg0*).
(*case* (*length msg1* = *prf-clen* ∧ *length msg0* = *prf-clen*) *of*
False ⇒ *return-spmf* (*None*, (*bad*, *D*))
| *True* ⇒ *do* {
x ← *spmf-of-set prf-domain*;
P ← *spmf-of-set* (*nlists UNIV prf-clen*);
let (*p*, *F*) = (*case D x of Some r* ⇒ (*P*, *True*) | *None* ⇒ (*P*, *False*));
let c = *p* [⊕] (*if b then msg1 else msg0*);
let t = *upf-fun k-upf* (*x* @ *c*);
return-spmf (*Some* (*x*, *c*, *t*), (*bad* ∨ *F*, *D*(*x* ↦ *p*)))
}))

lemma *lossless-oracle-encrypt3* [*simp*]:
lossless-spmf (*oracle-encrypt3 k b D m10*)
by (*cases m10*) (*simp add*: *oracle-encrypt3-def prf-domain-nonempty prf-domain-finite split-def Let-def split*: *bool.splits*)

lemma *callee-invariant-oracle-encrypt3* [*simp*]: *callee-invariant* (*oracle-encrypt3 key b*) *fst*
by (*unfold-locales*) (*auto simp add*: *oracle-encrypt3-def split-def Let-def split*: *bool.splits*)

definition *game3* :: (*bitstring*, *'hash cipher-text*) *ind-cca.adversary* ⇒ (*bool* × *bool*) *spmf*
where
game3 𝒜 ≡ *do* {
key ← *key-gen*;
b ← *coin-spmf*;
(*b'*, (*bad*, *D*)) ← *exec-gpv* (*oracle-encrypt3 key b* $\oplus_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*);
return-spmf (*b* = *b'*, *bad*)
}

lemma *round-3*:
assumes *lossless-gpv* (ℐ-*full* $\oplus_\mathcal{I}$ ℐ-*full*) 𝒜
shows |*measure* (*measure-spmf* (*game3* 𝒜)) {(*b*, *bad*). *b*} − *spmf* (*game2* 𝒜) *True*|
≤ *measure* (*measure-spmf* (*game3* 𝒜)) {(*b*, *bad*). *bad*}
proof −
def *oracle-encrypt2'* ≡ λ(*k-prf* :: *'prf-key*, *k-upf*) *b* (*bad*, *D*) (*msg1*, *msg0*).
case length msg1 = *prf-clen* ∧ *length msg0* = *prf-clen of*
False ⇒ *return-spmf* (*None*, (*bad*, *D*))
| *True* ⇒ *do* {
x ← *spmf-of-set prf-domain*;
P ← *spmf-of-set* (*nlists UNIV prf-clen*);
let (*p*, *F*) = (*case D x of Some r* ⇒ (*r*, *True*) | *None* ⇒ (*P*, *False*));
let c = *p* [⊕] (*if b then msg1 else msg0*);

let t = upf-fun k-upf (x @ c);
return-spmf (Some (x, c, t), (bad ∨ F, D(x ↦ p)))
}

have [*simp*]: *lossless-spmf (oracle-encrypt2′ key b D msg10)* **for** *key b D msg10*
by (*cases msg10*) (*simp add*: *oracle-encrypt2′-def prf-domain-nonempty prf-domain-finite split-def Let-def split*: *bool.split*)
have [*simp*]: *callee-invariant (oracle-encrypt2′ key b) fst* **for** *key b*
by (*unfold-locales*) (*auto simp add*: *oracle-encrypt2′-def split-def Let-def split*: *bool.splits*)

def *game2′* ≡ λ𝒜. *do* {
key ← key-gen;
b ← coin-spmf;
(*b′*, (*bad*, *D*)) ← *exec-gpv* (*oracle-encrypt2′ key b* ⊕$_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*);
return-spmf (*b* = *b′*, *bad*)}

