Game-based cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar and Bhargav Bhatt

May 26, 2024

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 [19] and Bellare and Rogaway [4], 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 ← *sample-uniform* (*order* $\mathcal{G}$);
    y ← *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 ← *sample-uniform* (*order* $\mathcal{G}$);
    y ← *sample-uniform* (*order* $\mathcal{G}$);
    z ← *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* $\mathcal{A}$ ⟷ (∀ α β γ. *lossless-spmf* ($\mathcal{A}$ α β γ))

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** [^] *x*) (**g** [^] *y*);
return-spmf (**g** [^] (*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 [3].

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* | -

$\Rightarrow 0$)) *auto*
end

1.3.1 Single-user setting

type-synonym ('*plain*', '*cipher*') $call_1$ = *unit* + '*cipher*' + '*plain*' × '*plain*'
type-synonym ('*ekey*', '*plain*', '*cipher*') ret_1 = '*ekey*' + '*plain*' *option* + '*cipher*'

definition $oracle_1$ :: *bool* ⇒ *security*
⇒ (('*ekey*, '*dkey*, '*cipher*) *state-oracle*, ('*plain*, '*cipher*) $call_1$, ('*ekey*, '*plain*, '*cipher*) ret_1)
oracle'
where $oracle_1$ *b* η = *ekey-oracle* η $\oplus_O$ (*decrypt-oracle* η $\oplus_O$ *encrypt-oracle* *b* η)

lemma $oracle_1$*-simps* [*simp*]:
$oracle_1$ *b* η *s* (*Inl x*) = *map-spmf* (*apfst Inl*) (*ekey-oracle* η *s x*)
$oracle_1$ *b* η *s* (*Inr* (*Inl y*)) = *map-spmf* (*apfst* (*Inr* ∘ *Inl*)) (*decrypt-oracle* η *s y*)
$oracle_1$ *b* η *s* (*Inr* (*Inr z*)) = *map-spmf* (*apfst* (*Inr* ∘ *Inr*)) (*encrypt-oracle b* η *s z*)
by(*simp-all add*: $oracle_1$*-def spmf.map-comp apfst-compose o-def*)

type-synonym ('*ekey*', '*plain*', '*cipher*') $adversary_1$' =
(*bool*, ('*plain*', '*cipher*') $call_1$, ('*ekey*', '*plain*', '*cipher*') ret_1) *gpv*
type-synonym ('*ekey*', '*plain*', '*cipher*') $adversary_1$ =
security ⇒ ('*ekey*', '*plain*', '*cipher*') $adversary_1$'

definition *ind-cca2*$_1$ :: ('*ekey*, '*plain*, '*cipher*) $adversary_1$ ⇒ *security* ⇒ *bool spmf*
where
ind-cca2$_1$ $\mathscr{A}$ η = *TRY do* {
b ← *coin-spmf*;
(*guess*, *s*) ← *exec-gpv* ($oracle_1$ *b* η) ($\mathscr{A}$ η) *None*;
return-spmf (*guess* = *b*)
} *ELSE coin-spmf*

definition $advantage_1$ :: ('*ekey*, '*plain*, '*cipher*) $adversary_1$ ⇒ *advantage*
where $advantage_1$ $\mathscr{A}$ η = |*spmf* (*ind-cca2*$_1$ $\mathscr{A}$ η) *True* − *1/2*|

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

abbreviation *secure-for*$_1$ :: ('*ekey*, '*plain*, '*cipher*) $adversary_1$ ⇒ *bool*
where *secure-for*$_1$ $\mathscr{A}$ ≡ *negligible* ($advantage_1$ $\mathscr{A}$)

definition *ibounded-by*$_1$' :: ('*ekey*, '*plain*, '*cipher*) $adversary_1$' ⇒ *nat* ⇒ *bool*
where *ibounded-by*$_1$' $\mathscr{A}$ *q* = *interaction-any-bounded-by* $\mathscr{A}$ *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*$_1$' $\mathscr{A}$ = *lossless-gpv* $\mathscr{I}$*-full* $\mathscr{A}$

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$_n$ $\mathscr{A}$ η = *TRY do* {
b ← *coin-spmf*;
(*guess*, σ) ← *exec-gpv* (*oracle*$_n$ *b* η) ($\mathscr{A}$ η) (λ-. *None*);
return-spmf (*guess* = *b*)
} *ELSE coin-spmf*

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

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

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

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

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

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

abbreviation *lossless*$_n$ :: ('*i*, '*ekey*, '*plain*, '*cipher*) *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 𝒜 = do {
   key ← key-gen;
   b ← coin-spmf;
   (b', L') ← exec-gpv (oracle-encrypt key b ⊕O oracle-decrypt key) 𝒜 {};
   return-spmf (b = b')
  }

definition advantage :: ('message, 'cipher) adversary ⇒ real
where advantage 𝒜 = |spmf (game 𝒜) True − 1 / 2|

lemma advantage-nonneg: 0 ≤ advantage 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 𝒜 = *do* {
let (𝒜*1*, 𝒜*2*) = 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* $\mathscr{A}$ = |*spmf* (*ind-cpa* $\mathscr{A}$) *True* − *1/2*|

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

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

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

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

definition *lossless* :: (*'plain*, *'cipher*, *'state*) *adversary* ⇒ *bool*
where *lossless* = (λ($\mathscr{A}$*1*, $\mathscr{A}$*2*). *lossless-gpv* $\mathscr{I}$*-full* $\mathscr{A}$*1* ∧ (∀*cipher* σ. *lossless-gpv* $\mathscr{I}$*-full* ($\mathscr{A}$*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* × *'privkey*, *'call*, *'ret*) *gpv* — probabilistic
and *aencrypt* :: *'pubkey* ⇒ *'plain* ⇒ (*'cipher*, *'call*, *'ret*) *gpv* — probabilistic w/ access to an oracle
and *adecrypt* :: *'privkey* ⇒ *'cipher* ⇒ (*'plain*, *'call*, *'ret*) *gpv* — not used
and *valid-plains* :: *'plain* ⇒ *'plain* ⇒ *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 $(\mathscr{A}1, \mathscr{A}2)$ = *TRY do* {
$(pk, sk) \leftarrow$ *key-gen*;
$b \leftarrow$ *lift-spmf coin-spmf*;
$((m0, m1), \sigma) \leftarrow (\mathscr{A}1\ pk)$;
assert-gpv (*valid-plains m0 m1*);
cipher $\leftarrow$ *aencrypt pk* (*if b then m0 else m1*);
guess $\leftarrow \mathscr{A}2$ *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* $\sigma\ \mathscr{A} = |spmf\ (run\text{-}gpv\ oracle\ (ind\text{-}cpa\ \mathscr{A})\ \sigma)\ True - 1/2|$

lemma *advantage-nonneg*: *advantage oracle* $\sigma\ \mathscr{A} \geq 0$ **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 $= (\lambda(\mathscr{A}1, \mathscr{A}2)\ q.$
$(\exists q1\ q2.\ (\forall pk.$ *interaction-bounded-by consider* $(\mathscr{A}1\ pk)\ q1) \wedge (\forall cipher\ \sigma.$ *interaction-bounded-by consider* $(\mathscr{A}2\ cipher\ \sigma)\ q2) \wedge q1 + q2 \leq q))$

lemma *ibounded-by'E* [*consumes 1*, *case-names ibounded-by'*, *elim?*]:
assumes *ibounded-by consider* $(\mathscr{A}1, \mathscr{A}2)\ q$
obtains *q1 q2*
where $q1 + q2 \leq q$
and $\bigwedge pk.$ *interaction-bounded-by consider* $(\mathscr{A}1\ pk)\ q1$
and $\bigwedge cipher\ \sigma.$ *interaction-bounded-by consider* $(\mathscr{A}2\ cipher\ \sigma)\ q2$
using *assms* **by**(*auto simp add*: *ibounded-by-def*)

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

definition *lossless* :: $('pubkey, 'plain, 'cipher, 'call, 'ret, 'state)$ *adversary* $\Rightarrow$ *bool*

where *lossless* = (λ(𝒜*1*, 𝒜*2*). (∀*pk*. *lossless-gpv* ℐ*-full* (𝒜*1 pk*)) ∧ (∀*cipher* σ. *lossless-gpv* ℐ*-full* (𝒜*2 cipher* σ)))

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-key* × *'priv-key*) *spmf* — probabilistic
and *aencrypt* :: *'pub-key* ⇒ *'plain* ⇒ *'cipher spmf* — probabilistic
and *adecrypt* :: *'priv-key* ⇒ *'cipher* ⇒ *'plain option* — deterministic, but not used
and *valid-plains* :: *'plain* ⇒ *'plain* ⇒ *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-key'*, *'plain'*, *'cipher'*, *'state*) *adversary* =
(*'pub-key'* ⇒ ((*'plain'* × *'plain'*) × *'state*) *spmf*)
× (*'cipher'* ⇒ *'state* ⇒ *bool spmf*)

primrec *ind-cpa* :: (*'pub-key*, *'plain*, *'cipher*, *'state*) *adversary* ⇒ *bool spmf*
where
ind-cpa (𝒜*1*, 𝒜*2*) = *TRY do* {
(*pk*, *sk*) ← *key-gen*;
((*m0*, *m1*), σ) ← 𝒜*1 pk*;
- :: *unit* ← *assert-spmf* (*valid-plains m0 m1*);
b ← *coin-spmf*;
cipher ← *aencrypt pk* (*if b then m0 else m1*);
b' ← 𝒜*2 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* 𝒜 = |*spmf* (*ind-cpa* 𝒜) *True* − *1/2*|

definition *lossless* :: (*'pub-key*, *'plain*, *'cipher*, *'state*) *adversary* ⇒ *bool*
where

lossless $\mathscr{A}$ $\longleftrightarrow$
((∀*pk*. *lossless-spmf* (*fst* $\mathscr{A}$ *pk*)) ∧
(∀*cipher* σ. *lossless-spmf* (*snd* $\mathscr{A}$ *cipher* σ)))

lemma *lossless-ind-cpa*:
⟦ *lossless* $\mathscr{A}$; *lossless-spmf* (*key-gen*) ⟧ ⟹ *lossless-spmf* (*ind-cpa* $\mathscr{A}$)
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-oracle*, '*message call*$_1$, ('*vkey*, '*signature*) *ret*$_1$) *oracle'*
where *oracle*$_1$ η = *vkey-oracle* η $\oplus_O$ *sign-oracle* η

lemma *oracle*$_1$*-simps* [*simp*]:
oracle$_1$ η *s* (*Inl x*) = *map-spmf* (*apfst Inl*) (*vkey-oracle* η *s x*)
oracle$_1$ η *s* (*Inr y*) = *map-spmf* (*apfst Inr*) (*sign-oracle* η *s y*)
by(*simp-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-cma*$_1$ :: $('vkey, 'message, 'signature)$ *adversary*$_1$ ⇒ *security* ⇒ *bool spmf*
where
$\bigwedge log.$ *suf-cma*$_1$ $\mathscr{A}$ η = *do* {
$((m, sig), \sigma) \leftarrow$ *exec-gpv* (*oracle*$_1$ η) ($\mathscr{A}$ η) *None*;
return-spmf (
case σ *of None* ⇒ *False*
| *Some* (*vkey*, *skey*, *log*) ⇒ *verify* η *vkey m sig* ∧ $(m, sig) \notin$ *set log*)
}

definition *advantage*$_1$ :: $('vkey, 'message, 'signature)$ *adversary*$_1$ ⇒ *advantage*
where *advantage*$_1$ $\mathscr{A}$ η = *spmf* (*suf-cma*$_1$ $\mathscr{A}$ η) *True*

lemma *advantage*$_1$*-nonneg*: *advantage*$_1$ $\mathscr{A}$ $\eta \geq 0$ **by**(*simp add*: *advantage*$_1$*-def pmf-nonneg*)

abbreviation *secure-for*$_1$ :: $('vkey, 'message, 'signature)$ *adversary*$_1$ ⇒ *bool*
where *secure-for*$_1$ $\mathscr{A}$ ≡ *negligible* (*advantage*$_1$ $\mathscr{A}$)

definition *ibounded-by*$_1{}'$ :: $('vkey, 'message, 'signature)$ *adversary*$_1{}'$ ⇒ *nat* ⇒ *bool*
where *ibounded-by*$_1{}'$ $\mathscr{A}$ *q* = (*interaction-any-bounded-by* $\mathscr{A}$ *q*)

abbreviation *ibounded-by*$_1$ :: $('vkey, 'message, 'signature)$ *adversary*$_1$ ⇒ (*security* ⇒ *nat*)
⇒ *bool*
where *ibounded-by*$_1$ ≡ *rel-envir ibounded-by*$_1{}'$

definition *lossless*$_1{}'$ :: $('vkey, 'message, 'signature)$ *adversary*$_1{}'$ ⇒ *bool*
where *lossless*$_1{}'$ $\mathscr{A}$ = (*lossless-gpv* $\mathscr{I}$*-full* $\mathscr{A}$)

abbreviation *lossless*$_1$ :: $('vkey, 'message, 'signature)$ *adversary*$_1$ ⇒ *bool*
where *lossless*$_1$ ≡ *pred-envir lossless*$_1{}'$

1.8.2 Multi-user setting

definition *oracle*$_n$ :: *security*
⇒ $('i \Rightarrow ('vkey, 'sigkey, 'message, 'signature)$ *state-oracle*, $'i \times 'message$ *call*$_1$, $('vkey,$
$'signature)$ *ret*$_1$) *oracle'*
where *oracle*$_n$ η = *family-oracle* (λ-. *oracle*$_1$ η)

lemma *oracle*$_n$*-apply* [*simp*]:
oracle$_n$ η *s* (*i*, *x*) = *map-spmf* (*apsnd* (*fun-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 ⇒ $('i, 'vkey', 'message', 'signature')$ *adversary*$_n{}'$

definition *suf-cma*$_n$ :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$ ⇒ *security* ⇒ *bool spmf*
where
⋀*log*. *suf-cma*$_n$ $\mathscr{A}$ η = *do* {
((*i*, *m*, *sig*), σ) ← *exec-gpv* (*oracle*$_n$ η) ($\mathscr{A}$ η) (λ-. *None*);
return-spmf (
case σ *i of None* ⇒ *False*
| *Some* (*vkey*, *skey*, *log*) ⇒ *verify* η *vkey m sig* ∧ (*m*, *sig*) ∉ *set log*)
}

definition *advantage*$_n$ :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$ ⇒ *advantage*
where *advantage*$_n$ $\mathscr{A}$ η = *spmf* (*suf-cma*$_n$ $\mathscr{A}$ η) *True*

lemma *advantage*$_n$*-nonneg*: *advantage*$_n$ $\mathscr{A}$ η ≥ *0* **by**(*simp add*: *advantage*$_n$*-def pmf-nonneg*)

abbreviation *secure-for*$_n$ :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$ ⇒ *bool*
where *secure-for*$_n$ $\mathscr{A}$ ≡ *negligible* (*advantage*$_n$ $\mathscr{A}$)

definition *ibounded-by*$_n$' :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$' ⇒ *nat* ⇒ *bool*
where *ibounded-by*$_n$' $\mathscr{A}$ *q* = (*interaction-any-bounded-by* $\mathscr{A}$ *q*)

abbreviation *ibounded-by*$_n$ :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$ ⇒ (*security* ⇒ *nat*) ⇒ *bool*
where *ibounded-by*$_n$ ≡ *rel-envir ibounded-by*$_n$'

definition *lossless*$_n$' :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$' ⇒ *bool*
where *lossless*$_n$' $\mathscr{A}$ = (*lossless-gpv* $\mathscr{I}$*-full* $\mathscr{A}$)

abbreviation *lossless*$_n$ :: ('*i*, '*vkey*, '*message*, '*signature*) *adversary*$_n$ ⇒ *bool*
where *lossless*$_n$ ≡ *pred-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* ⇀ '*a*'

definition *random-oracle* :: ('*b*, '*a*) *dict* ⇒ '*b* ⇒ ('*a* × ('*b*, '*a*) *dict*) *spmf*
where

random-oracle σ x =
(*case σ x of Some y* ⇒ *return-spmf* (*y*, σ)
| *None* ⇒ *p* ⪢= (λ*y*. *return-spmf* (*y*, σ(*x* ↦ *y*))))

definition *forgetful-random-oracle* :: *unit* ⇒ '*b* ⇒ ('*a* × *unit*) *spmf*
where
forgetful-random-oracle σ x = *p* ⪢= (λ*y*. *return-spmf* (*y*, ()))

lemma *weight-random-oracle* [*simp*]:
weight-spmf p = *1* ⟹ *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 ⟹ *lossless-spmf* (*random-oracle σ x*)
by(*simp add*: *lossless-spmf-def*)

sublocale *finite*: *callee-invariant-on random-oracle* λσ. *finite* (*dom* σ) ℐ-*full*
by(*unfold-locales*)(*auto simp add*: *random-oracle-def split*: *option.splits*)

lemma *card-dom-random-oracle*:
assumes *interaction-any-bounded-by* 𝒜 *q*
and (*y*, σ') ∈ *set-spmf* (*exec-gpv random-oracle* 𝒜 σ)
and *fin*: *finite* (*dom* σ)
shows *card* (*dom* σ') ≤ *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 𝒜 = *do* {
key ← *key-gen*;

$(b, \text{-}) \leftarrow$ *exec-gpv* (*prf-oracle key*) $\mathscr{A}$ ();
return-spmf b
}

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

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

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

abbreviation *lossless* :: ('*domain*, '*range*) *adversary* ⇒ *bool*
where *lossless* ≡ *lossless-gpv* $\mathscr{I}$*-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* ⇀ '*b*) ⇒ '*a* ⇒ ('*b* × ('*a* ⇀ '*b*)) *spmf*
where
random-permutation σ *x* =
(*case* σ *x of Some y* ⇒ *return-spmf* (*y*, σ)
| *None* ⇒ *spmf-of-set* (*A* − *ran* σ) ⤜ (λ*y*. *return-spmf* (*y*, σ(*x* ↦ *y*))))

lemma *weight-random-oracle* [*simp*]:
⟦ *finite A*; *A* − *ran* σ ≠ {} ⟧ ⟹ *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* σ ≠ {} ⟧ ⟹ *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*, σ') ∈ *set-spmf* (*exec-gpv random-permutation* $\mathcal{A}$ σ)
  and *fin*: *finite* (*dom* σ)
  shows *card* (*dom* σ') ≤ *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-o* × *'c-a* × *'s*) *spmf*
  and *oracle* :: *'c-o* ⇒ *'s* ⇒ *'call* ⇒ (*'ret* × *'s*) *spmf*
  and *eval* :: *'c-o* ⇒ *'c-a* ⇒ *'s* ⇒ *'guess* ⇒ *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 𝒜 = *do* {
  (*c-o*, *c-a*, *s*) ← *init*;
  (-, (*b*, -)) ← *exec-gpv*
    (†(*oracle c-o*) $\oplus_O$ *eval-oracle c-o c-a*)
    (𝒜 *c-a*)
    (*False*, *s*);
  return-spmf b
}

definition *advantage-multi* :: (*'c-a*, *'guess*, *'call*, *'ret*) *adversary-many* ⇒ *real*
where *advantage-multi* 𝒜 = *spmf* (*game-multi* 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 𝒜 *c-a* = *do* {
  j-star ← *lift-spmf* (*spmf-of-set* {..<*q*});
  (-, *s*) ← *inline-stop reduction-oracle* (𝒜 *c-a*) (*Inr j-star*);

