

# Game-based cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar and Bhargav Bhatt

September 23, 2021

## 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

<b>1</b>	<b>Specifying security using games</b>	<b>3</b>
1.1	The DDH game . . . . .	3
1.2	The LCDH game . . . . .	4
1.3	The IND-CCA2 game for public-key encryption . . . . .	4
1.3.1	Single-user setting . . . . .	6
1.3.2	Multi-user setting . . . . .	7
1.4	The IND-CCA2 security for symmetric encryption schemes . . . . .	8
1.5	The IND-CPA game for symmetric encryption schemes . . . . .	10
1.6	The IND-CPA game for public-key encryption with oracle access . . . . .	11
1.7	The IND-CPA game (public key, single instance) . . . . .	13
1.8	Strongly existentially unforgeable signature scheme . . . . .	14
1.8.1	Single-user setting . . . . .	15
1.8.2	Multi-user setting . . . . .	16
1.9	Pseudo-random function . . . . .	17
1.10	Pseudo-random function . . . . .	18

1.11	Random permutation . . . . .	19
1.12	Reducing games with many adversary guesses to games with single guesses . . . . .	20
1.13	Unpredictable function . . . . .	27
<b>2</b>	<b>Cryptographic constructions and their security</b>	<b>29</b>
2.1	Elgamal encryption scheme . . . . .	29
2.2	Hashed Elgamal in the Random Oracle Model . . . . .	32
2.3	The random-permutation random-function switching lemma . . . . .	40
2.4	Extending the input length of a PRF using a universal hash function . . . . .	44
2.5	IND-CPA from PRF . . . . .	51
2.6	IND-CCA from a PRF and an unpredictable function . . . . .	62
<b>A</b>	<b>Tutorial Introduction to CryptHOL</b>	<b>81</b>
<b>3</b>	<b>Introduction</b>	<b>81</b>
3.1	Getting started . . . . .	82
3.2	Getting started . . . . .	83
<b>4</b>	<b>Modelling cryptography using CryptHOL</b>	<b>83</b>
4.1	Security notions without oracles: the CDH assumption . . . . .	83
4.2	A Random Oracle . . . . .	86
4.3	Cryptographic concepts: public-key encryption . . . . .	87
4.4	Security notions with oracles: IND-CPA security . . . . .	88
4.5	Concrete cryptographic constructions: the hashed ElGamal encryption scheme . . . . .	90
<b>5</b>	<b>Cryptographic proofs in CryptHOL</b>	<b>93</b>
5.1	The reduction . . . . .	93
5.2	Concrete security statement . . . . .	95
5.3	Recording adversary queries . . . . .	96
5.4	Equational program transformations . . . . .	98
5.5	Capturing a failure event . . . . .	100
5.6	Game hop based on a failure event . . . . .	101
5.7	Optimistic sampling: the one-time-pad . . . . .	104
5.8	Combining several game hops . . . . .	105
<b>6</b>	<b>Asymptotic security</b>	<b>106</b>
6.1	Introducing a security parameter . . . . .	106
6.2	Asymptotic security statements . . . . .	107

# 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'  $\Rightarrow$  'grp'  $\Rightarrow$  'grp'  $\Rightarrow$  bool spmf
```

```
definition ddh-0 :: 'grp adversary  $\Rightarrow$  bool spmf  
where ddh-0  $\mathcal{A} =$  do {  
   $x \leftarrow$  sample-uniform (order  $\mathcal{G}$ );  
   $y \leftarrow$  sample-uniform (order  $\mathcal{G}$ );  
   $\mathcal{A}$  ( $\mathbf{g} [\wedge] x$ ) ( $\mathbf{g} [\wedge] y$ ) ( $\mathbf{g} [\wedge] (x * y)$ )  
}
```

```
definition ddh-1 :: 'grp adversary  $\Rightarrow$  bool spmf  
where ddh-1  $\mathcal{A} =$  do {  
   $x \leftarrow$  sample-uniform (order  $\mathcal{G}$ );  
   $y \leftarrow$  sample-uniform (order  $\mathcal{G}$ );  
   $z \leftarrow$  sample-uniform (order  $\mathcal{G}$ );  
   $\mathcal{A}$  ( $\mathbf{g} [\wedge] x$ ) ( $\mathbf{g} [\wedge] y$ ) ( $\mathbf{g} [\wedge] z$ )  
}
```

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

```
definition lossless :: 'grp adversary  $\Rightarrow$  bool  
where lossless  $\mathcal{A} \longleftrightarrow (\forall \alpha \beta \gamma. \text{lossless-spmf } (\mathcal{A} \alpha \beta \gamma))$ 
```