have *game2′-eq*: *game2* 𝒜 = *map-spmf fst* (*game2′* 𝒜)
proof −
def *S* ≡ λ(*D1* :: (*bitstring*, *bitstring*) *PRF.dict*) (*bad* :: *bool*, *D2*). *D1* = *D2*
have [*transfer-rule*, *simp*]: *S Map-empty* (*b*, *Map-empty*) **for** *b* **by** (*simp add*: *S-def*)

have [*transfer-rule*]: (*op* = ===> *op* = ===> *S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *S*))
oracle-encrypt2 oracle-encrypt2′
unfolding *oracle-encrypt2-def*[*abs-def*] *oracle-encrypt2′-def*[*abs-def*]
by (*auto simp add*: *rel-fun-def Let-def split-def S-def*
intro!: *rel-spmf-bind-reflI split*: *bool.split option.split*)
have [*transfer-rule*]: (*op* = ===> *S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *S*))
oracle-decrypt2 oracle-decrypt2
by(*auto simp add*: *rel-fun-def oracle-decrypt2-def*)

show *?thesis* **unfolding** *game2-def game2′-def*
by (*simp add*: *map-spmf-bind-spmf o-def split-def Map-empty-def*[*symmetric*] *del*: *Map-empty-def*)
transfer-prover
qed
moreover have ∗: *rel-spmf* (λ(*b′1*, *bad1*, *L1*) (*b′2*, *bad2*, *L2*). (*bad1* ⟷ *bad2*) ∧ (¬ *bad2* ⟶ *b′1* ⟷ *b′2*))
(*exec-gpv* (*oracle-encrypt3 key b* ⊕$_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*))
(*exec-gpv* (*oracle-encrypt2′ key b* ⊕$_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*))
for *key b*
apply(*rule exec-gpv-oracle-bisim-bad*[**where** *X*=*op* = **and** *X-bad* = λ- -. *True* **and** *?bad1.0*=*fst* **and** *?bad2.0*=*fst* **and** ℐ=ℐ*-full* ⊕$_{\mathcal{I}}$ ℐ*-full*])
apply(*simp-all add*: *assms*)
apply(*auto simp add*: *assms spmf-rel-map Let-def oracle-encrypt2′-def oracle-encrypt3-def*

split: *plus-oracle-split prod.split bool.split option.split intro*!: *rel-spmf-bind-reflI rel-spmf-reflI*)
done
have |*measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *b*} − *measure* (*measure-spmf* (*game2'* $\mathcal{A}$)) {(*b*, *bad*). *b*}| ≤
measure (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *bad*}
unfolding *game2'-def game3-def*
by(*rule fundamental-lemma*[**where** *?bad2.0=snd*])(*intro rel-spmf-bind-reflI rel-spmf-bindI*[*OF* ∗]; *clarsimp*)
ultimately show *?thesis* **by**(*simp add*: *spmf-conv-measure-spmf measure-map-spmf vimage-def fst-def*)
qed

lemma *round-4*:
assumes *lossless-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) $\mathcal{A}$
shows *map-spmf fst* (*game3* $\mathcal{A}$) = *coin-spmf*
proof −
def *oracle-encrypt4* ≡ λ(*k-prf* :: *'prf-key*, *k-upf*) (*s* :: *unit*) (*msg1* :: *bitstring*, *msg0* :: *bitstring*).
case length msg1 = *prf-clen* ∧ *length msg0* = *prf-clen of*
False ⇒ *return-spmf* (*None*, *s*)
| *True* ⇒ *do* {
x ← *spmf-of-set prf-domain*;
P ← *spmf-of-set* (*nlists UNIV prf-clen*);
let c = *P*;
let t = *upf-fun k-upf* (*x* @ *c*);
return-spmf (*Some* (*x*, *c*, *t*), *s*) }

have [*simp*]: *lossless-spmf* (*oracle-encrypt4 k s msg10*) **for** *k s msg10*
by (*cases msg10*) (*simp add*: *oracle-encrypt4-def prf-domain-finite prf-domain-nonempty split-def Let-def split*: *bool.splits*)