Done (*projl s*)
}

lemma *many-single-reduction*:
assumes *bound*: $\bigwedge$*c-a c-o s*. (*c-o*, *c-a*, *s*) $\in$ *set-spmf init* $\Longrightarrow$ *interaction-bounded-by* (*Not* $\circ$ *isl*) ($\mathscr{A}$ *c-a*) *q*
and *lossless-oracle*: $\bigwedge$*c-a c-o s s' x*. (*c-o*, *c-a*, *s*) $\in$ *set-spmf init* $\Longrightarrow$ *lossless-spmf* (*oracle c-o s' x*)
and *lossless-eval*: $\bigwedge$*c-a c-o s s' guess*. (*c-o*, *c-a*, *s*) $\in$ *set-spmf init* $\Longrightarrow$ *lossless-spmf* (*eval c-o c-a s' guess*)
shows *advantage-multi* $\mathscr{A}$ $\leq$ *advantage-single* (*reduction q* $\mathscr{A}$) $*$ *q*
including *lifting-syntax*
proof −
define *eval-oracle'*
where *eval-oracle'* = (λ*c-o c-a* ((*id*, *occ* :: *nat option*), *s'*) *guess*.
map-spmf (λ*b'*. *case occ of Some* j_0 $\Rightarrow$ ((), (*Suc id*, *Some* j_0), *s'*)
| *None* $\Rightarrow$ ((), (*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* ($\dagger$(*oracle c-o*) $\oplus_O$ *eval-oracle' c-o c-a*) ($\mathscr{A}$ *c-a*) ((*0*, *None*), *s*)
define *game-multi'* **where** *game-multi'* = (λ*c-o c-a s*. *do* {
(-, ((*id*, j_0), *s'* :: *'s*)) $\leftarrow$ *?multi'-body c-o c-a s*;
return-spmf ($j_0 \neq$ *None*) })

define *initialize* :: (*'c-o* $\Rightarrow$ *'c-a* $\Rightarrow$ *'s* $\Rightarrow$ *nat* $\Rightarrow$ *bool spmf*) $\Rightarrow$ *bool spmf* **where**
initialize body = *do* {
(*c-o*, *c-a*, *s*) $\leftarrow$ *init*;
j_s $\leftarrow$ *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'*) $\leftarrow$ *?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*

define *stop-oracle* **where** *stop-oracle* = (λ*c-o*.
(λ(*idgs*, *s*) *x*. *case idgs of Inr* - $\Rightarrow$ *map-spmf* (λ(*y*, *s*). (*Some y*, (*idgs*, *s*))) (*oracle c-o s x*) | *Inl* - $\Rightarrow$ *return-spmf* (*None*, (*idgs*, *s*)))
$\oplus_O{}^S$
(λ(*idgs*, *s*) *guess* :: *'guess*. *return-spmf* (*case idgs of Inr 0* $\Rightarrow$ (*None*, *Inl* (*guess*, *s*), *s*) | *Inr* (*Suc i*) $\Rightarrow$ (*Some* (), *Inr i*, *s*) | *Inl* - $\Rightarrow$ (*None*, *idgs*, *s*))))
define *body3* **where** *body3 c-o c-a s* j_s = *do* {
(- :: *unit option*, *idgs*, -) $\leftarrow$ *exec-gpv-stop* (*stop-oracle c-o*) ($\mathscr{A}$ *c-a*) (*Inr* j_s, *s*);
(*b'* :: *bool*) $\leftarrow$ *case idgs of Inr* - $\Rightarrow$ *return-spmf False* | *Inl* (*g*, *s'*) $\Rightarrow$ *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* $\Rightarrow$ *nat* $\times$ *nat option* $\Rightarrow$ *bool* **where** *S* $\equiv$ λ*b'* (*id*, *occ*). *b'* $\longleftrightarrow$ ($\exists j_0$. *occ* = *Some* j_0)
let *?S* = *rel-prod S* (=)

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* ===> (=)) (*λb*. *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*]:
((=) ===> (=) ===> *?S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *?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* $\mathscr{A}$ = *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*) ($\mathscr{A}$ *c-a*) *q* **using** *bound that* **by** *blast*

have *bound-occ*: j_s < *q* **if** *that*: ((), (*id*, *Some* j_s), *s′*) ∈ *set-spmf* (*?multi′-body c-o c-a s*)
for *s′ id* j_s
proof −
have *id* ≤ *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* = *λ*((*id*, *occ*), *s′*). *case occ of None* ⇒ *True* | *Some* j_s ⇒ j_s < *id*
have *callee-invariant* (†(*oracle c-o*) ⊕$_O$ *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 ≠ *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 ≠ *None*} =
{((), (*id*, *Some* j_s), *s′*) |j_s *s′ id*. j_s < *q*} ∪ {((), (*id*, *Some* j_s), *s′*) |j_s *s′ id*. j_s ≥ *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*)))
$\{(j_s, (), (id, j_0), s') \mid j_s\, j_0\, s'\, id.\ j_0 = \mathit{Some}\ j_s\} * q$
(**is** - = *measure ?M′ ?B* ∗ -)
proof −
have $?B = \{(j_s, (), (id, j_0), s') \mid j_s\, j_0\, s'\, id.\ j_0 = \mathit{Some}\ j_s \wedge j_s < q\} \cup$
$\{(j_s, (), (id, j_0), s') \mid j_s\, j_0\, s'\, id.\ j_0 = \mathit{Some}\ j_s \wedge j_s \geq q\}$ (**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 ... = $(\sum j \in \{0..<q\}.\ \mathit{measure}\ ?M'\ (\{j\} \times \{((), (id, \mathit{Some}\ j), s') \mid s'\ id.\ \mathit{True}\}))$
by(*subst measure-spmf.finite-measure-finite-Union*[*symmetric*])
(*auto intro*!: *arg-cong2*[**where** *f*=*measure*] *simp add*: *disjoint-family-on-def*)
also have ... = $(\sum j \in \{0..<q\}.\ 1\ /\ q * \mathit{measure}\ (\mathit{measure\text{-}spmf}\ (?multi'\text{-}body\ c\text{-}o\ c\text{-}a\ s))\ \{((), (id, \mathit{Some}\ j), s') \mid s'\ id.\ \mathit{True}\})$
by(*simp add*: *measure-pair-spmf-times spmf-conv-measure-spmf*[*symmetric*] *spmf-of-set*)
also have ... = $1\ /\ q * \mathit{measure}\ (\mathit{measure\text{-}spmf}\ (?multi'\text{-}body\ c\text{-}o\ c\text{-}a\ s))\ \{((), (id, \mathit{Some}\ j_s), s') \mid j_s\ s'\ id.\ j_s < q\}$
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 $?B = (\lambda(j_s, \text{-}, (\text{-}, j_0), \text{-}).\ j_0 = \mathit{Some}\ j_s)\ -`\ \{\mathit{True}\}$
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* (⟶) (*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: $j_s < \mathit{Suc}\ q$ **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* (=) *?R*)
(*exec-gpv* (*extend-state-oracle* (*oracle c-o*) $\oplus_O$ *eval-oracle*′ *c-o c-a*) (𝒜 *c-a*) ((*0*, *None*), *s*))
(*exec-gpv* (*extend-state-oracle* (*extend-state-oracle* (*oracle c-o*)) $\oplus_O$ *oracle2*′) (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* (⟶) (*body2 c-o c-a s* j_s)
(*do* {
(-, *b*′, -, -) ← *exec-gpv* (††(*oracle c-o*) $\oplus_O$ *oracle2*′) (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* 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*′)
})
(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*) (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*)) ($\mathscr{A}$ *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* (⟶) ... (*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*)))
($\mathscr{A}$ *c-a*) ((*0*, *None*), *s*) ≫=
(λ*us′*. *case snd* (*fst* (*snd us′*)) *of None* ⇒ *return-spmf False* | *Some a* ⇒ *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* ≠ *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* ∧ (*case* (*gs*, *idgs*) *of* (*None*, *Inr id′*) ⇒ *id′* = j_s − *id* ∧ *id* ≤ j_s | (*Some gs*, *Inl gs′*) ⇒ *gs* = *gs′* ∧ *id* > j_s | - ⇒ *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* (⟶) *?game2 ?game3*
by(*clarsimp simp add*: *initialize-def intro*!: *ord-spmf-bind-reflI*)
also
let *?X* = λ(*gsid*, *s*) (*gid*, *s′*). *s* = *s′* ∧ *rel-sum* (λ(*g*, *s1*) *g′*. *g* = *g′* ∧ *s1* = *s′*) (=) *gsid gid*
have *rel-spmf* (⟶) *?game3* (*game-single* (*reduction q* $\mathscr{A}$))
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* (⟶) *?game2* (*game-single* (*reduction q* $\mathscr{A}$))
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* ≤ *spmf* (*game-single* (*reduction q* $\mathscr{A}$)) *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* :: $'key \Rightarrow 'x \Rightarrow 'hash$
begin

type-synonym $('x', 'hash')$ *adversary* = $(unit, 'x' + ('x' \times 'hash'), 'hash' + unit)$ *gpv*

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

definition *oracle-flag* :: $'key \Rightarrow ('x \times 'hash, unit, bool \times 'x\ set)$ *callee*
where
oracle-flag = $(\lambda key\ (flg, L)\ (y, t).$
return-spmf $((), (flg \vee (t = (hash\ key\ y) \wedge y \notin L), L)))$

abbreviation *oracle* :: $'key \Rightarrow ('x + 'x \times 'hash, 'hash + unit, bool \times 'x\ set)$ *callee*
where *oracle key* $\equiv \dagger(oracle\text{-}hash\ key) \oplus_O oracle\text{-}flag\ key$

definition *game* :: $('x, 'hash)$ *adversary* $\Rightarrow$ *bool spmf*
where
game $\mathscr{A}$ = *do* {
key $\leftarrow$ *key-gen*;
$(\text{-}, (flag', L')) \leftarrow exec\text{-}gpv\ (oracle\ key)\ \mathscr{A}\ (False, \{\})$;
return-spmf flag'
}

definition *advantage* :: $('x, 'hash)$ *adversary* $\Rightarrow$ *real*
where *advantage* $\mathscr{A}$ = *spmf* (*game* $\mathscr{A}$) *True*

type-synonym $('x', 'hash')$ *adversary1* = $('x' \times 'hash', 'x', 'hash')$ *gpv*

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

definition *advantage1* :: (*'x*, *'hash*) *adversary1* ⇒ *real*
where *advantage1* $\mathscr{A}$ = *spmf* (*game1* $\mathscr{A}$) *True*

lemma *advantage-advantage1*:
assumes *bound*: *interaction-bounded-by* (*Not* ∘ *isl*) $\mathscr{A}$ *q*
shows *advantage* $\mathscr{A}$ ≤ *advantage1* (*guessing-many-one.reduction q* (λ- :: *unit*. $\mathscr{A}$) ()) ∗ *q*
proof −
let *?init* = *map-spmf* (λ*key*. (*key*, (), {})) *key-gen*
let *?oracle* = λ*key* . *oracle-hash key*
let *?eval* = λ*key* (- :: *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* $\mathscr{A}$ = *game-multi* (λ-. $\mathscr{A}$)
by(*auto simp add*: *game-multi-def game-def bind-map-spmf intro*!: *bind-spmf-cong*[*OF refl*])
hence *advantage* $\mathscr{A}$ = *advantage-multi* (λ-. $\mathscr{A}$) **by**(*simp add*: *advantage-def advantage-multi-def*)
also have ... ≤ *advantage-single* (*reduction q* (λ-. $\mathscr{A}$)) ∗ *q* **using** *bound*
by(*rule many-single-reduction*)(*auto simp add*: *oracle-hash-def*)
also have *advantage-single* (*reduction q* (λ-. $\mathscr{A}$)) = *advantage1* (*reduction q* (λ-. $\mathscr{A}$) ()) **for** $\mathscr{A}$
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 ← *sample-uniform* (*order* $\mathcal{G}$);
  return-spmf (**g** [^] *x*, *x*)
}

lemma *key-gen-alt*:
key-gen = *map-spmf* (λ*x*. (**g** [^] *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 ← *sample-uniform* (*order* $\mathcal{G}$);
  return-spmf (**g** [^] *y*, (α [^] *y*) ⊗ *msg*)
}

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

definition *adecrypt* :: *'grp priv-key* ⇒ *'grp cipher* ⇒ *'grp option*
where
adecrypt x = (λ(β, ζ). *Some* (ζ ⊗ (*inv* (β [^] *x*))))

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

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

fun *elgamal-adversary* :: (*'grp pub-key*, *'grp plain*, *'grp cipher*, *'state*) *ind-cpa.adversary* ⇒ *'grp ddh.adversary*
where
elgamal-adversary ($\mathscr{A}$*1*, $\mathscr{A}$*2*) α β γ = *TRY do* {
b ← *coin-spmf*;
((*msg1*, *msg2*), σ) ← $\mathscr{A}$*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 ← $\mathscr{A}$*2* (β, γ ⊗ (*if b then msg1 else msg2*)) σ;
return-spmf (*guess* = *b*)
} *ELSE coin-spmf*

end

locale *elgamal* = *elgamal-base* + *cyclic-group* $\mathscr{G}$
begin

theorem *advantage-elgamal*: *ind-cpa.advantage* $\mathscr{A}$ = *ddh.advantage* (*elgamal-adversary* $\mathscr{A}$)
including *monad-normalisation*
proof −
obtain $\mathscr{A}$*1* **and** $\mathscr{A}$*2* **where** $\mathscr{A}$ = ($\mathscr{A}$*1*, $\mathscr{A}$*2*) **by**(*cases* $\mathscr{A}$)
note [*simp*] = *this order-gt-0-iff-finite finite-carrier 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* $\mathscr{A}$) = *TRY do* {
x ← *sample-uniform* (*order* $\mathscr{G}$);
y ← *sample-uniform* (*order* $\mathscr{G}$);
((*msg1*, *msg2*), σ) ← $\mathscr{A}$*1* (**g** [^] *x*);
\- :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
b ← *coin-spmf*;
z ← *map-spmf* (λ*z*. **g** [^] *z* ⊗ (*if b then msg1 else msg2*)) (*sample-uniform* (*order* $\mathscr{G}$));
guess ← $\mathscr{A}$*2* (**g** [^] *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* $\mathscr{G}$);
y ← *sample-uniform* (*order* $\mathscr{G}$);
((*msg1*, *msg2*), σ) ← $\mathscr{A}$*1* (**g** [^] *x*);
- :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
z ← *map-spmf* (λ*z*. **g** [^] *z*) (*sample-uniform* (*order* $\mathscr{G}$));
guess ← $\mathscr{A}$*2* (**g** [^] *y*, *z*) σ;
map-spmf ((=) *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* $\mathscr{A}$) = *ind-cpa.ind-cpa* $\mathscr{A}$
by(*simp add*: *ddh.ddh-0-def IND-CPA-PK-Single.ind-cpa.ind-cpa-def key-gen-def aen-crypt-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 $\mathscr{G}$ :: *security* ⇒ ′*grp cyclic-group*
assumes *elgamal*: ⋀η. *elgamal* ($\mathscr{G}$ η)
begin

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

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

end

context *elgamal-base* **begin**

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

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

lemma *lossless-elgamal-adversary*:
⟦ *ind-cpa.lossless* $\mathscr{A}$; *0* < *order* $\mathscr{G}$ ⟧
⟹ *ddh.lossless* (*elgamal-adversary* $\mathscr{A}$)
by(*cases* $\mathscr{A}$)(*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* ≡ *Map.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* ⇒ (*bitstring*, *'grp*, *bitstring*) *gpv*
where *hash x* ≡ *Pause x Done*

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

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

definition *aencrypt* :: *'grp pub-key* ⇒ *plain* ⇒ (*'grp cipher*, *'grp*, *bitstring*) *gpv*
where
  aencrypt α *msg* = *do* {
    y ← *lift-spmf* (*sample-uniform* (*order* $\mathcal{G}$));

```
    h ← hash (α [^] y);
    Done (g [^] y, h [⊕] 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 (𝒜1, 𝒜2) α β = do {
    (((msg1, msg2), σ), s) ← exec-gpv hash.oracle (𝒜1 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 (𝒜2 (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 *lossless*: *ind-cpa.lossless 𝒜*
shows *ind-cpa.advantage hash.oracle hash.initial 𝒜* ≤ *lcdh.advantage* (*elgamal-adversary 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 *𝒜1 𝒜2* **where** *𝒜* [*simp*]: *𝒜* = (*𝒜1*, *𝒜2*) **by**(*cases 𝒜*)

interpret *cyclic-group*: *cyclic-group 𝒢* **by**(*rule cyclic-group*)
from *finite-carrier* **have** [*simp*]: *order 𝒢* > *0* **using** *order-gt-0-iff-finite* **by**(*simp*)

from *lossless* **have** *lossless1* [*simp*]: ⋀*pk*. *lossless-gpv ℐ-full* (*𝒜1 pk*)
and *lossless2* [*simp*]: ⋀σ *cipher*. *lossless-gpv ℐ-full* (*𝒜2* σ *cipher*)
by(*auto simp add*: *ind-cpa.lossless-def*)

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

define *hash-oracle′* **where** *hash-oracle′* = (λσ *x*. *do* {
h ← *hash x*;
Done (*h*, *insert x* σ)
})
have [*simp*]: *lossless-gpv ℐ-full* (*hash-oracle′* σ *x*) **for** σ *x* **by**(*simp add*: *hash-oracle′-def*)
have [*simp*]: *lossless-gpv ℐ-full* (*inline hash-oracle′* (*𝒜1* α) *s*) **for** α *s*
by(*rule lossless-inline*[**where** *ℐ*=*ℐ-full*]) *simp-all*
define *game0* **where** *game0* = *TRY do* {
(*pk*, -) ← *lift-spmf key-gen*;
b ← *lift-spmf coin-spmf*;
(((*msg1*, *msg2*), σ), *s*) ← *inline hash-oracle′* (*𝒜1 pk*) {};
assert-gpv (*valid-plains msg1 msg2*);
cipher ← *aencrypt pk* (*if b then msg1 else msg2*);
(*guess*, *s′*) ← *inline hash-oracle′* (*𝒜2 cipher* σ) *s*;
Done (*guess* = *b*)
} *ELSE lift-spmf coin-spmf*
{ define *cr* **where** *cr* = (λ- :: *unit*. λ- :: *′a set*. *True*)
have [*transfer-rule*]: *cr* () {} **by**(*simp add*: *cr-def*)
have [*transfer-rule*]: ((=) ===> *cr* ===> *cr*) (λ- σ. σ) *insert* **by**(*simp add*: *rel-fun-def cr-def*)
have [*transfer-rule*]: (*cr* ===> (=) ===> *rel-gpv* (*rel-prod* (=) *cr*) (=)) *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 𝒜* = *game0* **unfolding** *game0-def 𝒜 ind-cpa-pk.ind-cpa.simps*
by(*transfer fixing*: *𝒢 len-plain 𝒜1 𝒜2*)(*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 𝒢*));
b ← *lift-spmf coin-spmf*;

$(((msg1, msg2), \sigma), s) \leftarrow$ *inline hash-oracle′* ($\mathscr{A}$ *1* (**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′* ($\mathscr{A}$*2 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*])

define *hash-oracle″* **where** *hash-oracle″* = (λ(*s*, σ) (*x* :: ′*a*). *do* {
(*h*, σ′) ← *case* σ *x of*
None ⇒ *bind-spmf* (*spmf-of-set* (*nlists UNIV len-plain*)) (λ*bs*. *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′* $\mathscr{A}$ *s*) σ =
map-spmf (λ(*a*, *b*, *c*). ((*a*, *b*), *c*)) (*exec-gpv hash-oracle″* $\mathscr{A}$ (*s*, σ)) **for** $\mathscr{A}$ 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″* ($\mathscr{A}$ *1* α) *s*) **for** *s* α
by(*rule lossless-exec-gpv*[**where** $\mathscr{I}$=$\mathscr{I}$*-full*]) *simp-all*
have [*simp*]: *lossless-spmf* (*exec-gpv hash-oracle″* ($\mathscr{A}$*2* σ *cipher*) *s*) **for** σ *cipher s*
by(*rule lossless-exec-gpv*[**where** $\mathscr{I}$=$\mathscr{I}$*-full*]) *simp-all*

let *?sample* = λ*f*. *bind-spmf* (*sample-uniform* (*order* $\mathscr{G}$)) (λ*x*. *bind-spmf* (*sample-uniform* (*order* $\mathscr{G}$)) (*f x*))
define *game1* **where** *game1* = (λ(*x* :: *nat*) (*y* :: *nat*). *do* {
b ← *coin-spmf*;
(((*msg1*, *msg2*), σ), (*s*, *s-h*)) ← *exec-gpv hash-oracle″* ($\mathscr{A}$ *1* (**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″* ($\mathscr{A}$*2 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** $\mathscr{I}$=$\mathscr{I}$*-full*] *bind-rpv-def nat-pow-pow del*: *bind-spmf-const*)
including *monad-normalisation* **by**(*simp add*: *bind-rpv-def nat-pow-pow*)