def *game4* ≡ λ$\mathcal{A}$. *do* {
key ← *key-gen*;
(*b'*, -) ← *exec-gpv* (*oracle-encrypt4 key* $\oplus_O$ *oracle-decrypt2 key*) $\mathcal{A}$ ();
map-spmf (*op* = *b'*) *coin-spmf*}

have *map-spmf fst* (*game3* $\mathcal{A}$) = *game4* $\mathcal{A}$
proof −
note [*split del*] = *if-split*
def *S* ≡ λ(- :: *unit*) (- :: *bool* × (*bitstring*, *bitstring*) *PRF.dict*). *True*
def *initial3* ≡ (*False*, *Map.empty* :: (*bitstring*, *bitstring*) *PRF.dict*)
have [*transfer-rule*]: *S* () *initial3* **by**(*simp add*: *S-def*)
have [*transfer-rule*]: (*op* = ===> *op* = ===> *S* ===> *op* = ===> *rel-spmf* (*rel-prod op* = *S*))
(λ*key b*. *oracle-encrypt4 key*) *oracle-encrypt3*
proof(*intro rel-funI*; *hypsubst*)
fix *key unit msg10 b Dbad*
have *map-spmf fst* (*oracle-encrypt4 key* () *msg10*) = *map-spmf fst* (*oracle-encrypt3 key b Dbad msg10*)

unfolding *oracle-encrypt3-def oracle-encrypt4-def*
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf Let-def split*: *bool.split prod.split*; *rule conjI*; *clarsimp*)
apply (*rewrite* **in** ⌑ = - *one-time-pad*[*symmetric*, **where** *xs*=*if b then fst msg10 else snd msg10*])
apply(*simp split*: *if-split*)
apply(*simp add*: *bind-map-spmf o-def option.case-distrib case-option-collapse xor-list-commute split-def cong del*: *option.case-cong-weak if-weak-cong*)
done
then show *rel-spmf* (*rel-prod op* = *S*) (*oracle-encrypt4 key unit msg10*) (*oracle-encrypt3 key b Dbad msg10*)
by(*auto simp add*: *spmf-rel-eq*[*symmetric*] *spmf-rel-map S-def elim*: *rel-spmf-mono*)
qed

show *?thesis*
unfolding *game3-def game4-def* **including** *monad-normalisation*
by (*simp add*: *map-spmf-bind-spmf o-def split-def map-spmf-conv-bind-spmf initial3-def*[*symmetric*] *eq-commute*)
transfer-prover
qed
also have ... = *coin-spmf*
by(*simp add*: *map-eq-const-coin-spmf game4-def bind-spmf-const split-def lossless-exec-gpv*[*OF assms*] *lossless-weight-spmfD*)
finally show *?thesis* **.**
qed

lemma *game3-bad*:
assumes *interaction-bounded-by isl* $\mathcal{A}$ *q*
shows *measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *bad*} ≤ *q* / *card prf-domain* ∗ *q*
proof −
have *measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *bad*} = *spmf* (*map-spmf snd* (*game3* $\mathcal{A}$)) *True*
by (*simp add*: *spmf-conv-measure-spmf measure-map-spmf vimage-def snd-def*)
also
have *spmf* (*map-spmf* (*fst* ∘ *snd*) (*exec-gpv* (*oracle-encrypt3 k b* $\oplus_O$ *oracle-decrypt2 k*) $\mathcal{A}$ (*False*, *Map.empty*))) *True* ≤ *q* / *card prf-domain* ∗ *q*
(**is** *spmf* (*map-spmf* - (*exec-gpv ?oracle* - -)) - ≤ -)
if *k*: *k* ∈ *set-spmf key-gen* **for** *k b*
proof(*rule callee-invariant-on.interaction-bounded-by'-exec-gpv-bad-count*)
obtain *k-prf k-upf* **where** *k*: *k* = (*k-prf*, *k-upf*) **by**(*cases k*)
let *?I* = λ(*bad*, *D*). *finite* (*dom D*) ∧ *dom D* ⊆ *prf-domain*
have *callee-invariant* (*oracle-encrypt3 k b*) *?I*
by *unfold-locales*(*clarsimp simp add*: *prf-domain-finite oracle-encrypt3-def Let-def split-def split*: *bool.splits*)+
moreover have *callee-invariant* (*oracle-decrypt2 k*) *?I*
by *unfold-locales* (*clarsimp simp add*: *prf-domain-finite oracle-decrypt2-def*)+
ultimately show *callee-invariant ?oracle ?I* **by** *simp*