```
define game2 where game2 = (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 ← 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 [^] 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)
  }
})
interpret inv'': callee-invariant-on hash-oracle'' λ(s, s-h). s = dom s-h 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 ∈ s) for x
  by unfold-locales(auto simp add: hash-oracle''-def)

{ fix x y :: nat
  let ?bad = λ(s, s-h). g [^] (x ∗ y) ∈ s
  let ?X = λ(s, s-h) (s', s-h'). s = dom s-h ∧ s' = s ∧ s-h = s-h'(g [^] (x ∗ y) := None)
  have bisim:
    rel-spmf (λ(a, s1') (b, s2'). ?bad s1' = ?bad s2' ∧ (¬ ?bad s2' ⟶ a = b ∧ ?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' ∧ (¬ bad' ⟶ 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 [^] (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** $\mathscr{I}=\mathscr{I}$*-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* $\longleftrightarrow$ *bad'*) $\wedge$ ($\neg$ *bad'* $\longrightarrow$ *win* $\longleftrightarrow$ *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*}|
$\leq$ *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*| $\leq$ *spmf* (*map-spmf snd* (*?sample game2*)) *True* **by** *simp*

define *game3*
where *game3* = (λ*f* :: - $\Rightarrow$ - $\Rightarrow$ - $\Rightarrow$ *bitstring spmf* $\Rightarrow$ - *spmf*. λ(*x* :: *nat*) (*y* :: *nat*). *do* {
b $\leftarrow$ *coin-spmf*;
(((*msg1*, *msg2*), σ), (*s*, *s-h*)) $\leftarrow$ *exec-gpv hash-oracle''* ($\mathscr{A}$*1* (**g** [^] *x*)) ({}, *hash.initial*);
TRY do {
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains msg1 msg2*);
h' $\leftarrow$ *f b msg1 msg2* (*spmf-of-set* (*nlists UNIV len-plain*));
let cipher = (**g** [^] *y*, *h'*);
(*guess*, (*s'*, *s-h'*)) $\leftarrow$ *exec-gpv hash-oracle''* ($\mathscr{A}$*2 cipher* σ) (*s*, *s-h*);
return-spmf (*guess* = *b*, **g** [^] (*x* $*$ *y*) $\in$ *s'*)
} *ELSE do* {
b $\leftarrow$ *coin-spmf*;
return-spmf (*b*, **g** [^] (*x* $*$ *y*) $\in$ *s*)
}
})
let *?f* = λ*b msg1 msg2*. *map-spmf* (λ*h*. (*if b then msg1 else msg2*) [$\oplus$] *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* **.**

define *hash-oracle'''* **where** *hash-oracle'''* = (λ(σ :: *'a* ⇒ -). *hash.oracle* σ)
{ define *bisim* **where** *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* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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* ===> (=)) *dom fst* **by**(*simp add*: *bisim-def rel-fun-def*)
have ∗ [*transfer-rule*]: (*bisim* ===> (=)) (λ*x*. *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'''* (𝒜*1* (**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'''* (𝒜*2 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*: 𝒜*1* 𝒢 *len-plain x y* 𝒜*2*) *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* 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*)) ← *exec-gpv hash-oracle''* (𝒜*1* (**g** [^] *x*)) ({}, *hash.initial*);
  TRY do {
    - :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
    h' ← *spmf-of-set* (*nlists UNIV len-plain*);
    (*guess*, (*s'*, *s-h'*)) ← *exec-gpv hash-oracle''* (𝒜*2* (**g** [^] *y*, *h'*) σ) (*s*, *s-h*);

map-spmf ((=) *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* 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* ⟷ *0* < *order* 𝒢
by(*simp add*: *key-gen-def Let-def*)

lemma *lossless-elgamal-adversary*:
⟦ *ind-cpa.lossless* 𝒜; ⋀η. *0* < *order* 𝒢 ⟧
⟹ *lcdh.lossless* (*elgamal-adversary* 𝒜)
by(*cases* 𝒜)(*auto simp add*: *lcdh.lossless-def ind-cpa.lossless-def split-def Let-def intro*!: *lossless-exec-gpv*[**where** ℐ=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* = {} ∨ *A* ∩ *B* = {})
case *False*
define *A′* **where** *A′* ≡ *A* ∩ *B*
from *False* **have** *C*: *C* ≠ {} **and** *A′*: *A′* ≠ {} **by**(*auto simp add*: *A′-def*)
have *B*: *B* = *A′* ∪ *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′* ∩ *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 (∑*x*∈*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* ≠ {}
begin

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

definition *game* :: *bool* ⇒ *′a adversary* ⇒ *bool spmf* **where**
game b 𝒜 = *run-gpv* (*if b then rp.random-permutation else rf.random-oracle*) 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 𝒜 = |*spmf* (*prp-game* 𝒜) *True* − *spmf* (*prf-game* 𝒜) *True*|

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

lemma *advantage-le-1*: *advantage* 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 $\mathscr{I}$.*lifting* **begin**
lift-definition $\mathscr{I}$ :: ('*a*, '*a*) $\mathscr{I}$ **is** (λx. *if* $x \in A$ *then A else* {}) **.**
lemma *outs-*$\mathscr{I}$*-*$\mathscr{I}$ [*simp*]: *outs-*$\mathscr{I}$ $\mathscr{I}$ = *A* **by** *transfer auto*
lemma *responses-*$\mathscr{I}$*-*$\mathscr{I}$ [*simp*]: *responses-*$\mathscr{I}$ $\mathscr{I}$ *x* = (*if* $x \in A$ *then A else* {}) **by** *transfer simp*
lifting-update $\mathscr{I}$.*lifting*
lifting-forget $\mathscr{I}$.*lifting*
end

lemma *rp-rf*:
assumes *bound*: *interaction-any-bounded-by* $\mathscr{A}$ *q*
and *lossless*: *lossless-gpv* $\mathscr{I}$ $\mathscr{A}$
and *WT*: $\mathscr{I} \vdash g\ \mathscr{A}\ \surd$
shows *advantage* $\mathscr{A} \leq q * q$ / *card A*
including *lifting-syntax*
proof −
let *?run* = λb. *exec-gpv* (*if b then rp.random-permutation else rf.random-oracle*) $\mathscr{A}$ *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*) (λy. *if* $y \in$ *ran* σ *then map-spmf* ($\lambda y'$. (*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*) (λy. *if* $y \in$ *ran* σ *then map-spmf* ($\lambda y'$. (*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* (=)
define *init* :: *bool* × ('*a* ⇀ '*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*) (λy. *if* $y \in$ *ran* σ *then spmf-of-set* (*A* − *ran* σ) *else return-spmf y*)) (λy. *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* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *?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* $\mathscr{A}$) *True* = *spmf* (*run-gpv rp-bad* $\mathscr{A}$ *init*) *True*
unfolding *vimage-def game-def if-True* **by** *transfer-prover*
moreover {
define *collision* :: ('*a* ⇀ '*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*: $\llbracket$ *m x = None*; *y* $\in$ *ran m* $\rrbracket \Longrightarrow$ *collision* $(m(x \mapsto y))$ **for** *m x y*
by(*auto simp add*: *collision-def ran-def intro*: *rev-image-eqI*)
have *collision-map-upd-iff*: $\neg$ *collision m* $\Longrightarrow$ *collision* $(m(x \mapsto y)) \longleftrightarrow$ *y* $\in$ *ran m* $\wedge$ *m x* $\neq$ *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 $?X = \lambda\sigma 1\ (bad, \sigma 2).\ \sigma 1 = \sigma 2 \wedge \neg$ *collision* $\sigma 1 \wedge \neg$ *bad*
and $?I1 = \lambda\sigma 1.$ *dom* $\sigma 1 \subseteq A \wedge$ *ran* $\sigma 1 \subseteq A$
and $?I2 = \lambda(bad, \sigma 2).$ *dom* $\sigma 2 \subseteq A \wedge$ *ran* $\sigma 2 \subseteq A$
let *?X-bad* $= \lambda\sigma 1\ s2.\ ?I1\ \sigma 1 \wedge ?I2\ s2$
have [*simp*]: $\mathscr{I} \vdash c$ *rf.random-oracle s1* $\surd$ **if** *ran s1* $\subseteq A$ **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* $\mathscr{I}$
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* $\mathscr{I}$ **.**
have [*simp*]: $\mathscr{I} \vdash c$ *rp-bad s2* $\surd$ **if** *ran* (*snd s2*) $\subseteq A$ **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* $(\lambda\sigma 1.\ ?bad1\ \sigma 1 \wedge ?I1\ \sigma 1)\ \mathscr{I}$
by(*unfold-locales*)(*clarsimp simp add*: *rf.random-oracle-def finite-A split*: *option.split-asm*)+
have [*simp*]: *callee-invariant-on rp-bad* $(\lambda s2.\ ?I2\ s2)\ \mathscr{I}$
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* $(\lambda s2.\ ?bad2\ s2 \wedge ?I2\ s2)\ \mathscr{I}$
by(*unfold-locales*)(*auto 4 3 simp add*: *rp-bad-simps finite-A split*: *option.splits if-split-asm iff del*: *domIff*)
have [*simp*]: $\mathscr{I} \vdash c$ *rp-bad* $(bad, \sigma 2)\ \surd$ **if** *ran* $\sigma 2 \subseteq A$ **for** *bad* $\sigma 2$ **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* $(b, \sigma 2)$ *x*) **if** $x \in A$ *dom* $\sigma 2 \subseteq A$ *ran* $\sigma 2 \subseteq A$ **for** *b* $\sigma 2$ *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* $(\lambda(b1, \sigma 1)\ (b2, state2).\ (?bad1\ \sigma 1 \longleftrightarrow ?bad2\ state2) \wedge$ (*if ?bad2 state2 then ?X-bad* $\sigma 1$ *state2 else* $b1 = b2 \wedge ?X\ \sigma 1\ state2$))
((*if False then rp.random-permutation else rf.random-oracle*) *s1 x*) (*rp-bad s2 x*)
if *?X s1 s2 x* $\in$ *outs-*$\mathscr{I}$ $\mathscr{I}$ *?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*
$(\lambda(b1, \sigma 1)\ (b2, state2).\ (?bad1\ \sigma 1 \longleftrightarrow ?bad2\ state2) \wedge$ (*if ?bad2 state2 then ?X-bad* $\sigma 1$ *state2 else* $b1 = b2 \wedge ?X\ \sigma 1\ state2$))
(*?run False*) (*exec-gpv rp-bad* $\mathscr{A}$ *init*)
by(*rule exec-gpv-oracle-bisim-bad-invariant*[**where** $\mathscr{I} = \mathscr{I}$ **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* 𝒜 *init*) *True*| ≤ *spmf* (*map-spmf* (*?bad1* ∘ *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* ... ≤ *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* ∘ *snd*) (*?run False*)) *True* ≤ *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* 𝒜 *init*) *True* − *spmf* (*run-gpv rf*.*random-oracle* 𝒜 *Map.empty*) *True*| ≤ *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 [19, §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 … ≤ *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*. ∃*xa*∈*W*. ∃*y*∈*W*. *hash x xa* = *hash x y* ∧ *xa* ≠ *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*. ∃*xa*∈*W*. ∃*y*∈*W*. *hash x xa* = *hash x y* ∧ *xa* ≠ *y*} = (⋃(*w*, *w′*) ∈ *W*× *W*. {*x*. *hash x w* = *hash x w′* ∧ *w* ≠ *w′*})
    by(*auto*)
  also have *?M* … ≤ (∑(*w*, *w′*)∈*W* × *W*. *?M* {*x*. *hash x w* = *hash x w′* ∧ *w* ≠ *w′*})
   by(*auto intro*!: *measure-spmf*.*finite-measure-subadditive-finite simp add*: *split-def assms*)
  also have … ≤ (∑(*w*, *w′*)∈*W* × *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

  { define cr where 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*]

define *semi-forgetful-RO* **where** *semi-forgetful-RO* = (λ*seed* :: ′*seed*. λ(σ :: ′α ⇀ ′β × ′γ, *b* :: *bool*). λ*x*.
case σ (*h seed x*) *of Some* (*a*, *y*) ⇒ *return-spmf* (*y*, (σ, *a* ≠ *x* ∨ *b*))
| *None* ⇒ *bind-spmf rand* (λ*y*. *return-spmf* (*y*, (σ(*h seed x* ↦ (*x*, *y*)), *b*))))

define *game-semi-forgetful* **where** *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*)

{ define *cr*
where *cr* = (λ(- :: *unit*, σ) (σ′ :: ′α ⇒ (′β × ′γ) *option*, - :: *bool*). σ = *map-option snd* ∘ σ′)
define *initial* **where** *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*]: ((=) ===> *cr* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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*

define *game-semi-forgetful-bad* **where** *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* ($\lambda\sigma$. ¬ *inj-on* (*h seed*) (*dom* σ)) **for** *seed*
by *unfold-locales*(*auto simp add*: *prf.random-oracle-def split*: *option.split-asm*)

define *invar*
where *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*) $\mathscr{A}$ (*Map.empty*, *False*))
(*exec-gpv prf.random-oracle* $\mathscr{A}$ *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* σ*1b x*) (*prf.random-oracle s2 x*)
if *X*: *invar seed* σ*1b s2* **for** *s2* σ*1b x*
proof −
obtain σ*1 b* **where** [*simp*]: σ*1b* = (σ*1*, *b*) **by**(*cases* σ*1b*)
from *X* **have** *not-b*: ¬ *b*
and *dom*: *dom* σ*1* = *h seed* ‘ *dom s2*
and *eq*: ∀*x*∈*dom s2*. σ*1* (*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** σ*1* (*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 σ*1* (*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

define *game-A* **where** *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* $\circ$ *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*| $\leq$ *spmf* (*map-spmf* (*snd* $\circ$ *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*| $\leq$ *spmf* (*map-spmf snd game-A*) *True*
using *bad1-bad2* **by** (*simp*)