let *?count* = λ(*bad*, *D*). *card* (*dom D*)
show $\bigwedge$*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*?oracle s x*); *?I s*; *isl x* ⟧ ⟹ *?count s'* ≤ *Suc* (*?count s*)
by(*clarsimp simp add*: *isl-def oracle-encrypt3-def split-def Let-def card-insert-if split*: *bool.splits*)
show ⟦ (*y*, *s'*) ∈ *set-spmf* (*?oracle s x*); *?I s*; ¬ *isl x* ⟧ ⟹ *?count s'* ≤ *?count s* **for** *s x y s'*
by(*cases x*)(*simp-all add*: *oracle-decrypt2-def*)
show *spmf* (*map-spmf* (*fst* ∘ *snd*) (*?oracle s' x*)) *True* ≤ *q* / *card prf-domain*
if *I*: *?I s'* **and** *bad*: ¬ *fst s'* **and** *count*: *?count s'* < *q* + *?count* (*False*, *Map.empty*)
and *x*: *isl x*
for *s' x*
proof −
obtain *bad D* **where** *s'* [*simp*]: *s'* = (*bad*, *D*) **by**(*cases s'*)
from *x* **obtain** *m1 m0* **where** *x* [*simp*]: *x* = *Inl* (*m1*, *m0*) **by**(*auto elim*: *islE*)
have ∗: (*case D x of None* ⇒ *False* | *Some x* ⇒ *True*) ⟷ *x* ∈ *dom D* **for** *x*
by(*auto split*: *option.split*)
show *?thesis*
proof(*cases length m1* = *prf-clen* ∧ *length m0* = *prf-clen*)
case *True*
with *bad*
have *spmf* (*map-spmf* (*fst* ∘ *snd*) (*?oracle s' x*)) *True* = *pmf* (*bernoulli-pmf* (*card* (*dom D* ∩ *prf-domain*) / *card prf-domain*)) *True*
by(*simp add*: *spmf.map-comp o-def oracle-encrypt3-def k* ∗ *bool.case-distrib*[**where** *h*=λ*p*. *spmf* (*map-spmf* - *p*) -] *option.case-distrib*[**where** *h*=*snd*] *map-spmf-bind-spmf Let-def split-beta bind-spmf-const cong*: *bool.case-cong option.case-cong split del*: *if-split split*: *bool.split*)
(*simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *map-mem-spmf-of-set prf-domain-finite prf-domain-nonempty*)
also have ... = *card* (*dom D* ∩ *prf-domain*) / *card prf-domain*
by(*rule pmf-bernoulli-True*)(*auto simp add*: *field-simps prf-domain-finite prf-domain-nonempty card-gt-0-iff card-mono*)
also have *dom D* ∩ *prf-domain* = *dom D* **using** *I* **by** *auto*
also have *card* (*dom D*) ≤ *q* **using** *count* **by** *simp*
finally show *?thesis* **by**(*simp add*: *divide-right-mono o-def*)
next
case *False*
thus *?thesis* **using** *bad*
by(*auto simp add*: *spmf.map-comp o-def oracle-encrypt3-def k split*: *bool.split*)
qed
qed
qed(*auto split*: *plus-oracle-split-asm simp add*: *oracle-decrypt2-def assms*)
then have *spmf* (*map-spmf snd* (*game3* $\mathcal{A}$)) *True* ≤ *q* / *card prf-domain* ∗ *q*
by(*auto 4 3 simp add*: *game3-def map-spmf-bind-spmf o-def split-def map-spmf-conv-bind-spmf intro*: *spmf-bind-leI*)
finally show *?thesis* **.**