define *game-B* **where** *game-B* = *do* {
(b, σ) $\leftarrow$ *exec-gpv prf*.*random-oracle* $\mathscr{A}$ *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* $\leq$ *hash*.ε-*uh* $*$ *q* $*$ *q* **unfolding** *game-B-def*
proof(*rule spmf-bind-leI*, *clarify*)
fix *b* σ
assume $*$: (b, σ) $\in$ *set-spmf* (*exec-gpv prf*.*random-oracle* $\mathscr{A}$ *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* $\leq$ *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) $\leq$ *q* + *card* (*dom* (*Map*.*empty* :: $'\beta \Rightarrow\ '\gamma$ *option*))
by(*rule prf*.*card-dom-random-oracle*[*OF bounded* $*$]) *simp*
then have *card* (*dom* σ) $*$ *card* (*dom* σ) $\leq$ *q* $*$ *q* **using** *mult-le-mono* **by** *auto*
finally show *spmf* (*hash*.*game-hash-set* (*dom* σ)) *True* $\leq$ *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* $\mathscr{A}$ = *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* $\mathscr{A}$ $\leq$ |*spmf* (*prf*.*game-0* ?$\mathscr{A}$) *True* − *spmf* (*prf* $'$.*game-1* $\mathscr{A}$) *True*|
using *hop1* **by**(*simp add*: *prf* $'$.*advantage-def*)
also have ... $\leq$ *prf*.*advantage* ?$\mathscr{A}$ + |*spmf* (*prf*.*game-1* ?$\mathscr{A}$) *True* − *spmf* (*prf* $'$.*game-1* $\mathscr{A}$) *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 [16].

```
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: ⋀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* $]\!]$
$\Longrightarrow$ *encrypt key plain* $\gg=$ (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* $\mathscr{I}$*-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 $\mathscr{A}$ = *do* {
let ($\mathscr{A}$*1*, $\mathscr{A}$*2*) = $\mathscr{A}$;
(((*p1*, *p2*), σ), *n*) ← *inline prf-encrypt-oracle* $\mathscr{A}$*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* ($\mathscr{A}$*2 c* σ) *n*;
Done (*b'* = *b*)
} *else lift-spmf coin-spmf*
}

theorem *prf-encrypt-advantage*:
assumes *ind-cpa.ibounded-by* $\mathscr{A}$ *q*
and *lossless-gpv* $\mathscr{I}$*-full* (*fst* $\mathscr{A}$)
and ⋀*cipher* σ. *lossless-gpv* $\mathscr{I}$*-full* (*snd* $\mathscr{A}$ *cipher* σ)
shows *ind-cpa.advantage* $\mathscr{A}$ ≤ *prf.advantage* (*prf-adversary* $\mathscr{A}$) + *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 *𝒜1 𝒜2* **where** *𝒜*: *𝒜* = (*𝒜1*, *𝒜2*) **by**(*cases 𝒜*)
from ‹*ind-cpa.ibounded-by - -*›
obtain *q1 q2* :: *nat*
where *q1*: *interaction-any-bounded-by 𝒜1 q1*
and *q2*: ⋀*cipher* σ. *interaction-any-bounded-by* (*𝒜2 cipher* σ) *q2*
and *q1* + *q2* ≤ *q*
unfolding *𝒜* **by**(*rule ind-cpa.ibounded-byE*)(*auto simp add*: *iadd-le-enat-iff*)
from *𝒜 assms* **have** *lossless1*: *lossless-gpv ℐ-full 𝒜1*
and *lossless2*: ⋀*cipher* σ. *lossless-gpv ℐ-full* (*𝒜2 cipher* σ) **by** *simp-all*
have *weight1*: ⋀*oracle s*. (⋀*s x*. *lossless-spmf* (*oracle s x*))
⟹ *weight-spmf* (*exec-gpv oracle 𝒜1 s*) = *1*
by(*rule lossless-weight-spmfD*)(*rule lossless-exec-gpv*[*OF lossless1*], *simp-all*)
have *weight2*: ⋀*oracle s cipher* σ. (⋀*s x*. *lossless-spmf* (*oracle s x*))
⟹ *weight-spmf* (*exec-gpv oracle* (*𝒜2 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*: ⋀*key*. *rel-spmf* (λ(*x*, -) (*y*, -). *x* = *y*)
(*exec-gpv* (*ind-cpa.encrypt-oracle key*) *𝒜1* ())
(*exec-gpv* (*?oracle1 key*) *𝒜1* ((), ()))
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*: ⋀*key cipher* σ. *rel-spmf* (λ(*x*, -) (*y*, -). *x* = *y*)
(*exec-gpv* (*ind-cpa.encrypt-oracle key*) (*𝒜2 cipher* σ) ())
(*exec-gpv* (*?oracle1 key*) (*𝒜2 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* (=) (*ind-cpa.ind-cpa 𝒜*) (*prf.game-0* (*prf-adversary 𝒜*))
unfolding *IND-CPA.ind-cpa.ind-cpa-def 𝒜 key-gen-def Let-def prf-adversary-def Pseudo-Random-Function.prf.gam*
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

define *rf-encrypt* **where** *rf-encrypt* = (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* λs. *finite* (*dom s*) $\mathscr{I}$*-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*)

define *game2* **where** *game2* = *do* {
(((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* $\mathscr{A}$ *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* ($\mathscr{A}$*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* $\mathscr{A}$ *1* ((), *Map.empty*)) (*exec-gpv rf-encrypt* $\mathscr{A}$ *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*: $\bigwedge$*cipher* σ *s*. *rel-spmf ?I* (*exec-gpv ?oracle2* ($\mathscr{A}$*2 cipher* σ) ((), *s*)) (*exec-gpv rf-encrypt* ($\mathscr{A}$*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* (=) (*prf.game-1* (*prf-adversary* $\mathscr{A}$)) *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* $\mathscr{A}$ *if-distribs intro*!: *rel-spmf-bindI*[*OF bisim2*] *rel-spmf-bind-reflI rel-spmf-bindI*[*OF bisim1*])

define *game2-a* **where** *game2-a* = *do* {
r ← *spmf-of-set* (*nlists UNIV len*);
(((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* $\mathscr{A}$ *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* ($\mathscr{A}$*2 cipher* σ) *s2*;
return-spmf (*b'* = *b*, *bad*)
} *else coin-spmf* ⤜ (λ*b*. *return-spmf* (*b*, *bad*))
}

define *game2-b* **where** *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* (𝒜*2 cipher* σ) (*s1*(*r* ↦ *pad*));
return-spmf (*b*′ = *b*, *bad*)
} *else coin-spmf* ≫= (λ*b*. *return-spmf* (*b*, *bad*))
}

have *game2* = *do* {
r ← *spmf-of-set* (*nlists UNIV len*);
(((*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*);
(*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*)
} *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 if-distribR 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* (=) (*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*)

define *game2-a-bad* **where** *game2-a-bad* = *do* {
r ← *spmf-of-set* (*nlists UNIV len*);
(((*p0*, *p1*), σ), *s1*) ← *exec-gpv rf-encrypt* 𝒜 *1 Map.empty*;
return-spmf (*r* ∈ *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*: ⋀*B* :: *bool list set*. *card* (*B* ∩ *nlists UNIV len*) ≤ *card* (*nlists UNIV len* :: *bool list set*)
by(*rule card-mono*) *simp-all*
then have *spmf game2-a-bad True* = ∫⁺ *x*. *card* (*dom* (*snd x*) ∩ *nlists UNIV len*) / 2 ^ *len* ∂*measure-spmf* (*exec-gpv rf-encrypt* 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*) ∈ *set-spmf* (*exec-gpv rf-encrypt* 𝒜 *1 Map.empty*)
hence *finite* (*dom s*) **by**(*rule rf-finite.exec-gpv-invariant*) *simp-all*
hence *1*: *card* (*dom s* ∩ *nlists UNIV len*) ≤ *card* (*dom s*) **by**(*intro card-mono*) *simp-all*
moreover from *q1* ∗
have *card* (*dom s*) ≤ *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* ∩ *nlists UNIV len*) ≤ *q1* **by**(*simp*) }
then have ... ≤ ∫⁺ *x*. *q1* / 2 ^ *len* ∂*measure-spmf* (*exec-gpv rf-encrypt* 𝒜 *1 Map.empty*)
by(*intro nn-integral-mono-AE*)(*clarsimp simp add*: *field-simps*)
also have ... ≤ *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* ≤ *q1* / 2 ^ *len* **by** *simp*

define *rf-encrypt-bad*
where *rf-encrypt-bad* = (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* ∨ *r* = *secret*)))))
have *rf-encrypt-bad-sticky* [*simp*]: ⋀*s*. *callee-invariant* (*rf-encrypt-bad s*) *snd*
by(*unfold-locales*)(*auto simp add*: *rf-encrypt-bad-def*)
have *lossless-rf-encrypt* [*simp*]: ⋀*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*)

define *game2-c* **where** *game2-c* = *do* {
r ← *spmf-of-set* (*nlists UNIV len*);
(((*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);
    pad ← spmf-of-set (nlists UNIV len);
    let cipher = (r, xor-list pb pad);
    (b', (s2, bad)) ← exec-gpv (rf-encrypt-bad r) (𝒜2 cipher σ) (s1(r ↦ pad), False);
    return-spmf (b' = b, bad)
  } else coin-spmf ≫= (λb. return-spmf (b, False))
}

have bisim2c-bad: ⋀cipher σ s x r. rel-spmf (λ(x, -) (y, -). x = y)
  (exec-gpv rf-encrypt (𝒜2 cipher σ) (s(x ↦ r)))
  (exec-gpv (rf-encrypt-bad x) (𝒜2 cipher σ) (s(x ↦ r), False))
  by(rule exec-gpv-oracle-bisim[where X=λs (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 (=) (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])

define game2-d where game2-d = 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 ← spmf-of-set (nlists UNIV len);
    let cipher = (r, xor-list pb pad);
    (b', (s2, bad)) ← exec-gpv (rf-encrypt-bad r) (𝒜2 cipher σ) (s1, False);
    return-spmf (b' = b, bad)
  } else coin-spmf ≫= (λb. 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) (𝒜2 cipher σ) (s(x ↦ r), False))
        (exec-gpv (rf-encrypt-bad x) (𝒜2 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 (=) (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* (=) (*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* ∩ (*λx*. *x* = *challenge*) −' {*True*}) ≤ *card* {*challenge*}
by(*rule card-mono*) *auto*
then have *spmf* (*map-spmf* (*snd* ∘ *snd*) (*exec-gpv* (*rf-encrypt-bad challenge*) (*𝒜2 cipher σ*) (*s*, *False*))) *True* ≤ (*1* / *2* ^ *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*) (*𝒜2 cipher σ*) (*s*, *False*)))) ≤ (*1* / *2* ^ *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* ^ *len* ∂*measure-spmf* (*spmf-of-set* (*nlists UNIV len*)) ∂*measure-spmf coin-spmf*
else 0)
∂*measure-spmf* (*exec-gpv rf-encrypt 𝒜1 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*=*λx*. *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* ^ *len*) *else q2* / *2* ^ *len*)
∂*measure-spmf* (*exec-gpv rf-encrypt 𝒜1 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* ^ *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* ^ *len* **by** *simp*

define *game3* **where** *game3* = *do* {

(((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);
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 (𝒜2 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) (𝒜2 cipher σ) (s, False))
(exec-gpv rf-encrypt (𝒜2 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 (=) (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 ^ 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 ^ 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 ^ len + q2 / 2 ^ 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 𝒜1 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 (𝒜2 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 𝒜1 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 (𝒜2 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 𝒜1 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 (𝒜2 cipher σ) s1;
       map-spmf ((=) 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 𝒜 ≤ prf.advantage (prf-adversary 𝒜) + |spmf (prf.game-1
(prf-adversary 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 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* $\mathscr{A}$ *q*
shows *prf.ibounded-by* (*prf-adversary* $\mathscr{A}$) (*1* + *q*)
proof −
fix η
from *assms* **have** *ind-cpa.ibounded-by* $\mathscr{A}$ *q* **by** *blast*
then obtain *q1 q2* **where** *q*: *q1* + *q2* ≤ *q*
and [*interaction-bound*]: *interaction-any-bounded-by* (*fst* $\mathscr{A}$) *q1*
⋀*x* σ. *interaction-any-bounded-by* (*snd* $\mathscr{A}$ *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* $\mathscr{A}$) (*1* + *q*) **using** *q*
apply (*simp only*: *prf-adversary-def Let-def split-def*)
apply −
apply *interaction-bound*
apply (*auto simp add*: *iadd-SUP-le-iff SUP-le-iff add.assoc* [*symmetric*] *one-enat-def cong del*: *image-cong-simp cong add*: *SUP-cong-simp*)
done
qed

lemma *lossless-prf-adversary*: *ind-cpa.lossless* $\mathscr{A}$ ⟹ *prf.lossless* (*prf-adversary* $\mathscr{A}$)
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*: ⋀η *xs ys*. ⟦ *length xs* = *len* η; *length ys* = *len* η ⟧ ⟹ *length* (*f* η *xs ys*) = *len* η
and *negligible-len* [*negligible-intros*]: *negligible* (λη. *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* (λη. *prf.advantage* η (*prf-adversary* η ($\mathscr{A}$ η)))
and *q*: ⋀η. *ind-cpa.ibounded-by* ($\mathscr{A}$ η) (*q* η) **and** [*negligible-intros*]: *polynomial q*
and *lossless*: ⋀η. *ind-cpa.lossless* ($\mathscr{A}$ η)
shows *negligible* (λη. *ind-cpa.advantage* η ($\mathscr{A}$ η))
proof(*rule negligible-mono*)
show *negligible* (λη. *prf.advantage* η (*prf-adversary* η ($\mathscr{A}$ η)) + *q* η / *2* ^ *len* η)
by(*intro negligible-intros*)
{ **fix** η
from ‹*ind-cpa.ibounded-by* - -› **have** *ind-cpa.ibounded-by* ($\mathscr{A}$ η) (*q* η) **by** *blast*
moreover from *lossless* **have** *ind-cpa.lossless* ($\mathscr{A}$ η) **by** *blast*
hence *lossless-gpv* $\mathscr{I}$*-full* (*fst* ($\mathscr{A}$ η)) ⋀*cipher* σ. *lossless-gpv* $\mathscr{I}$*-full* (*snd* ($\mathscr{A}$ η) *cipher* σ)

by(*auto simp add*: *ind-cpa.lossless-def*)
ultimately have *ind-cpa.advantage* η ($\mathscr{A}$ η) $\leq$ *prf.advantage* η (*prf-adversary* η ($\mathscr{A}$ η)) + *q* η / 2 ^ *len* η
by(*rule prf-encrypt-advantage*) **}**
hence *eventually* ($\lambda\eta$. |*ind-cpa.advantage* η ($\mathscr{A}$ η)| $\leq$ *1* $*$ |*prf.advantage* η (*prf-adversary* η ($\mathscr{A}$ 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 ($\mathscr{A}$ η)) $\in$ *O*($\lambda\eta$. *prf.advantage* η (*prf-adversary* η ($\mathscr{A}$ 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 [19, §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* ⇒ *bitstring* ⇒ *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* ⇒ *bitstring* ⇒ *'hash*
+
assumes *prf-domain-finite*: *finite prf-domain*
assumes *prf-domain-nonempty*: *prf-domain* ≠ {}
assumes *prf-domain-length*: *x* ∈ *prf-domain* ⟹ *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* λ*m*. *length m* = *prf-clen* **.**
interpretation *ind-cca'*: *ind-cca key-gen encrypt* λ - -. *None* λ*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* $\mathscr{A}$ = *do* {
k ← *lift-spmf prf-key-gen*;
b ← *lift-spmf coin-spmf*;
(-, (*L*, *D*)) ← *inline* (*intercept-upf k b*) $\mathscr{A}$ ({}, {});
Done () }

lemma *lossless-reduction-upf* [*simp*]:
lossless-gpv ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) $\mathscr{A}$ $\Longrightarrow$ *lossless-gpv* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) (*reduction-upf* $\mathscr{A}$)
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* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) $\mathscr{A}$
shows |*spmf* (*ind-cca.game* $\mathscr{A}$) *True* − *spmf* (*ind-cca′.game* $\mathscr{A}$) *True*| ≤ *UPF.advantage* (*reduction-upf* $\mathscr{A}$)
proof −
define *oracle-decrypt0′* **where** *oracle-decrypt0′* ≡ (λ*key* (*bad*, *L*) (*x′*, *c′*, *t′*). *return-spmf* (
if (*x′*, *c′*, *t′*) ∈ *L* ∨ *length x′* ≠ *prf-dlen then* (*None*, (*bad*, *L*))
else (*decrypt key* (*x′*, *c′*, *t′*), (*bad* ∨ *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′*) ∈ *L* ∨ *length x′* ≠ *prf-dlen then* (*None*, (*bad*, *L*))
else (*decrypt key* (*x′*, *c′*, *t′*), (*bad* ∨ *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*)

define *oracle-decrypt1′*
where *oracle-decrypt1′* = (λ(*key* :: *′prf-key* × *′upf-key*) (*bad*, *L*) (*x′*, *c′*, *t′*).
return-spmf (*None* :: *bitstring option*,
(*bad* ∨ *upf-fun* (*snd key*) (*x′* @ *c′*) = *t′* ∧ (*x′*, *c′*, *t′*) ∉ *L* ∧ *length x′* = *prf-dlen*), *L*))
have *oracle-decrypt1′-simps*:
oracle-decrypt1′ key (*bad*, *L*) (*x′*, *c′*, *t′*) =
return-spmf (*None*,
(*bad* ∨ *upf-fun* (*snd key*) (*x′* @ *c′*) = *t′* ∧ (*x′*, *c′*, *t′*) ∉ *L* ∧ *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*)

define *game01′*
where *game01′* = (λ(*decrypt* :: *′prf-key* × *′upf-key* ⇒ (*bitstring* × *bitstring* × *′hash*, *bitstring option*, *bool* × (*bitstring* × *bitstring* × *′hash*) *set*) *callee*) $\mathscr{A}$. *do* {
key ← *key-gen*;
b ← *coin-spmf*;

$(b', (bad', L')) \leftarrow$ *exec-gpv* ($\dagger$(*ind-cca.oracle-encrypt key b*) $\oplus_O$ *decrypt key*) $\mathscr{A}$ (*False*, {});
return-spmf ($b = b'$, *bad'*) })
let *?game0'* = *game01' oracle-decrypt0'*
let *?game1'* = *game01' oracle-decrypt1'*

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

have [*transfer-rule*]:
((=) ===> *?S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *?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*]:
((=) ===> *?S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *?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*)). $bad1 = bad2 \wedge (\neg\, bad2 \longrightarrow b'1 = b'2)$)
(*exec-gpv* ($\dagger$(*ind-cca.oracle-encrypt k b*) $\oplus_O$ *oracle-decrypt1' k*) $\mathscr{A}$ (*False*, {}))
(*exec-gpv* ($\dagger$(*ind-cca.oracle-encrypt k b*) $\oplus_O$ *oracle-decrypt0' k*) $\mathscr{A}$ (*False*, {}))
for *k b*
by (*cases k*; *rule exec-gpv-oracle-bisim-bad*[**where** *X*=(=) **and** *?bad1.0=fst* **and** *?bad2.0=fst* **and** $\mathscr{I} = \mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-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 $\mathscr{A}$ 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'* $\mathscr{A}$)) {(*b*, *bad*). *b*} − *measure* (*measure-spmf* (*?game0'* $\mathscr{A}$)) {(*b*, *bad*). *b*}|
≤ *measure* (*measure-spmf* (*?game1'* $\mathscr{A}$)) {(*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′* 𝒜)) *True*
by (*simp add*: *spmf-conv-measure-spmf measure-map-spmf split-def vimage-def*)
also have *map-spmf snd* (*?game1′* 𝒜) = *UPF*.*game* (*reduction-upf* 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*)) 𝒜 (*False*, {})) =
map-spmf (λ*x*. *fst* (*snd x*)) (*exec-gpv* (*UPF*.*oracle k-upf*) (*inline* (*intercept-upf k-prf b*) 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*)
define *oracle-intercept*
where [*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* 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

define *S* :: *bool* × *′hash cipher-text set* ⇒ (*′hash cipher-text set* × *′hash cipher-text set*) × *bool* × *bitstring set* ⇒ *bool*
where *S* = (λ(*bad*, *L1*) ((*L2*, *D*), -). *bad* = (∃(*x*, *c*, *t*)∈*D*. *upf-fun k-upf* (*x* @ *c*) = *t*) ∧ *L1* = *L2*) ↾ (λ-. *True*) ⊗ *?I*
define *initial* :: (*′hash cipher-text set* × *′hash cipher-text set*) × *bool* × *bitstring set*
where *initial* = (({}, {}), (*False*, {}))
have [*transfer-rule*]: *S ?init-normal initial* **by**(*simp add*: *S-def initial-def*)
have [*transfer-rule*]: (*S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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* (=) *S*) *?lhs* (*exec-gpv oracle-intercept* 𝒜 *initial*)

by(*transfer-prover*)
then show *rel-spmf* ($\lambda x\ y.$ *?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* (=) (*rel-prod* (=) *H*) ===> *rel-spmf* (*rel-prod* (=) *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 $\mathscr{A}$ ≡ *do* {

```
  key ← key-gen;
  b ← coin-spmf;
  (b', D) ← exec-gpv
    (oracle-encrypt2 key b ⊕O oracle-decrypt2 key) 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 𝒜 = do {
  k ← lift-spmf upf-key-gen;
  b ← lift-spmf coin-spmf;
  (b', -) ← inline (intercept-prf k b) 𝒜 ();
  Done (b' = b)
 }
```

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

```
 define oracle-encrypt1''
   where 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), ())})
 define game1'' where game1'' = do {
  key ← key-gen;
  b ← coin-spmf;
```

$(b', D) \leftarrow$ *exec-gpv* (*oracle-encrypt1''* *key b* $\oplus_O$ *oracle-decrypt2 key*) $\mathscr{A}$ ();
return-spmf $(b = b')$}

have *ind-cca'.game* $\mathscr{A}$ = *game1''*
proof −
define *S* **where** *S* = (λ(*L* :: *'hash cipher-text set*) (*D* :: *unit*). *True*)
have [*transfer-rule*]: *S* {} () **by** (*simp add*: *S-def*)
have [*transfer-rule*]:
((=) ===> (=) ===> *S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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*]:
((=) ===> *S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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* $\mathscr{A}$)
proof −
{ **fix** *k-prf k-upf b*
define *oracle-normal*
where *oracle-normal* = *oracle-encrypt1''* (*k-prf*, *k-upf*) *b* $\oplus_O$ *oracle-decrypt2* (*k-prf*, *k-upf*)
define *oracle-intercept*
where *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*) ()))
define *initial* **where** *initial* = ()
define *S* **where** *S* = (λ(*s2* :: *unit*, - :: *unit*) (*s1* :: *unit*). *True*)
have [*transfer-rule*]: *S* ((), ()) *initial* **by**(*simp add*: *S-def initial-def*)
have [*transfer-rule*]: (*S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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* $\mathscr{A}$ *initial*) =
map-spmf (λx. *b* = *fst* (*fst x*)) (*exec-gpv* (*PRF.prf-oracle k-prf*) (*inline* (*intercept-prf k-upf b*) $\mathscr{A}$ ()) ())
by(*transfer fixing*: *b* $\mathscr{A}$ *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 𝒜 = PRF.game-1 (reduction-prf 𝒜)*
proof −
note [*split del*] = *if-split*
{ **fix** *k-upf b k-prf*
define *oracle2*
where *oracle2 = oracle-encrypt2 (k-prf, k-upf) b* $\oplus_O$ *oracle-decrypt2 (k-prf, k-upf)*
define *oracle-intercept*
where *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))*
define *S*
where *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 ===> (=) ===> rel-spmf (rel-prod (=) 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) 𝒜 ()) Map.empty) =*
map-spmf (λx. b = fst x) (exec-gpv oracle2 𝒜 Map-empty)
by(*transfer fixing*: *b prf-clen prf-domain upf-fun k-upf 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 $\mathscr{A}$ ≡ *do* {
key ← *key-gen*;
b ← *coin-spmf*;
(*b'*, (*bad*, *D*)) ← *exec-gpv* (*oracle-encrypt3 key b* $\oplus_O$ *oracle-decrypt2 key*) $\mathscr{A}$ (*False*, *Map-empty*);
return-spmf (*b* = *b'*, *bad*)
}

lemma *round-3*:
assumes *lossless-gpv* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) $\mathscr{A}$
shows |*measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *b*} − *spmf* (*game2* $\mathscr{A}$) *True*|
≤ *measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *bad*}
proof −
define *oracle-encrypt2'*
where *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*)

define *game2'*
where *game2'* = (λ$\mathscr{A}$. *do* {

key ← *key-gen*;
b ← *coin-spmf*;
(*b*′, (*bad*, *D*)) ← *exec-gpv* (*oracle-encrypt2*′ *key b* $\oplus_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*);
return-spmf (*b* = *b*′, *bad*)})

have *game2*′*-eq*: *game2* 𝒜 = *map-spmf fst* (*game2*′ 𝒜)
proof −
define *S* **where** *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*]: ((=) ===> (=) ===> *S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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*]: ((=) ===> *S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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* $\oplus_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*))
(*exec-gpv* (*oracle-encrypt2*′ *key b* $\oplus_O$ *oracle-decrypt2 key*) 𝒜 (*False*, *Map-empty*))
for *key b*
apply(*rule exec-gpv-oracle-bisim-bad*[**where** *X*=(=) **and** *X-bad* = λ- -. *True* **and** *?bad1.0*=*fst* **and** *?bad2.0*=*fst* **and** ℐ=ℐ*-full* $\oplus_{\mathscr{I}}$ 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* 𝒜)) {(*b*, *bad*). *b*} − *measure* (*measure-spmf* (*game2*′ 𝒜)) {(*b*, *bad*). *b*}| ≤
measure (*measure-spmf* (*game3* 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 −
define *oracle-encrypt4*
where *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*)

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

have *map-spmf fst* (*game3* $\mathcal{A}$) = *game4* $\mathcal{A}$
proof −
note [*split del*] = *if-split*
define *S* **where** *S* = (λ(- :: *unit*) (- :: *bool* × (*bitstring*, *bitstring*) *PRF.dict*). *True*)
define *initial3* **where** *initial3* = (*False*, *Map.empty* :: (*bitstring*, *bitstring*) *PRF.dict*)
have [*transfer-rule*]: *S* () *initial3* **by**(*simp add*: *S-def*)
have [*transfer-rule*]: ((=) ===> (=) ===> *S* ===> (=) ===> *rel-spmf* (*rel-prod* (=) *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* (=) *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* $\mathscr{A}$ *q*
shows *measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *bad*} ≤ *q* / *card prf-domain* ∗ *q*
proof −
have *measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *bad*} = *spmf* (*map-spmf snd* (*game3* $\mathscr{A}$)) *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*) $\mathscr{A}$ (*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 ⋀*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*=*λp. 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* 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* (ℐ*-full* ⊕$_{\mathcal{I}}$ ℐ*-full*) 𝒜
and *bound*: *interaction-bounded-by isl* 𝒜 *q*
shows *ind-cca.advantage* 𝒜 ≤
PRF.advantage (*reduction-prf* 𝒜) + *UPF.advantage* (*reduction-upf* 𝒜) +
real q / *real* (*card prf-domain*) ∗ *real q* (**is** *?LHS* ≤ -)
proof −
have *?LHS* ≤ |*spmf* (*ind-cca.game* 𝒜) *True* − *spmf* (*ind-cca′.game* 𝒜) *True*| + |*spmf* (*ind-cca′.game* 𝒜) *True* − *1* / *2*|
(**is** - ≤ *?round1* + *?rest*) **using** *abs-triangle-ineq* **by**(*simp add*: *ind-cca.advantage-def*)
also have *?round1* ≤ *UPF.advantage* (*reduction-upf* 𝒜)
using *lossless* **by**(*rule round-1*)
also have *?rest* ≤ |*spmf* (*ind-cca′.game* 𝒜) *True* − *spmf* (*game2* 𝒜) *True*| + |*spmf*

(*game2* $\mathscr{A}$) *True* − *1* / *2*|
(**is** - ≤ *?round2* + *?rest*) **using** *abs-triangle-ineq* **by** *simp*
also have *?round2* = *PRF.advantage* (*reduction-prf* $\mathscr{A}$) **by**(*rule round-2*)
also have *?rest* ≤ |*measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *b*} − *spmf* (*game2* $\mathscr{A}$) *True*| +
|*measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *b*} − *1* / *2*|
(**is** - ≤ *?round3* + -) **using** *abs-triangle-ineq* **by** *simp*
also have *?round3* ≤ *measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*b*, *bad*). *bad*}
using *round-3*[*OF lossless*] **.**
also have ... ≤ *q* / *card prf-domain* ∗ *q* **using** *bound* **by**(*rule game3-bad*)
also have *measure* (*measure-spmf* (*game3* $\mathscr{A}$)) {(*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* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) $\mathscr{A}$
assumes *q*: *interaction-bounded-by isl* $\mathscr{A}$ *q*
and *q′*: *interaction-bounded-by* (*Not* ∘ *isl*) $\mathscr{A}$ *q′*
shows *ind-cca.advantage* $\mathscr{A}$ ≤
PRF.advantage (*reduction-prf* $\mathscr{A}$) +
UPF.advantage1 (*guessing-many-one.reduction q′* (λ-. *reduction-upf* $\mathscr{A}$) ()) ∗ *q′* +
real q ∗ *real q* / *real* (*card prf-domain*)
proof −
have *ind-cca.advantage* $\mathscr{A}$ ≤
PRF.advantage (*reduction-prf* $\mathscr{A}$) + *UPF.advantage* (*reduction-upf* $\mathscr{A}$) +
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* $\mathscr{A}$) *q′*
unfolding *reduction-upf-def* **by**(*interaction-bound*)(*simp-all add*: *SUP-le-iff*)
then have *UPF.advantage* (*reduction-upf* $\mathscr{A}$) ≤ *UPF.advantage1* (*guessing-many-one.reduction q′* (λ-. *reduction-upf* $\mathscr{A}$) ()) ∗ *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* (ℐ*-full* ⊕ℐ ℐ*-full*) (𝒜 η)
and *bound*: ⋀η. *interaction-bounded-by isl* (𝒜 η) (*q* η)
and *bound'*: ⋀η. *interaction-bounded-by* (*Not* ∘ *isl*) (𝒜 η) (*q'* η)
and [*negligible-intros*]:
polynomial q' polynomial q
negligible (λη. *PRF.advantage* η (*reduction-prf* η (𝒜 η)))
negligible (λη. *UPF.advantage1* η (*guessing-many-one.reduction* (*q'* η) (λ-. *reduction-upf* η (𝒜 η)) ()))
negligible (λη. *1* / *card* (*prf-domain* η))
shows *negligible* (λη. *ind-cca.advantage* η (𝒜 η))
proof −
have *negligible* (λη. *PRF.advantage* η (*reduction-prf* η (𝒜 η)) +
UPF.advantage1 η (*guessing-many-one.reduction* (*q'* η) (λ-. *reduction-upf* η (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

A Tutorial Introduction to CryptHOL

Andreas Lochbihler
Digital Asset (Switzerland) GmbH, Zurich, Switzerland,
mail@andreas-lochbihler.de

S. Reza Sefidgar
Institute of Information Security, Department of Computer Science, ETH Zurich, Zurich, Switzerland,
reza.sefidgar@inf.ethz.ch

Abstract

This tutorial demonstrates how cryptographic security notions, constructions, and game-based security proofs can be formalized using the CryptHOL framework. As a running example, we formalize a variant of the hash-based ElGamal encryption scheme and its IND-CPA security in the random oracle model. This tutorial assumes basic familiarity with Isabelle/HOL and standard cryptographic terminology.

3 Introduction

CryptHOL [2, 11] is a framework for constructing rigorous game-based proofs using the proof assistant Isabelle/HOL [15]. Games are expressed as probabilistic functional programs that are shallowly embedded in higher-order logic (HOL) using CryptHOL's combinators. The security statements, both concrete and asymptotic, are expressed as Isabelle/HOL theorem statements, and their proofs are written declaratively in Isabelle's proof language Isar [21]. This way, Isabelle mechanically checks that all definitions and statements are type-correct and each proof step is a valid logical inference in HOL. This ensures that the resulting theorems are valid in higher-order logic.

This tutorial explains the CryptHOL essentials using a simple security proof. Our running example is a variant of the hashed ElGamal encryption scheme [7]. We formalize the scheme, the indistinguishability under chosen plaintext (IND-CPA) security property, the computational Diffie-Hellman (CDH) hardness assumption [5], and the security proof in the random oracle model. This illustrates how the following aspects of a cryptographic security proof are formalized using CryptHOL:

- Game-based security definitions (CDH in §4.1 and IND-CPA in §4.4)
- Oracles (a random oracle in §4.2)
- Cryptographic schemes, both generic (the concept of an encryption scheme) and a particular instance (the hashed Elgamal scheme in §4.5)
- Security statements (concrete and asymptotic, §5.2 and §6.2)

- Reductions (from IND-CPA to CDH for hashed Elgamal in §5.1)
- Different kinds of proof steps (§5.3–5.8):
  - Using intermediate games
  - Defining failure events and applying indistinguishability-up-to lemmas
  - Equivalence transformations on games

This tutorial assumes that the reader knows the basics of Isabelle/HOL and game-based cryptography and wants to get hands-on experience with CryptHOL. The semantics behind CryptHOL's embedding in higher-order logic and its soundness are not discussed; we refer the reader to the scientific articles for that [2, 11]. Shoup's tutorial [19] provides a good introduction to game-based proofs. The following Isabelle features are frequently used in CryptHOL formalizations; the tutorials are available from the Documentation panel in Isabelle/jEdit.

- Function definitions (tutorials `prog-prove` and `functions`, [10]) for games and reductions
- Locales (tutorial `locales`, [1]) to modularize the formalization
- The Transfer package [9] for automating parametricity and representation independence proofs

This document is generated from a corresponding Isabelle theory file available online [13].[1] It contains this text and all examples, including the security definitions and proofs. We encourage all readers to download the latest version of the tutorial and follow the proofs and examples interactively in Isabelle/HOL. In particular, a Ctrl-click on a formal entity (function, constant, theorem name, ...) jumps to the definition of the entity.

We split the tutorial into a series of recipes for common formalization tasks. In each section, we cover a familiar cryptography concept and show how it is formalized in CryptHOL. Simultaneously, we explain the Isabelle/HOL and functional programming topics that are essential for formalizing game-based proofs.

3.1 Getting started

CryptHOL is available as part of the Archive of Formal Proofs [12]. Cryptography formalizations based on CryptHOL are arranged in Isabelle theory files that import the relevant libraries.

[1]The tutorial has been added to the Archive of Formal Proofs after the release of Isabelle2018. Until the subsequent Isabelle release, the tutorial is only available in the development version at https://devel.isa-afp.org/entries/Game_Based_Crypto.html. The version for Isabelle2018 is available at http://www.andreas-lochbihler.de/pub/crypthol_tutorial.zip.

3.2 Getting started

CryptHOL is available as part of the Archive of Formal Proofs [12]. Cryptography formalizations based on CryptHOL are arranged in Isabelle theory files that import the relevant libraries.

theory *CryptHOL-Tutorial* **imports**
CryptHOL.CryptHOL
begin

The file *CryptHOL.CryptHOL* is the canonical entry point into CryptHOL. For the hashed Elgamal example in this tutorial, the CryptHOL library contains everything that is needed. Additional Isabelle libraries can be imported if necessary.

4 Modelling cryptography using CryptHOL

This section demonstrates how the following cryptographic concepts are modelled in CryptHOL.

- A security property without oracles (§4.1)
- An oracle (§4.2)
- A cryptographic concept (§4.3)
- A security property with an oracle (§4.4)
- A concrete cryptographic scheme (§4.5)

4.1 Security notions without oracles: the CDH assumption

In game-based cryptography, a security property is specified using a game between a benign challenger and an adversary. The probability of an adversary to win the game against the challenger is called its advantage. A cryptographic construction satisfies a security property if the advantage for any "feasible" adversary is "negligible". A typical security proof reduces the security of a construction to the assumed security of its building blocks. In a concrete security proof, where the security parameter is implicit, it is therefore not necessary to formally define "feasibility" and "negligibility", as the security statement establishes a concrete relation between the advantages of specific adversaries.[2] We return to asymptotic security statements in §6.

A formalization of a security property must therefore specify all of the following:

[2]The cryptographic literature sometimes abstracts over the adversary and defines the advantage to be the advantage of the best "feasible" adversary against a game. Such abstraction would require a formalization of feasibility, for which CryptHOL currently does not offer any support. We therefore always consider the advantage of a specific adversary.

- The operations of the scheme (e.g., an algebraic group, an encryption scheme)
- The type of adversary
- The game with the challenger
- The advantage of the adversary as a function of the winning probability

For hashed Elgamal, the cyclic group must satisfy the computational Diffie-Hellman assumption. To keep the proof simple, we formalize the equivalent list version of CDH.

Definition (The list computational Diffie-Hellman game)**.** Let $\mathscr{G}$ be a group of order q with generator $\mathbf{g}$. The List Computational Diffie-Hellman (LCDH) assumption holds for $\mathscr{G}$ if any "feasible" adversary has "negligible" probability in winning the following **LCDH game** against a challenger:

1. The challenger picks x and y randomly (and independently) from $\{0,\ldots,q-1\}$.
2. It passes $\mathbf{g}^x$ and $\mathbf{g}^y$ to the adversary. The adversary generates a set L of guesses about the value of $\mathbf{g}^{xy}$.
3. The adversary wins the game if $\mathbf{g}^{xy} \in L$.

The scheme for LCDH uses only a cyclic group. To make the LCDH formalisation reusable, we formalize the LCDH game for an arbitrary cyclic group $\mathscr{G}$ using Isabelle's module system based on locales. The locale *list-cdh* fixes $\mathscr{G}$ to be a finite cyclic group that has elements of type *'grp* and comes with a generator $\mathbf{g}_{\mathscr{G}}$. Basic facts about finite groups are formalized in the CryptHOL theory *CryptHOL.Cyclic-Group*.[3]

locale *list-cdh* = *cyclic-group* $\mathscr{G}$
for $\mathscr{G}$ :: *'grp cyclic-group* (**structure**)
begin

The LCDH game does not need oracles. The adversary is therefore just a probabilistic function from two group elements to a set of guesses, which are again group elements. In CryptHOL, the probabilistic nature is expressed by the adversary returning a discrete subprobability distribution over sets of guesses, as expressed by the type constructor *spmf*. (Subprobability distributions are like probability distributions except that the whole probability mass may be less than 1, i.e., some

[3]The syntax directive **structure** tells Isabelle that all group operations in the context of the locale refer to the group $\mathscr{G}$ unless stated otherwise. For example, $\mathbf{g}_{\mathscr{G}}$ can be written as $\mathbf{g}$ inside the locale.

Isabelle automatically adds the locale parameters and the assumptions on them to all definitions and lemmas inside that locale. Of course, we could have made the group $\mathscr{G}$ an explicit argument of all functions ourselves, but then we would not benefit from Isabelle's module system, in particular locale instantiation.

probability may be "lost". A subprobability distribution is called lossless, written *lossless-spmf*, if its probability mass is 1.) We define the following abbreviation as a shorthand for the type of LCDH adversaries.[4]

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

The LCDH game itself is expressed as a function from the adversary $\mathscr{A}$ to the subprobability distribution of the adversary winning. CryptHOL provides operators to express these distributions as probabilistic programs and reason about them using program logics:

- The *do* notation desugars to monadic sequencing in the monad of subprobabilities [20]. Intuitively, every line $x \leftarrow p$; samples an element x from the distribution p. The sampling is independent, unless the distribution p depends on previously sampled variables. At the end of the block, the *return-spmf* _ returns whether the adversary has won the game.

- *sample-uniform n* denotes the uniform distribution over the set $\{0, ..., n - 1\}$.

- *order* $\mathscr{G}$ denotes the order of $\mathscr{G}$ and ([^]) :: *'grp* ⇒ *nat* ⇒ *'grp* is the group exponentiation operator.

The LCDH game formalizes the challenger's behavior against an adversary $\mathscr{A}$. In the following definition, the challenger randomly (and independently) picks two natural numbers x and y that are between 0 and $\mathscr{G}$'s order and passes them to the adversary. The adversary then returns a set *zs* of guesses for $g^{x * y}$, where g is the generator of $\mathscr{G}$. The game finally returns a *bool*ean that indicates whether the adversary produced a right guess. Formally, *game* $\mathscr{A}$ is a *bool*ean random variable.

definition *game* :: *'grp adversary* ⇒ *bool spmf* **where**
game $\mathscr{A}$ = *do* {
  $x \leftarrow$ *sample-uniform* (*order* $\mathscr{G}$);
  $y \leftarrow$ *sample-uniform* (*order* $\mathscr{G}$);
  $zs \leftarrow \mathscr{A}$ (**g** [^] x) (**g** [^] y);
  return-spmf (**g** [^] $(x * y) \in zs$)
}

The advantage of the adversary is equivalent to its probability of winning the LCDH game. The function *spmf* :: *'a spmf* ⇒ *'a* ⇒ *real* returns the probability of an elementary event under a given subprobability distribution.

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

[4] Actually, the type of group elements has already been fixed in the locale *list-cdh* to the type variable *'grp*. Unfortunately, such fixed type variables cannot be used in type declarations inside a locale in Isabelle2018. The **type-synonym** *adversary* is therefore parametrized by a different type variable *'grp'*, but it will be used below only with *'grp*.

end

This completes the formalisation of the LCDH game and we close the locale *list-cdh* with **end**. The above definitions are now accessible under the names *game* and *advantage*. Furthermore, when we later instantiate the locale *list-cdh*, they will be specialized to the given pararameters. We will return to this topic in §4.5.

4.2 A Random Oracle

A cryptographic oracle grants an adversary black-box access to a certain information or functionality. In this section, we formalize a random oracle, i.e., an oracle that models a random function with a finite codomain. In the Elgamal security proof, the random oracle represents the hash function: the adversary can query the oracle for a value and the oracle responds with the corresponding "hash".

Like for the LCDH formalization, we wrap the random oracle in the locale *random-oracle* for modularity. The random oracle will return a *bitstring*, i.e. a list of booleans, of length *len*.

type-synonym *bitstring* = *bool list*

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

In CryptHOL, oracles are modeled as probabilistic transition systems that given an initial state and an input, return a subprobability distribution over the output and the successor state. The type synonym $('s, 'a, 'b)$ *oracle'* abbreviates $'s \Rightarrow 'a \Rightarrow ('b \times 's)$ *spmf*.

A random oracle accepts queries of type $'a$ and generates a random bitstring of length *len*. The state of the random oracle remembers its previous responses in a mapping of type $'a \rightharpoonup$ *bitstring*. Upon a query *x*, the oracle first checks whether this query was received before. If so, the oracle returns the same answer again. Otherwise, the oracle randomly samples a bitstring of length *len*, stores it in its state, and returns it alongside with the new state.

type-synonym $'a$ *state* = $'a \rightharpoonup$ *bitstring*

definition *oracle* :: $'a$ *state* $\Rightarrow$ $'a \Rightarrow$ (*bitstring* $\times$ $'a$ *state*) *spmf*
where
  oracle σ *x* = (*case* σ *x of*
    None $\Rightarrow$ *do* {
      bs $\leftarrow$ *spmf-of-set* (*nlists UNIV len*);
      return-spmf (*bs*, $\sigma(x \mapsto bs)$) }
  | *Some bs* $\Rightarrow$ *return-spmf* (*bs*, σ))

Initially, the state of a random oracle is the empty map $\lambda x.$ *None*, as no queries have been asked. For readability, we introduce an abbreviation:

abbreviation $(input)$ *initial* :: *'a state* **where** *initial* $\equiv$ *Map.empty*

This actually completes the formalization of the random oracle. Before we close the locale, we prove two technical lemmas:

1. The lemma *lossless-oracle* states that the distribution over answers and successor states is *lossless*, i.e., a full probability distribution. Many reasoning steps in game-based proofs are only valid for lossless distributions, so it is generally recommended to prove losslessness of all definitions if possible.

2. The lemma *fresh* describes random oracle's behavior when the query is fresh. This lemma makes it possible to automatically unfold the random oracle only when it is known that the query is fresh.

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

lemma *fresh*:
oracle $\sigma\ x =$
$(do\ \{\ bs \leftarrow spmf\text{-}of\text{-}set\ (nlists\ UNIV\ len);$
$return\text{-}spmf\ (bs, \sigma(x \mapsto bs))\ \})$
if $\sigma\ x = None$
using *that* **by**(*simp add*: *oracle-def*)

end

Remark: Independence is the default. Note that - *spmf* represents a discrete probability distribution rather than a random variable. The difference is that every spmf is independent of all other spmfs. There is no implicit space of elementary events via which information may be passed from one random variable to the other. If such information passing is necessary, this must be made explicit in the program. That is why the random oracle explicitly takes a state of previous responses and returns the updated states. Later, whenever the random oracle is used, the user must pass the state around as needed. This also applies to adversaries that may want to store some information.

4.3 Cryptographic concepts: public-key encryption

A cryptographic concept consists of a set of operations and their functional behaviour. We have already seen two simple examples: the cyclic group in §4.1 and the random oracle in §4.2. We have formalized both of them as locales; we have not modelled their functional behavior as this is not needed for the proof. In this section, we now present a more realistic example: public-key encryption with oracle access.

A public-key encryption scheme consists of three algorithms: key generation, encryption, and decryption. They are all probabilistic and, in the most general case, they may access an oracle jointly with the adversary, e.g., a random oracle modelling a hash function. As before, the operations are modelled as parameters of a locale, *ind-cpa-pk*.

- The key generation algorithm *key-gen* outputs a public-private key pair.
- The encryption operation *encrypt* takes a public key and a plaintext of type *'plain* and outputs a ciphertext of type *'cipher*.
- The decryption operation *decrypt* takes a private key and a ciphertext and outputs a plaintext.
- Additionally, the predicate *valid-plains* tests whether the adversary has chosen a valid pair of plaintexts. This operation is needed only in the IND-CPA game definition in the next section, but we include it already here for convenience.

locale *ind-cpa-pk* =
 fixes *key-gen* :: (*'pubkey* × *'privkey*, *'query*, *'response*) *gpv*
  and *encrypt* :: *'pubkey* ⇒ *'plain* ⇒ (*'cipher*, *'query*, *'response*) *gpv*
  and *decrypt* :: *'privkey* ⇒ *'cipher* ⇒ (*'plain*, *'query*, *'response*) *gpv*
  and *valid-plains* :: *'plain* ⇒ *'plain* ⇒ *bool*
begin

The three actual operations are generative probabilistic values (GPV) of type (-, *'query*, *'response*) *gpv*. A GPV is a probabilistic algorithm that has not yet been connected to its oracles; see the theoretical paper [2] for details. The interface to the oracle is abstracted in the two type parameters *'query* for queries and *'response* for responses. As before, we omit the specification of the functional behavior, namely that decrypting an encryption with a key pair returns the plaintext.

4.4 Security notions with oracles: IND-CPA security

In general, there are several security notions for the same cryptographic concept. For encryption schemes, an indistinguishability notion of security [8] is often used. We now formalize the notion indistinguishability under chosen plaintext attacks (IND-CPA) for public-key encryption schemes. Goldwasser et al. [18] showed that IND-CPA is equivalent to semantic security.

Definition (IND-CPA [19])**.** Let *key-gen*, *encrypt* and *decrypt* denote a public-key encryption scheme. The IND-CPA game is a two-stage game between the *adversary* and a *challenger*:

Stage 1 (find):

1. The challenger generates a public key *pk* using *key-gen* and gives the public key to the adversary.
2. The adversary returns two messages m_0 and m_1.
3. The challenger checks that the two messages are a valid pair of plaintexts. (For example, both messages must have the same length.)

Stage 2 (guess):

1. The challenger flips a coin *b* (either 0 or 1) and gives *encrypt pk* m_b to the adversary.
2. The adversary returns a bit b'.

The adversary wins the game if his guess b' is the value of *b*. Let P_{win} denote the winning probability. His advantage is $|P_{win} - 1\ /\ 2|$

Like with the encryption scheme, we will define the game such that the challenger and the adversary have access to a shared oracle, but the oracle is still unspecified. Consequently, the corresponding CryptHOL game is a GPV, like the operations of the abstract encryption scheme. When we specialize the definitions in the next section to the hashed Elgamal scheme, the GPV will be connected to the random oracle.

The type of adversary is now more complicated: It is a pair of probabilistic functions with oracle access, one for each stage of the game. The first computes the pair of plaintext messages and the second guesses the challenge bit. The additional *'state* parameter allows the adversary to maintain state between the two stages.

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

The IND-CPA game formalization below follows the above informal definition. There are three points that need some explanation. First, this game differs from the simpler LCDH game in that it works with GPVs instead of SPMFs. Therefore, probability distributions like coin flips *coin-spmf* must be lifted from SPMFs to GPVs using the coercion *lift-spmf*. Second, the assertion *assert-gpv* (*valid-plains* m_0 m_1) ensures that the pair of messages is valid. Third, the construct *TRY _ ELSE _* catches a violated assertion. In that case, the adversary's advantage drops to 0 because the result of the game is a coin flip, as we are in the *ELSE* branch.

fun *game* :: (*'pubkey, 'plain, 'cipher, 'query, 'response, 'state*) *adversary*
  ⇒ (*bool*, *'query*, *'response*) *gpv*
where
  game ($\mathscr{A}_1$, $\mathscr{A}_2$) = *TRY do* {
    (*pk*, *sk*) ← *key-gen*;
    ((m_0, m_1), σ) ← $\mathscr{A}_1$ *pk*;
    assert-gpv (*valid-plains* m_0 m_1);
    b ← *lift-spmf coin-spmf*;

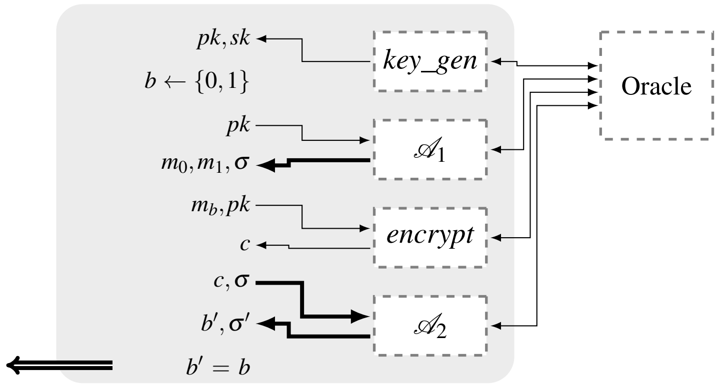

Figure 1: Graphic representation of the generic IND-CPA game.

cipher ← *encrypt pk* (*if b then* m_0 *else* m_1);
b' ← $\mathscr{A}_2$ *cipher* σ;
Done ($b' = b$)
} *ELSE lift-spmf coin-spmf*

Figure 1 visualizes this game as a grey box. The dashed boxes represent parameters of the game or the locale, i.e., parts that have not yet been instantiated. The actual probabilistic program is shown on the left half, which uses the dashed boxes as sub-programs. Arrows in the grey box from the left to the right pass the contents of the variables to the sub-program. Those in the other direction bind the result of the sub-program to new variables. The arrows leaving box indicate the query-response interaction with an oracle. The thick arrows emphasize that the adversary's state is passed around explicitly. The double arrow represents the return value of the game. We will use this to define the adversary's advantage.

As the oracle is not specified in the game, the advantage, too, is parametrized by the oracle, given by the transition function *oracle* :: (*′s*, *′query*, *′response*) *oracle′* and the initial state σ :: *′s* its initial state. The operator *run-gpv* connects the game with the oracle, whereby the GPV becomes an SPMF.

fun *advantage* :: (*′*σ, *′query*, *′response*) *oracle′* × *′*σ
⇒ (*′pubkey*, *′plain*, *′cipher*, *′query*, *′response*, *′state*) *adversary* ⇒ *real*
where
advantage (*oracle*, σ) $\mathscr{A}$ = |*spmf* (*run-gpv oracle* (*game* $\mathscr{A}$) σ) *True* − *1/2*|

end

4.5 Concrete cryptographic constructions: the hashed ElGamal encryption scheme

With all the above modelling definitions in place, we are now ready to explain how concrete cryptographic constructions are expressed in CryptHOL. In general, a cryptographic construction builds a cryptographic concept from possibly several simpler cryptographic concepts. In the running example, the hashed ElGamal cipher [7] constructs a public-key encryption scheme from a finite cyclic group and a hash function. Accordingly, the formalisation consists of three steps:

1. Import the cryptographic concepts on which the construction builds.
2. Define the concrete construction.
3. Instantiate the abstract concepts with the construction.

First, we declare a new locale that imports the two building blocks: the cyclic group from the LCDH game with namespace *lcdh* and the random oracle for the hash function with namespace *ro*. This ensures that the construction can be used for arbitrary cyclic groups. For the message space, it suffices to fix the length *len-plain* of the plaintexts.

locale *hashed-elgamal* =
 lcdh: *list-cdh* $\mathscr{G}$ +
 ro: *random-oracle len-plain*
 for $\mathscr{G}$ :: *'grp cyclic-group* (**structure**)
 and *len-plain* :: *nat*
begin

Second, we formalize the hashed ElGamal encryption scheme. Here is the well-known informal definition.

Definition (Hashed Elgamal encryption scheme)**.** Let G be a cyclic group of order q that has a generator g. Furthermore, let h be a hash function that maps the elements of G to bitstrings, and $\oplus$ be the xor operator on bitstrings. The Hashed-ElGamal encryption scheme is given by the following algorithms:

Key generation Pick an element x randomly from the set $\{0,\dots,q-1\}$ and output the pair (g^x, x), where g^x is the public key and x is the private key.

Encryption Given the public key pk and the message m, pick y randomly from the set $\{0,\dots,q-1\}$ and output the pair $(g^y, h(pk^y) \oplus m)$. Here $\oplus$ denotes the bitwise exclusive-or of two bitstrings.

Decryption Given the private key sk and the ciphertext (α, β), output $h(\alpha^{sk}) \oplus \beta$.

As we can see, the public key is a group element, the private key a natural number, a plaintext a bitstring, and a ciphertext a pair of a group element and a bitstring.[5] For readability, we introduce meaningful abbreviations for these concepts.

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

[5] More precisely, the private key ranges between 0 and $q-1$ and the bitstrings are of length *len-plain*. However, Isabelle/HOL's type system cannot express such properties that depend on locale parameters.

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

We next translate the three algorithms into CryptHOL definitions. The definitions are straightforward except for the hashing. Since we analyze the security in the random oracle model, an application of the hash function H is modelled as a query to the random oracle using the GPV *hash*. Here, *Pause x Done* calls the oracle with query x and returns the oracle's response. Furthermore, we define the plaintext validity predicate to check the length of the adversary's messages produced by the adversary.

abbreviation *hash* :: $'grp \Rightarrow$ (*bitstring*, $'grp$, *bitstring*) *gpv*
where
  hash x ≡ *Pause x Done*

definition *key-gen* :: ($'grp$ *pub-key* × $'grp$ *priv-key*) *spmf*
where
  key-gen = *do* {
    x ← *sample-uniform* (*order* $\mathscr{G}$);
    return-spmf (**g** [^] *x*, *x*)
  }