qed

theorem *security*:
  assumes *lossless*: *lossless-gpv* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\mathcal{A}$
  and *bound*: *interaction-bounded-by isl* $\mathcal{A}$ *q*
  shows *ind-cca.advantage* $\mathcal{A}$ $\leq$
    PRF.advantage (*reduction-prf* $\mathcal{A}$) + *UPF.advantage* (*reduction-upf* $\mathcal{A}$) +
    real q / *real* (*card prf-domain*) $*$ *real q* (**is** *?LHS* $\leq$ -)
proof –
  have *?LHS* $\leq$ |*spmf* (*ind-cca.game* $\mathcal{A}$) *True* – *spmf* (*ind-cca'.game* $\mathcal{A}$) *True*| +
|*spmf* (*ind-cca'.game* $\mathcal{A}$) *True* – *1* / *2*|
   (**is** - $\leq$ *?round1* + *?rest*) **using** *abs-triangle-ineq* **by**(*simp add*: *ind-cca.advantage-def*)
  also have *?round1* $\leq$ *UPF.advantage* (*reduction-upf* $\mathcal{A}$)
    using *lossless* **by**(*rule round-1*)
  also have *?rest* $\leq$ |*spmf* (*ind-cca'.game* $\mathcal{A}$) *True* – *spmf* (*game2* $\mathcal{A}$) *True*| +
|*spmf* (*game2* $\mathcal{A}$) *True* – *1* / *2*|
    (**is** - $\leq$ *?round2* + *?rest*) **using** *abs-triangle-ineq* **by** *simp*
  also have *?round2* = *PRF.advantage* (*reduction-prf* $\mathcal{A}$) **by**(*rule round-2*)
  also have *?rest* $\leq$ |*measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *b*} – *spmf* (*game2* $\mathcal{A}$) *True*| +
      |*measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *b*} – *1* / *2*|
    (**is** - $\leq$ *?round3* + -) **using** *abs-triangle-ineq* **by** *simp*
  also have *?round3* $\leq$ *measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *bad*}
    using *round-3*[*OF lossless*] **.**
  also have ... $\leq$ *q* / *card prf-domain* $*$ *q* **using** *bound* **by**(*rule game3-bad*)
  also have *measure* (*measure-spmf* (*game3* $\mathcal{A}$)) {(*b*, *bad*). *b*} = *spmf coin-spmf True*
    using *round-4*[*OF lossless*, *symmetric*]
    by(*simp add*: *spmf-conv-measure-spmf measure-map-spmf vimage-def fst-def*)
  also have |... – *1* / *2*| = *0* **by**(*simp add*: *spmf-of-set*)
  finally show *?thesis* **by**(*simp*)
qed

theorem *security1*:
  assumes *lossless*: *lossless-gpv* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\mathcal{A}$
  assumes *q*: *interaction-bounded-by isl* $\mathcal{A}$ *q*
  and *q'*: *interaction-bounded-by* (*Not* $\circ$ *isl*) $\mathcal{A}$ *q'*
  shows *ind-cca.advantage* $\mathcal{A}$ $\leq$
    PRF.advantage (*reduction-prf* $\mathcal{A}$) +
    UPF.advantage1 (*guessing-many-one.reduction q'* (λ-. *reduction-upf* $\mathcal{A}$) ()) $*$
q' +
    real q $*$ *real q* / *real* (*card prf-domain*)
proof –
  have *ind-cca.advantage* $\mathcal{A}$ $\leq$
    PRF.advantage (*reduction-prf* $\mathcal{A}$) + *UPF.advantage* (*reduction-upf* $\mathcal{A}$) +
    real q / *real* (*card prf-domain*) $*$ *real q*
    using *lossless q* **by**(*rule security*)
  also note *q'*[*interaction-bound*]