definition *encrypt* :: $'grp$ *pub-key* ⇒ *plain* ⇒ ($'grp$ *cipher*, $'grp$, *bitstring*) *gpv*
where
  encrypt α *msg* = *do* {
    y ← *lift-spmf* (*sample-uniform* (*order* $\mathscr{G}$));
    h ← *hash* (α [^] *y*);
    Done (**g** [^] *y*, *h* [⊕] *msg*)
  }

definition *decrypt* :: $'grp$ *priv-key* ⇒ $'grp$ *cipher* ⇒ (*plain*, $'grp$, *bitstring*) *gpv*
where
  decrypt x = ($\lambda(\beta, \zeta)$. *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*

The third and last step instantiates the interface of the encryption scheme with the hashed Elgamal scheme. This specializes all definition and theorems in the locale *ind-cpa-pk* to our scheme.

sublocale *ind-cpa*: *ind-cpa-pk* (*lift-spmf key-gen*) *encrypt decrypt valid-plains* **.**

Figure 2 illustrates the instantiation. In comparison to Fig. 1, the boxes for the key generation and the encryption algorithm have been instantiated with the hashed El-

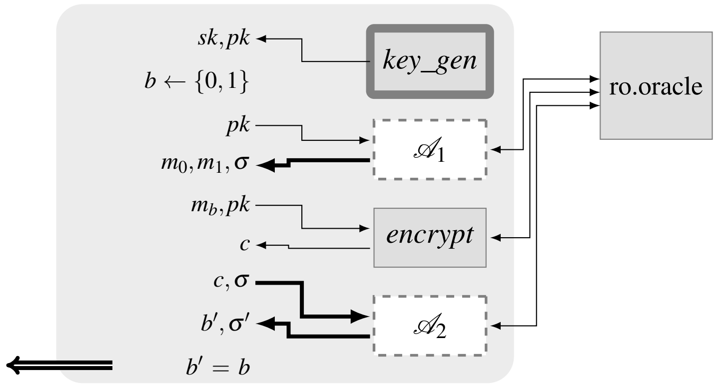

Figure 2: The IND-CPA game instantiated with the Hashed-ElGamal encryption scheme and accessing a random oracle.

gamal definitions from this section. We nevertheless draw the boxes to indicate that the definitions of these algorithms has not yet been inlined in the game definition. The thick grey border around the key generation algorithm denotes the *lift-spmf* operator, which embeds the probabilistic *key-gen* without oracle access into the type of GPVs with oracle access. The oracle has also been instantiated with the random oracle *oracle* imported from *hashed-elgamal*'s parent locale *random-oracle* with prefix *ro*.

5 Cryptographic proofs in CryptHOL

This section explains how cryptographic proofs are expressed in CryptHOL. We will continue our running example by stating and proving the IND-CPA security of the hashed Elgamal encryption scheme under the computational Diffie-Hellman assumption in the random oracle model, using the definitions from the previous section. More precisely, we will formalize a reduction argument (§5.1) and bound the IND-CPA advantage using the CDH advantage. We will *not* formally state the result that CDH hardness in the cyclic group implies IND-CPA security, which quantifies over all feasible adversaries–to that end, we would have to formally define feasibility, for which CryptHOL currently does not offer any support.

The actual proof of the bound consists of several game transformations. We will focus on those steps that illustrate common steps in cryptographic proofs (§5.3–§5.8) .

5.1 The reduction

The security proof involves a reduction argument: We will derive a bound on the advantage of an arbitrary adversary in the IND-CPA game *game* for hashed Elgamal that depends on another adversary's advantage in the LCDH game *game* of the

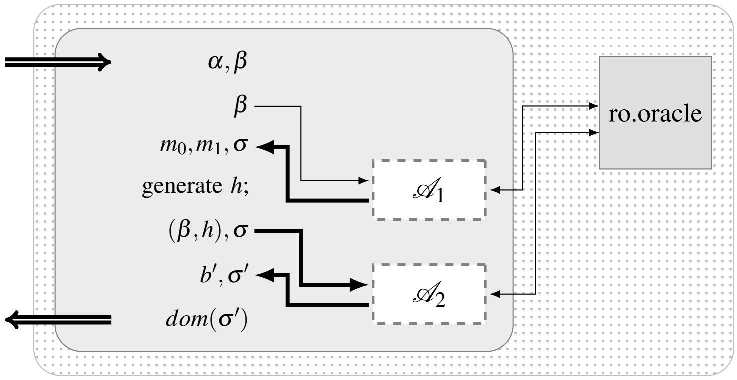

Figure 3: The reduction for the Elgamal security proof.

underlying group. The reduction transforms every IND-CPA adversary $\mathscr{A}$ into a LCDH adversary *elgamal-reduction* $\mathscr{A}$, using $\mathscr{A}$ as a black box. In more detail, it simulates an execution of the IND-CPA game including the random oracle. At the end of the game, the reduction outputs the set of queries that the adversary has sent to the random oracle. The reduction works as follows given a two part IND-CPA adversary $\mathscr{A} = (\mathscr{A}_1, \mathscr{A}_2)$ (Figure 3 visualizes the reduction as the dotted box):

1. It receives two group elements α and β from the LCDH challenger.

2. The reduction passes α to the adversary as the public key and runs $\mathscr{A}_1$ to get messages m_1 and m_2. The adversary is given access to the random oracle with the initial state $\lambda x.$ *None*.

3. The assertion checks that the adversary returns two valid plaintexts, i.e., m_1 and m_2 are strings of length *len-plain*.

4. Instead of actually performing an encryption, the reduction generates a random bitstring h of length *len-plain* (*nlists UNIV len-plain* denotes the set of all bitstrings of length *len-plain* and *spmf-of-set* converts the set into a uniform distribution over the set.)

5. The reduction passes (β, h) as the challenge ciphertext to the adversary in the second phase of the IND-CPA game.

6. The actual guess b' of the adversary is ignored; instead the reduction returns the set *dom* s' of all queries that the adversary made to the random oracle as its guess for the CDH game.

7. If any of the steps after the first phase fails, the reduction's guess is the set *dom* s of oracle queries made during the first phase.

fun *elgamal-reduction*
:: ('*grp pub-key*, *plain*, '*grp cipher*, '*grp*, *bitstring*, '*state*) *ind-cpa.adversary*
⇒ '*grp lcdh.adversary*
where
elgamal-reduction ($\mathscr{A}_1$, $\mathscr{A}_2$) α β = *do* {
(((m_1, m_2), σ), *s*) ← *exec-gpv ro.oracle* ($\mathscr{A}_1$ α) *ro.initial*;
TRY do {
- :: *unit* ← *assert-spmf* (*valid-plains* m_1 m_2);
h ← *spmf-of-set* (*nlists UNIV len-plain*);
(b', s') ← *exec-gpv ro.oracle* ($\mathscr{A}_2$ (β, *h*) σ) *s*;
return-spmf (*dom* s')
} *ELSE return-spmf* (*dom s*)
}

5.2 Concrete security statement

A concrete security statement in CryptHOL has the form: Subject to some side conditions for the adversary $\mathscr{A}$, the advantage in one game is bounded by a function of the transformed adversary's advantage in a different game.[6]

theorem *concrete-security*:
assumes *side conditions for* $\mathscr{A}$
shows $advantage_1$ $\mathscr{A}$ ≤ *f* ($advantage_2$ (*reduction* $\mathscr{A}$))

For the hashed Elgamal scheme, the theorem looks as follows, i.e., the function *f* is the identity function.

theorem *concrete-security-elgamal*:
assumes *lossless*: *ind-cpa.lossless* $\mathscr{A}$
shows *ind-cpa.advantage* (*ro.oracle*, *ro.initial*) $\mathscr{A}$ ≤ *lcdh.advantage* (*elgamal-reduction* $\mathscr{A}$)

Such a statement captures the essence of a concrete security proof. For if there was a feasible adversary $\mathscr{A}$ with non-negligible advantage against the *game*, then *elgamal-reduction* $\mathscr{A}$ would be an adversary against the *game* with at least the same advantage. This implies the existence of an adversary with non-negligible advantage against the cryptographic primitive that was assumed to be secure. What we cannot state formally is that the transformed adversary *elgamal-reduction* $\mathscr{A}$ is feasible as we have not formalized the notion of feasibility. The readers of the formalization must convince themselves that the reduction preserves feasibility.

In the case of *elgamal-reduction*, this should be obvious from the definition (given the theorem's side condition) as the reduction does nothing more than sampling and redirecting data.

[6] A security proof often involves several reductions. The bound then depends on several advantages, one for each reduction.

Our proof for the concrete security theorem needs the side condition that the adversary is lossless. Losslessness for adversaries is similar to losslessness for subprobability distributions. It ensures that the adversary always terminates and returns an answer to the challenger. For the IND-CPA game, we define losslessness as follows:

definition (**in** *ind-cpa-pk*) *lossless*
:: (*'pubkey*, *'plain*, *'cipher*, *'query*, *'response*, *'state*) *adversary* $\Rightarrow$ *bool*
where
lossless $= (\lambda(\mathscr{A}_1, \mathscr{A}_2).\ (\forall pk.\ \textit{lossless-gpv}\ \mathscr{I}\textit{-full}\ (\mathscr{A}_1\ pk))$
$\wedge\ (\forall \textit{cipher}\ \sigma.\ \textit{lossless-gpv}\ \mathscr{I}\textit{-full}\ (\mathscr{A}_2\ \textit{cipher}\ \sigma)))$

So now let's start with the proof.

proof –

As a preparatory step, we split the adversary $\mathscr{A}$ into its two phases $\mathscr{A}_1$ and $\mathscr{A}_2$. We could have made the two phases explicit in the theorem statement, but our form is easier to read and use. We also immediately decompose the losslessness assumption on $\mathscr{A}$.[7]

obtain $\mathscr{A}_1$ $\mathscr{A}_2$ **where** $\mathscr{A}$ [*simp*]: $\mathscr{A} = (\mathscr{A}_1, \mathscr{A}_2)$ **by** (*cases* $\mathscr{A}$)
from *lossless* **have** *lossless1* [*simp*]: $\bigwedge pk.\ \textit{lossless-gpv}\ \mathscr{I}\textit{-full}\ (\mathscr{A}_1\ pk)$
and *lossless2* [*simp*]: $\bigwedge \sigma\ \textit{cipher}.\ \textit{lossless-gpv}\ \mathscr{I}\textit{-full}\ (\mathscr{A}_2\ \sigma\ \textit{cipher})$
by(*auto simp add*: *ind-cpa.lossless-def*)

5.3 Recording adversary queries

As can be seen in Fig. 2, both the adversary and the encryption of the challenge ciphertext use the random oracle. The reduction, however, returns only the queries that the adversary makes to the oracle (in Fig. 3, *h* is generated independently of the random oracle). To bridge this gap, we introduce an *interceptor* between the adversary and the oracle that records all adversary's queries.

define *interceptor* :: *'grp set* $\Rightarrow$ *'grp* $\Rightarrow$ (*bitstring* $\times$ *'grp set*, -, -) *gpv*
where
interceptor σ *x* = (*do* {
h $\leftarrow$ *hash x*;
Done (*h*, *insert x* σ)
}) **for** σ *x*

We integrate this interceptor into the *game* using the *inline* function as illustrated in Fig. 4 and name the result $game_0$.

define $game_0$ **where**

[7] Later in the proof, we will often prove losslessness of the definitions in the proof. We will not show them in this document, but they are in the Isabelle sources from which this document is generated.

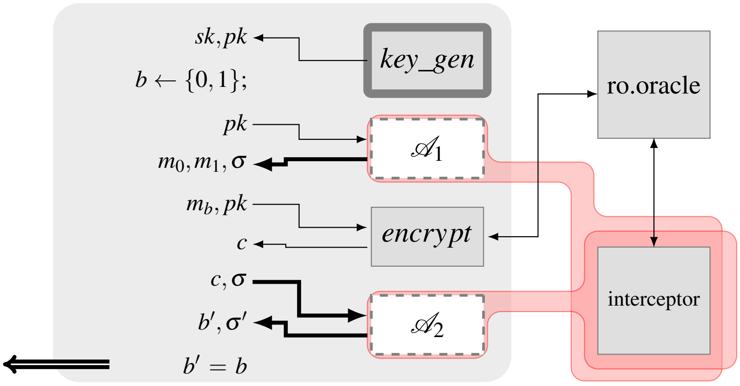

Figure 4: The IND-CPA game after expanding the key generation algorithm's definition and inlining the query-recording hash oracle. The red boxes represent the inline operator.

game$_0$ = *TRY do* {
(*pk*, -) ← *lift-spmf key-gen*;
(((*m*$_1$, *m*$_2$), σ), *s*) ← *inline interceptor* ($\mathscr{A}_1$ *pk*) {};
assert-gpv (*valid-plains m*$_1$ *m*$_2$);
b ← *lift-spmf coin-spmf*;
c ← *encrypt pk* (*if b then m*$_1$ *else m*$_2$);
(*b*′, *s*′) ← *inline interceptor* ($\mathscr{A}_2$ *c* σ) *s*;
Done (*b*′ = *b*)
} *ELSE lift-spmf coin-spmf*

We claim that the above modifications do not affect the output of the IND-CPA game at all. This might seem obvious since we are only logging the adversary's queries without modifying them. However, in a formal proof, this needs to be precisely justified.

More precisely, we have been very careful that the two games *game* $\mathscr{A}$ and *game*$_0$ have identical structure. They differ only in that *game*$_0$ uses the adversary (λ*pk*. *inline interceptor* ($\mathscr{A}_1$ *pk*) ∅, λ*cipher* σ. *inline interceptor* ($\mathscr{A}_2$ *cipher* σ)) instead of $\mathscr{A}$. The formal justification for this replacement happens in two steps:

1. We replace the oracle transformer *interceptor* with *id-oracle*, which merely passes queries and results to the oracle.

2. Inlining the identity oracle transformer *id-oracle* does not change an adversary and can therefore be dropped.

The first step is automated using Isabelle's Transfer package [9], which is based on Mitchell's representation independence [14]. The replacement is controlled by so-called transfer rules of the form *R x y* which indicates that *x* shall replace *y*; the correspondence relation *R* captures the kind of replacement. The *transfer* proof method then constructs a constraint system with one constraint for each atom in the proof goal where the correspondence relation and the replacement are unknown. It then tries to solve the constraint system using the rules that have been declared with the attribute [*transfer-rule*]. Atoms that do not have a suitable transfer rule are not changed and their correspondence relation is instantiated with the identity relation $(=)$.

The second step is automated using Isabelle's simplifier.

In the example, the crucial change happens in the state of the oracle transformer: *interceptor* records all queries in a set whereas *id-oracle* has no state, which is modelled with the singleton type *unit*. To capture the change, we define the correspondence relation *cr* on the states of the oracle transformers. (As we are in the process of adding this state, this state is irrelevant and *cr* is therefore always true. We nevertheless have to make an explicit definition such that Isabelle does not automatically beta-reduce terms, which would confuse *transfer*.) We then prove that it relates the initial states and that *cr* is a bisimulation relation for the two oracle transformers; see [2] for details. The bisimulation proof itself is automated, too: A bit of term rewriting (**unfolding**) makes the two oracle transformers structurally identical except for the state update function. Having proved that the state update function λ- σ. σ is a correct replacement for *insert* w.r.t. *cr*, the *transfer-prover* then lifts this replacement to the bisimulation rule. Here, *transfer-prover* is similar to *transfer* except that it works only for transfer rules and builds the constraint system only for the term to be replaced.

The theory source of this tutorial contains a step-by-step proof to illustrate how transfer works.

{ define *cr* :: *unit* ⇒ *'grp set* ⇒ *bool* **where** *cr* σ σ' = *True* **for** σ σ'
  have [*transfer-rule*]: *cr* () {} **by**(*simp add*: *cr-def*) — initial states
  have [*transfer-rule*]: ((=) ===> *cr* ===> *cr*) (λ- σ. σ) *insert* — state update
    by(*simp add*: *rel-fun-def cr-def*)
  have [*transfer-rule*]: — *cr* is a bisimulation for the oracle transformers
    (*cr* ===> (=) ===> *rel-gpv* (*rel-prod* (=) *cr*) (=)) *id-oracle interceptor*
   unfolding *interceptor-def*[*abs-def*] *id-oracle-def*[*abs-def*] *bind-gpv-Pause bind-rpv-Done*
    by *transfer-prover*
  have *ind-cpa.game* $\mathscr{A}$ = $game_0$ **unfolding** $game_0$*-def* $\mathscr{A}$ *ind-cpa.game.simps*
    by *transfer* (*simp add*: *bind-map-gpv o-def ind-cpa.game.simps split-def*)
}

5.4 Equational program transformations

Before we move on, we need to simplify $game_0$ and inline a few of the definitions. All these simplifications are equational program transformations, so the Isabelle simplifier can justify them. We combine the *interceptor* with the random oracle *oracle* into a new oracle *oracle'* with which the adversary interacts.

define *oracle'* :: *'grp set* × (*'grp* ⇀ *bitstring*) ⇒ *'grp* ⇒ -
where *oracle'* = (λ(*s*, σ) *x*. *do* {
  (*h*, σ') ← *case* σ *x of*

None ⇒ *do* {
bs ← *spmf-of-set* (*nlists UNIV len-plain*);
return-spmf (*bs*, $\sigma(x \mapsto bs)$) }
| *Some bs* ⇒ *return-spmf* (*bs*, σ);
return-spmf (*h*, *insert x s*, σ')
})
have ∗: *exec-gpv ro.oracle* (*inline interceptor* $\mathscr{A}$ *s*) σ =
map-spmf ($\lambda(a, b, c). ((a, b), c)$) (*exec-gpv oracle*′ $\mathscr{A}$ (*s*, σ)) **for** $\mathscr{A}$ σ *s*
by(*simp add*: *interceptor-def oracle*′*-def ro.oracle-def Let-def*
exec-gpv-inline exec-gpv-bind o-def split-def cong del: *option.case-cong-weak*)

We also want to inline the key generation and encryption algorithms, push the *TRY _ ELSE _* towards the assertion (which is possible because the adversary is lossless by assumption), and rearrange the samplings a bit. The latter is automated using *monad-normalisation* [17].[8]

have *game*$_0$: *run-gpv ro.oracle game*$_0$ *ro.initial* = *do* {
x ← *sample-uniform* (*order* $\mathscr{G}$);
y ← *sample-uniform* (*order* $\mathscr{G}$);
b ← *coin-spmf*;
(((*msg1*, *msg2*), σ), (*s*, *s-h*)) ←
exec-gpv oracle′ ($\mathscr{A}_1$ (**g** [^] *x*)) ({}, *ro.initial*);
TRY do {
- :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
(*h*, *s-h*′) ← *ro.oracle s-h* (**g** [^] (*x* ∗ *y*));
let cipher = (**g** [^] *y*, *h* [⊕] (*if b then msg1 else msg2*));
(*b*′, (*s*′, *s-h*′′)) ← *exec-gpv oracle*′ ($\mathscr{A}_2$ *cipher* σ) (*s*, *s-h*′);
return-spmf (*b*′ = *b*)
} *ELSE do* {
b ← *coin-spmf*;
return-spmf b
}
}
including *monad-normalisation*
by(*simp add*: *game*$_0$*-def key-gen-def encrypt-def* ∗ *exec-gpv-bind bind-map-spmf assert-spmf-def*
try-bind-assert-gpv try-gpv-bind-lossless split-def o-def if-distribs lcdh.nat-pow-pow)

This call to Isabelle's simplifier may look complicated at first, but it can be constructed incrementally by adding a few theorems and looking at the resulting goal state and searching for suitable theorems using **find-theorems**. As always in Isabelle, some intuition and knowledge about the library of lemmas is crucial.

- We knew that the definitions *game*$_0$*-def*, *key-gen-def*, and *encrypt-def* should be unfolded, so they are added first to the simplifier's set of rewrite rules.

[8] The tool *monad-normalisation* augments Isabelle's simplifier with a normalization procedure for commutative monads based on higher-order ordered rewriting. It can also commute across control structures like *if* and *case*. Although it is not complete as a decision procedure (as the normal forms are not unique), it usually works in practice.

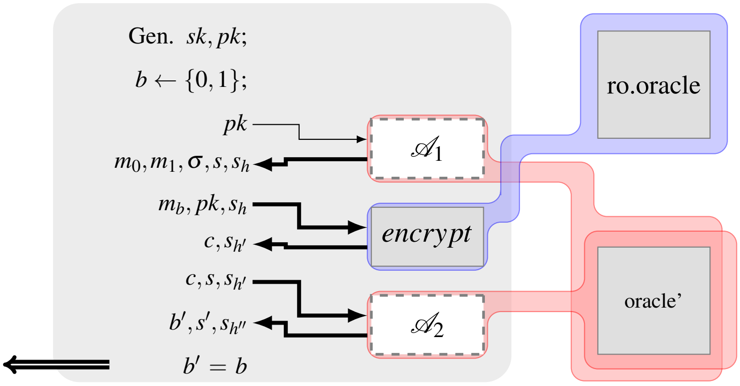

Figure 5: The IND-CPA game after flattening. The blue box around the encryption algorithm and the random oracle represents the expanded definition of them.

- The equations *exec-gpv-bind*, *try-bind-assert-gpv*, and *try-gpv-bind-lossless* ensure that the operator *exec-gpv*, which connects the $game_0$ with the random oracle, is distributed over the sequencing. Together with $*$, this gives the adversary access to $oracle'$ instead of the interceptor and the random oracle, and makes the call to the random oracle in the encryption of the chosen message explicit.

- The theorem *lcdh.nat-pow-pow* rewrites the iterated exponentiation $(\mathbf{g}\ [\hat{}]\ x)$ $[\hat{}]\ y$ to $\mathbf{g}\ [\hat{}]\ (x * y)$.

- The other theorems *bind-map-spmf*, *assert-spmf-def*, *split-def*, *o-def*, and *if-distribs* take care of all the boilerplate code that makes all these transformations type-correct. These theorems often have to be used together.

Note that the state of the oracle $oracle'$ is changed between $\mathscr{A}_1$ and $\mathscr{A}_2$. Namely, the random oracle's part *s-h* may change when the chosen message is encrypted, but the state that records the adversary's queries s is passed on unchanged.

5.5 Capturing a failure event

Suppose that two games behave the same except when a so-called failure event occurs [19]. Then the chance of an adversary distinguishing the two games is bounded by the probability of the failure event. In other words, the simulation of the reduction is allowed to break if the failure event occurs. In the running example, such an argument is a key step to derive the bound on the adversary's advantage. But to reason about failure events, we must first introduce them into the games we consider. This is because in CryptHOL, the probabilistic programs describe probability distributions over what they return (*return-spmf*). The variables that are used internally in the program are not accessible from the outside, i.e., there is no memory to which these are written. This has the advantage that we never have to worry about the names of the variables, e.g., to avoid clashes. The drawback is that we must explicitly introduce all the events that we are interested in.

Introducing a failure event into a game is straightforward. So far, the games *game* and $game_0$ simply denoted the probability distribution of whether the adversary has guessed right. For hashed Elgamal, the simulation breaks if the adversary queries the random oracle with the same query $\mathbf{g}$ [^] $(x * y)$ that is used for encrypting the chosen message m_b. So we simply change the return type of the game to return whether the adversary guessed right *and* whether the failure event has occurred. The next definition $game_1$ does so. (Recall that *oracle′* stores in its first state component *s* the queries by the adversary.) In preparation of the next reasoning step, we also split off the first two samplings, namely of *x* and *y*, and make them parameters of $game_1$.

define $game_1$ :: *nat* ⇒ *nat* ⇒ (*bool* × *bool*) *spmf*
where $game_1$ *x y* = *do* {
  b ← *coin-spmf*;
  $(((m_1, m_2), \sigma), (s, s\text{-}h))$ ← *exec-gpv oracle′* ($\mathscr{A}_1$ ($\mathbf{g}$ [^] *x*)) ({}, *ro.initial*);
  TRY do {
    - :: *unit* ← *assert-spmf* (*valid-plains* m_1 m_2);
    (*h*, *s-h′*) ← *ro.oracle s-h* ($\mathbf{g}$ [^] $(x * y)$);
    let c = ($\mathbf{g}$ [^] *y*, *h* [⊕] (*if b then* m_1 *else* m_2));
    (*b′*, (*s′*, *s-h″*)) ← *exec-gpv oracle′* ($\mathscr{A}_2$ *c* σ) (*s*, *s-h′*);
    return-spmf (*b′* = *b*, $\mathbf{g}$ [^] $(x * y) \in s'$)
  } *ELSE do* {
    b ← *coin-spmf*;
    return-spmf (*b*, $\mathbf{g}$ [^] $(x * y) \in s$)
  }
} **for** *x y*

It is easy to prove that $game_0$ combined with the random oracle is a projection of $game_1$ with the sampling added, as formalized in $game_0$-$game_1$.

let *?sample* = λ*f* :: *nat* ⇒ *nat* ⇒ - *spmf*. *do* {
  x ← *sample-uniform* (*order* $\mathscr{G}$);
  y ← *sample-uniform* (*order* $\mathscr{G}$);
  f x y }
have $game_0$-$game_1$:
  run-gpv ro.oracle $game_0$ *ro.initial* = *map-spmf fst* (*?sample* $game_1$)
  by(*simp add*: $game_0$ $game_1$*-def o-def split-def map-try-spmf map-scale-spmf*)

5.6 Game hop based on a failure event

A game hop based on a failure event changes one game into another such that they behave identically unless the failure event occurs. The *fundamental-lemma* bounds the absolute difference between the two games by the probability of the failure event. In the running example, we would like to avoid querying the random oracle when encrypting the chosen message. The next game $game_2$ is identical except that the call to the random oracle *oracle* is replaced with sampling a random bitstring.[9]

define $game_2$ :: *nat* ⇒ *nat* ⇒ (*bool* × *bool*) *spmf*
where $game_2$ *x y* = *do* {
  b ← *coin-spmf*;
  $(((m_1, m_2), \sigma), (s, s\text{-}h)) \leftarrow$ *exec-gpv oracle′* ($\mathscr{A}_1$ (**g** [^] *x*)) ({}, *ro.initial*);
  TRY do {
    - :: *unit* ← *assert-spmf* (*valid-plains* m_1 m_2);
    h ← *spmf-of-set* (*nlists UNIV len-plain*);
    — We do not query the random oracle for **g** [^] $(x * y)$, but instead sample a random bitstring *h* directly. So the rest differs from $game_1$ only if the adversary queries **g** [^] $(x * y)$.
    let cipher = (**g** [^] *y*, *h* [⊕] (*if b then* m_1 *else* m_2));
    $(b', (s', s\text{-}h')) \leftarrow$ *exec-gpv oracle′* ($\mathscr{A}_2$ *cipher* σ) (*s*, *s-h*);
    return-spmf ($b' = b$, **g** [^] $(x * y) \in s'$)
  } *ELSE do* {
    b ← *coin-spmf*;
    return-spmf (*b*, **g** [^] $(x * y) \in s$)
  }
} **for** *x y*

To apply the *fundamental-lemma*, we first have to prove that the two games are indeed the same except when the failure event occurs.

have *rel-spmf* $(\lambda(win, bad)\ (win', bad').\ bad = bad' \wedge (\neg\ bad' \longrightarrow win = win'))$ ($game_2$ *x y*) ($game_1$ *x y*) **for** *x y*
proof −

This proof requires two invariants on the state of *oracle′*. First, $s = dom\ s\text{-}h$. Second, *s* only becomes larger. The next two statements capture the two invariants:

interpret *inv-oracle′*: *callee-invariant-on oracle′* $(\lambda(s, s\text{-}h).\ s = dom\ s\text{-}h)$ $\mathscr{I}$*-full*
  by *unfold-locales*(*auto simp add*: *oracle′-def split*: *option.split-asm if-split*)
interpret *bad*: *callee-invariant-on oracle′* $(\lambda(s, \text{-}).\ z \in s)$ $\mathscr{I}$*-full* **for** *z*
  by *unfold-locales*(*auto simp add*: *oracle′-def*)

First, we identify a bisimulation relation *?X* between the different states of *oracle′* for the second phase of the game. Namely, the invariant $s = dom\ s\text{-}h$ holds, the set of queries are the same, and the random oracle's state (a map from queries to responses) differs only at the point **g** [^] $(x * y)$.

let $?X = \lambda(s, s\text{-}h)\ (s', s\text{-}h').\ s = dom\ s\text{-}h \wedge s' = s \wedge s\text{-}h = s\text{-}h'(\mathbf{g}\ [\string^]\ (x * y) := None)$

Then, we can prove that *?X* really is a bisimulation for *oracle′* except when the failure event occurs. The next statement expresses this.

let $?bad = \lambda(s, s\text{-}h).\ \mathbf{g}\ [\string^]\ (x * y) \in s$
let $?R = (\lambda(a, s1')\ (b, s2').\ ?bad\ s1' = ?bad\ s2' \wedge (\neg\ ?bad\ s2' \longrightarrow a = b \wedge ?X\ s1'\ s2'))$
have *bisim*: *rel-spmf ?R* (*oracle′ s1 plain*) (*oracle′ s2 plain*)

[9] In Shoup's terminology [19], such a step makes (a gnome sitting inside) the random oracle forgetting the query.

if *?X s1 s2* **for** *s1 s2 plain* **using** *that*
by(*auto split*: *prod.splits intro*!: *rel-spmf-bind-reflI simp add*: *oracle′-def rel-spmf-return-spmf2 fun-upd-twist split*: *option.split dest*!: *fun-upd-eqD*)
have *inv*: *callee-invariant oracle′ ?bad*
— Once the failure event has happened, it will not be forgotten any more.
by(*unfold-locales*)(*auto simp add*: *oracle′-def split*: *option.split-asm*)

Now we are ready to prove that the two games $game_1$ and $game_2$ are sufficiently similar. The Isar proof now switches into an **apply** script that manipulates the goal state directly. This is sometimes convenient when it would be too cumbersome to spell out every intermediate goal state.

show *?thesis*
unfolding $game_1$*-def* $game_2$*-def*
— Peel off the first phase of the game using the structural decomposition rules *rel-spmf-bind-reflI* and *rel-spmf-try-spmf*.
apply(*clarsimp intro*!: *rel-spmf-bind-reflI simp del*: *bind-spmf-const*)
apply(*rule rel-spmf-try-spmf*)
subgoal *TRY* **for** $b\ m_1\ m_2\ \sigma\ s$ *s-h*
apply(*rule rel-spmf-bind-reflI*)
— Exploit that in the first phase of the game, the set *s* of queried strings and the map of the random oracle *s-h* are updated in lock step, i.e., $s = dom\ s\text{-}h$.
apply(*drule inv-oracle′.exec-gpv-invariant*; *clarsimp*)
— Has the adversary queried the random oracle with **g** [^] $(x * y)$ during the first phase?
apply(*cases* **g** [^] $(x * y) \in s$)
subgoal *True* — Then the failure event has already happened and there is nothing more to do. We just have to prove that the two games on both sides terminate with the same probability.
by(*auto intro*!: *rel-spmf-bindI1 rel-spmf-bindI2 lossless-exec-gpv*[**where** $\mathscr{I}=\mathscr{I}$*-full*] *dest*!: *bad.exec-gpv-invariant*)
subgoal *False* — Then let's see whether the adversary queries **g** [^] $(x * y)$ in the second phase. Thanks to *ro.fresh*, the call to the random oracle simplifies to sampling a random bitstring.
apply(*clarsimp iff del*: *domIff simp add*: *domIff ro.fresh intro*!: *rel-spmf-bind-reflI*)
apply(*rule rel-spmf-bindI*[**where** *R=?R*])
— The lemma *exec-gpv-oracle-bisim-bad-full* lifts the bisimulation for *oracle′* to the adversary $\mathscr{A}_2$ interacting with *oracle′*.
apply(*rule exec-gpv-oracle-bisim-bad-full*[*OF - - bisim inv inv*])
apply(*auto simp add*: *fun-upd-idem*)
done
done
subgoal *ELSE* **by**(*rule rel-spmf-reflI*) *clarsimp*
done
qed

Now we can add the sampling of x and y in front of $game_1$ and $game_2$, apply the *fundamental-lemma*.

hence *rel-spmf* $(\lambda(win, bad)\ (win', bad').\ (bad \longleftrightarrow bad') \wedge (\neg\ bad' \longrightarrow win \longleftrightarrow win'))$ $(?sample\ game_2)$ $(?sample\ game_1)$
by(*intro rel-spmf-bind-reflI*)

hence $|$*measure* (*measure-spmf* (*?sample game*$_2$)) $\{$(*win*, -). *win*$\}$ $-$ *measure* (*measure-spmf* (*?sample game*$_1$)) $\{$(*win*, -). *win*$\}|$
$\leq$ *measure* (*measure-spmf* (*?sample game*$_2$)) $\{$(-, *bad*). *bad*$\}$
unfolding *split-def* **by**(*rule fundamental-lemma*)
moreover

The *fundamental-lemma* is written in full generality for arbitrary events, i.e., sets of elementary events. But in this formalization, the events of interest (correct guess and failure) are elementary events. We therefore transform the above statement to measure the probability of elementary events using *spmf*.

have *measure* (*measure-spmf* (*?sample game*$_2$)) $\{$(*win*, -). *win*$\}$ = *spmf* (*map-spmf fst* (*?sample game*$_2$)) *True*
and *measure* (*measure-spmf* (*?sample game*$_1$)) $\{$(*win*, -). *win*$\}$ = *spmf* (*map-spmf fst* (*?sample game*$_1$)) *True*
and *measure* (*measure-spmf* (*?sample game*$_2$)) $\{$(-, *bad*). *bad*$\}$ = *spmf* (*map-spmf snd* (*?sample game*$_2$)) *True*
unfolding *spmf-conv-measure-spmf measure-map-spmf* **by**(*auto simp add*: *vimage-def split-def*)
ultimately have *hop12*:
$|$*spmf* (*map-spmf fst* (*?sample game*$_2$)) *True* $-$ *spmf* (*map-spmf fst* (*?sample game*$_1$)) *True*$|$
$\leq$ *spmf* (*map-spmf snd* (*?sample game*$_2$)) *True*
by *simp*

5.7 Optimistic sampling: the one-time-pad

This step is based on the one-time-pad, which is an instance of optimistic sampling. If two runs of the two games in an optimistic sampling step would use the same random bits, then their results would be different. However, if the adversary's choices are independent of the random bits, we may relate runs that use different random bits, as in the end, only the probabilities have to match. The previous game hop from *game*$_1$ to *game*$_2$ made the oracle's responses in the second phase independent from the encrypted ciphertext. So we can now change the bits used for encrypting the chosen message and thereby make the ciphertext independent of the message.

To that end, we parametrize *game*$_2$ by the part that does the optimistic sampling and call this parametrized version *game*$_3$.

define *game*$_3$ :: (*bool* $\Rightarrow$ *bitstring* $\Rightarrow$ *bitstring* $\Rightarrow$ *bitstring spmf*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ (*bool* $\times$ *bool*) *spmf*
where *game*$_3$ *f x y* = *do* $\{$
b $\leftarrow$ *coin-spmf*;
$(((m_1, m_2), \sigma), (s, s\text{-}h)) \leftarrow$ *exec-gpv oracle'* ($\mathscr{A}_1$ (**g** [^] *x*)) ($\{\}$, *ro.initial*);
TRY do $\{$
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains* m_1 m_2);
h' $\leftarrow$ *f b* m_1 m_2;
let cipher = (**g** [^] *y*, h');
$(b', (s', s\text{-}h')) \leftarrow$ *exec-gpv oracle'* ($\mathscr{A}_2$ *cipher* σ) (*s*, *s-h*);

return-spmf $(b' = b, \mathbf{g}\ [\hat{}]\ (x * y) \in s')$
} *ELSE do* {
b ← *coin-spmf*;
return-spmf $(b, \mathbf{g}\ [\hat{}]\ (x * y) \in s)$
}
} **for** *f x y*

Clearly, if we plug in the appropriate function *?f*, then we get $game_2$:

let $?f = \lambda b\ m_1\ m_2.\ map\text{-}spmf\ (\lambda h.\ (if\ b\ then\ m_1\ else\ m_2)\ [\oplus]\ h)\ (spmf\text{-}of\text{-}set\ (nlists\ UNIV\ len\text{-}plain))$
have $game_2\text{-}game_3$: $game_2\ x\ y = game_3\ ?f\ x\ y$ **for** *x y*
by(*simp add*: $game_2$-*def* $game_3$-*def Let-def bind-map-spmf xor-list-commute o-def*)

CryptHOL's *one-time-pad* lemma now allows us to remove the exclusive or with the chosen message, because the resulting distributions are the same. The proof is slightly non-trivial because the one-time-pad lemma holds only if the xor'ed bitstrings have the right length, which the assertion *valid-plains* ensures. The congruence rules *try-spmf-cong bind-spmf-cong* [*OF refl*] *if-cong* [*OF refl*] extract this information from the program of the game.

let $?f' = \lambda b\ m_1\ m_2.\ spmf\text{-}of\text{-}set\ (nlists\ UNIV\ len\text{-}plain)$
have $game_3$: $game_3\ ?f\ x\ y = game_3\ ?f'\ x\ y$ **for** *x y*
by(*auto intro*!: *try-spmf-cong bind-spmf-cong*[*OF refl*] *if-cong*[*OF refl*]
simp add: $game_3$-*def split-def one-time-pad valid-plains-def*
simp del: *map-spmf-of-set-inj-on bind-spmf-const split*: *if-split*)

The rest of the proof consists of simplifying $game_3\ ?f'$. The steps are similar to what we have shown before, so we do not explain them in detail. The interested reader can look at them in the theory file from which this document was generated. At a high level, we see that there is no need to track the adversary's queries in $game_2$ or $game_3$ any more because this information is already stored in the random oracle's state. So we change the *oracle'* back into *oracle* using the Transfer package. With a bit of rewriting, the result is then the *game* for the adversary *elgamal-reduction* $\mathscr{A}$. Moreover, the guess b' of the adversary is independent of *b* in $game_3\ ?f$, so the first boolean returned by $game_3\ ?f'$ is just a coin flip.

have $game_3$-*bad*: $map\text{-}spmf\ snd\ (?sample\ (game_3\ ?f')) = lcdh.game\ (elgamal\text{-}reduction\ \mathscr{A})$
have $game_3$-*guess*: $map\text{-}spmf\ fst\ (game_3\ ?f'\ x\ y) = coin\text{-}spmf$ **for** *x y*

5.8 Combining several game hops

Finally, we combine all the (in)equalities of the previous steps to obtain the desired bound using the lemmas for reasoning about reals from Isabelle's library.

have $ind\text{-}cpa.advantage\ (ro.oracle, ro.initial)\ \mathscr{A} = |spmf\ (map\text{-}spmf\ fst\ (?sample\ game_1))\ True - 1\ /\ 2|$
using $ind\text{-}cpa\text{-}game\text{-}eq\text{-}game_0$ **by**(*simp add*: $game_0$-$game_1$ *o-def*)

also have … = |*1 / 2* − *spmf* (*map-spmf fst* (*?sample* $game_1$)) *True*|
by(*simp add*: *abs-minus-commute*)
also have *1 / 2* = *spmf* (*map-spmf fst* (*?sample* $game_2$)) *True*
by(*simp add*: $game_2$-$game_3$ $game_3$ *o-def* $game_3$-*guess spmf-of-set*)
also have |… − *spmf* (*map-spmf fst* (*?sample* $game_1$)) *True*| ≤ *spmf* (*map-spmf snd* (*?sample* $game_2$)) *True*
by(*rule hop12*)
also have … = *lcdh.advantage* (*elgamal-reduction* $\mathscr{A}$)
by(*simp add*: $game_2$-$game_3$ $game_3$ $game_3$-*bad lcdh.advantage-def o-def del*: *map-bind-spmf*)
finally show *?thesis* **.**

This completes the concrete proof and we can end the locale *hashed-elgamal*.

qed

end

6 Asymptotic security

An asymptotic security statement can be easily derived from a concrete security theorem. This is done in two steps: First, we have to introduce a security parameter η into the definitions and assumptions. Only then can we state asymptotic security. The proof is easy given the concrete security theorem.

6.1 Introducing a security parameter

Since all our definitions were done in locales, it is easy to introduce a security parameter after the fact. To that end, we define copies of all locales where their parameters now take the security parameter as an additional argument. We illustrate it for the locale *ind-cpa-pk*.

The **sublocale** command brings all the definitions and theorems of the original *ind-cpa-pk* into the copy and adds the security parameter where necessary. The type *security* is a synonym for *nat*.

locale *ind-cpa-pk′* =
fixes *key-gen* :: *security* ⇒ (*′pubkey* × *′privkey*, *′query*, *′response*) *gpv*
and *encrypt* :: *security* ⇒ *′pubkey* ⇒ *′plain* ⇒ (*′cipher*, *′query*, *′response*) *gpv*
and *decrypt* :: *security* ⇒ *′privkey* ⇒ *′cipher* ⇒ (*′plain*, *′query*, *′response*) *gpv*
and *valid-plains* :: *security* ⇒ *′plain* ⇒ *′plain* ⇒ *bool*
begin
sublocale *ind-cpa-pk key-gen* η *encrypt* η *decrypt* η *valid-plains* η **for** η **.**
end

We do so similarly for *list-cdh*, *random-oracle*, and *hashed-elgamal*.

locale *hashed-elgamal′* =
lcdh: *list-cdh′* $\mathscr{G}$ +

```
  ro: random-oracle' len-plain
  for 𝒢 :: security ⇒ 'grp cyclic-group
  and len-plain :: security ⇒ nat
begin
sublocale hashed-elgamal 𝒢 η len-plain η for η ..
```

6.2 Asymptotic security statements

For asymptotic security statements, CryptHOL defines the predicate *negligible*. It states that the given real-valued function approaches 0 faster than the inverse of any polynomial. A concrete security statement translates into an asymptotic one as follows:

- All advantages in the bound become negligibility assumptions.
- All side conditions of the concrete security theorems remain assumptions, but wrapped into an *eventually* statement. This expresses that the side condition holds eventually, i.e., there is a security parameter from which on it holds.
- The conclusion is that the bounded advantage is *negligible*.

```
theorem asymptotic-security-elgamal:
  assumes negligible (λη. lcdh.advantage η (elgamal-reduction η (𝒜 η)))
    and eventually (λη. ind-cpa.lossless (𝒜 η)) at-top
  shows negligible (λη. ind-cpa.advantage η (ro.oracle η, ro.initial) (𝒜 η))
```

The proof is canonical, too: Using the lemmas about *negligible* and Eberl's library for asymptotic reasoning [6], we transform the asymptotic statement into a concrete one and then simply use the concrete security statement.