have *interaction-bounded-by* (*Not* ∘ *isl*) (*reduction-upf* $\mathcal{A}$) *q'*
unfolding *reduction-upf-def* **by**(*interaction-bound*)(*simp-all add*: *SUP-le-iff*)
then have *UPF.advantage* (*reduction-upf* $\mathcal{A}$) ≤ *UPF.advantage1* (*guessing-many-one.reduction* *q'* (λ-. *reduction-upf* $\mathcal{A}$) ()) ∗ *q'*
by(*rule UPF.advantage-advantage1*)
finally show *?thesis* **by**(*simp*)
qed

end

end

locale *simple-cipher'* =
fixes *prf-key-gen* :: *security* ⇒ *'prf-key spmf*
and *prf-fun* :: *security* ⇒ *'prf-key* ⇒ *bitstring* ⇒ *bitstring*
and *prf-domain* :: *security* ⇒ *bitstring set*
and *prf-range* :: *security* ⇒ *bitstring set*
and *prf-dlen* :: *security* ⇒ *nat*
and *prf-clen* :: *security* ⇒ *nat*
and *upf-key-gen* :: *security* ⇒ *'upf-key spmf*
and *upf-fun* :: *security* ⇒ *'upf-key* ⇒ *bitstring* ⇒ *'hash*
assumes *simple-cipher*: ⋀η. *simple-cipher* (*prf-key-gen* η) (*prf-fun* η) (*prf-domain* η) (*prf-dlen* η) (*prf-clen* η) (*upf-key-gen* η)
begin

sublocale *simple-cipher*
prf-key-gen η *prf-fun* η *prf-domain* η *prf-range* η *prf-dlen* η *prf-clen* η *upf-key-gen* η *upf-fun* η
for η
by(*rule simple-cipher*)

theorem *security-asymptotic*:
fixes *q q'* :: *security* ⇒ *nat*
assumes *lossless*: ⋀η. *lossless-gpv* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) ($\mathcal{A}$ η)
and *bound*: ⋀η. *interaction-bounded-by isl* ($\mathcal{A}$ η) (*q* η)
and *bound'*: ⋀η. *interaction-bounded-by* (*Not* ∘ *isl*) ($\mathcal{A}$ η) (*q'* η)
and [*negligible-intros*]:
polynomial q' polynomial q
negligible (λη. *PRF.advantage* η (*reduction-prf* η ($\mathcal{A}$ η)))
negligible (λη. *UPF.advantage1* η (*guessing-many-one.reduction* (*q'* η) (λ-. *reduction-upf* η ($\mathcal{A}$ η)) ()))
negligible (λη. *1* / *card* (*prf-domain* η))
shows *negligible* (λη. *ind-cca.advantage* η ($\mathcal{A}$ η))
proof −
have *negligible* (λη. *PRF.advantage* η (*reduction-prf* η ($\mathcal{A}$ η)) +
UPF.advantage1 η (*guessing-many-one.reduction* (*q'* η) (λ-. *reduction-upf* η ($\mathcal{A}$ η)) ()) ∗ *q'* η +
real (*q* η) / *real* (*card* (*prf-domain* η)) ∗ *real* (*q* η))
by(*rule negligible-intros*)+

thus *?thesis* **by**(*rule negligible-le*)(*simp add*: *security1*[*OF lossless bound bound'*] *ind-cca.advantage-nonneg*)
qed

end

end

theory *Cryptographic-Constructions* **imports**
Elgamal
Hashed-Elgamal
RP-RF
PRF-UHF
PRF-IND-CPA
PRF-UPF-IND-CCA
begin

end

theory *Game-Based-Crypto* **imports**
Security-Spec
Cryptographic-Constructions
begin

end

References

[1] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In B. Preneel, editor, *Advances in Cryptology (EUROCRYPT 2000)*, volume 1807 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin Heidelberg, 2000.

[2] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, 2006.

[3] A. Petcher and G. Morrisett. The foundational cryptography framework. In *POST 2015*, volume 9036 of *LNCS*, pages 53–72. Springer, 2015.

[4] V. Shoup. Sequences of games: A tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.