```
apply(rule negligible-mono[OF assms(1)])
apply(rule landau-o.big-mono)
apply(rule eventually-rev-mp[OF assms(2)])
apply(intro eventuallyI impI)
apply(simp del: ind-cpa.advantage.simps add: ind-cpa.advantage-nonneg lcdh.advantage-nonneg)
by(rule concrete-security-elgamal)

end
```

References

[1] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, Feb 2014.

[2] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. Crypthol: Game-based proofs in higher-order logic. *IACR Cryptology ePrint Archive*, 2017:753, 2017.

[3] 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.

[4] 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.

[5] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[6] M. Eberl. Landau symbols. *Archive of Formal Proofs*, Jul 2015. http://isa-afp.org/entries/Landau_Symbols.html, Formal proof development.

[7] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[8] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *ACM Symposium on Theory of Computing (STOC 1982), Proceedings*, pages 365–377, 1982.

[9] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs (CPP 2013), Proceedings*, pages 131–146, 2013.

[10] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. Dissertation, Technische Universität München, 2009.

[11] A. Lochbihler. Probabilistic functions and cryptographic oracles in higher order logic. In *European Symposium on Programming (ESOP 2016), Proceedings*, pages 503–531, 2016.

[12] A. Lochbihler. Crypthol. *Archive of Formal Proofs*, May 2017. http://isa-afp.org/entries/CryptHOL.html, Formal proof development.

[13] A. Lochbihler, S. R. Sefidgar, and B. Bhatt. Game-based cryptography in hol. *Archive of Formal Proofs*, 2017. http://isa-afp.org/entries/Game_Based_Crypto.shtml, Formal proof development.

[14] J. C. Mitchell. Representation independence and data abstraction. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1986), Proceedings*, pages 263–276, 1986.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

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

[17] J. Schneider, M. Eberl, and A. Lochbihler. Monad normalisation. *Archive of Formal Proofs*, May 2017. http://isa-afp.org/entries/Monad_Normalisation.html, Formal proof development.

[18] G. Shafi and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

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

[20] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989), Proceedings*, pages 60–76, 1989.

[21] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL 1999), Proceedings*, pages 167–183, 1999.