```
lemma lossless-ddh-0:  
   $[[ \text{lossless } \mathcal{A}; 0 < \text{order } \mathcal{G} ]]$   
   $\implies \text{lossless-spmf } (\text{ddh-0 } \mathcal{A})$   
by(auto simp add: lossless-def ddh-0-def split-def Let-def)
```

```
lemma lossless-ddh-1:  
   $[[ \text{lossless } \mathcal{A}; 0 < \text{order } \mathcal{G} ]]$   
   $\implies \text{lossless-spmf } (\text{ddh-1 } \mathcal{A})$   
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'  $\Rightarrow$  'grp'  $\Rightarrow$  'grp' set spmf*

**definition** *lcdh* :: *'grp adversary  $\Rightarrow$  bool spmf*

**where** *lcdh*  $\mathcal{A}$  = do {

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

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

$zs \leftarrow \mathcal{A}$  ( $\mathbf{g} [\wedge] x$ ) ( $\mathbf{g} [\wedge] y$ );

return-spmf ( $\mathbf{g} [\wedge] (x * y) \in zs$ )

}

**definition** *advantage* :: *'grp adversary  $\Rightarrow$  real*

**where** *advantage*  $\mathcal{A}$  = spmf (*lcdh*  $\mathcal{A}$ ) True

**definition** *lossless* :: *'grp adversary  $\Rightarrow$  bool*

**where** *lossless*  $\mathcal{A} \longleftrightarrow (\forall \alpha \beta. \text{lossless-spmf } (\mathcal{A} \alpha \beta))$

**lemma** *lossless-lcdh*:

$\llbracket \text{lossless } \mathcal{A}; 0 < \text{order } \mathcal{G} \rrbracket$

$\implies \text{lossless-spmf } (\text{lcdh } \mathcal{A})$

**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  $\Rightarrow$  ('ekey  $\times$  'dkey) spmf* — probabilistic

**and** *encrypt* :: *security  $\Rightarrow$  'ekey  $\Rightarrow$  'plain  $\Rightarrow$  'cipher spmf* — probabilistic

**and** *decrypt* :: *security  $\Rightarrow$  'dkey  $\Rightarrow$  'cipher  $\Rightarrow$  'plain option* — deterministic, but not used

**and** *valid-plain* :: *security  $\Rightarrow$  'plain  $\Rightarrow$  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')  $call_1 = unit + 'cipher' + 'plain' \times 'plain'$   
**type-synonym** ('ekey', 'plain', 'cipher')  $ret_1 = 'ekey' + 'plain' option + 'cipher'$

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

**lemma**  $oracle_1-simps$  [*simp*]:  
 $oracle_1 b \eta s (Inl x) = map-spmf (apfst Inl) (ekey-oracle \eta s x)$   
 $oracle_1 b \eta s (Inr (Inl y)) = map-spmf (apfst (Inr \circ Inl)) (decrypt-oracle \eta s y)$   
 $oracle_1 b \eta s (Inr (Inr z)) = map-spmf (apfst (Inr \circ Inr)) (encrypt-oracle b \eta s z)$   
**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')  $adversary_1 =$   
 $security \Rightarrow ('ekey', 'plain', 'cipher) adversary_1'$

**definition**  $ind-cca2_1 :: ('ekey, 'plain, 'cipher) adversary_1 \Rightarrow security \Rightarrow bool$  *spmf*  
**where**  
 $ind-cca2_1 \mathcal{A} \eta = TRY do \{$   
 $b \leftarrow coin-spmf;$   
 $(guess, s) \leftarrow exec-gpv (oracle_1 b \eta) (\mathcal{A} \eta) None;$   
 $return-spmf (guess = b)$   
 $\} ELSE coin-spmf$

**definition**  $advantage_1 :: ('ekey, 'plain, 'cipher) adversary_1 \Rightarrow advantage$   
**where**  $advantage_1 \mathcal{A} \eta = |spmf (ind-cca2_1 \mathcal{A} \eta) True - 1/2|$

**lemma**  $advantage_1-nonneg: advantage_1 \mathcal{A} \eta \geq 0$  **by** (*simp add: advantage\_1-def*)

**abbreviation**  $secure-for_1 :: ('ekey, 'plain, 'cipher) adversary_1 \Rightarrow bool$   
**where**  $secure-for_1 \mathcal{A} \equiv negligible (advantage_1 \mathcal{A})$

**definition**  $ibounded-by_1' :: ('ekey, 'plain, 'cipher) adversary_1' \Rightarrow nat \Rightarrow bool$   
**where**  $ibounded-by_1' \mathcal{A} q = interaction-any-bounded-by \mathcal{A} q$

**abbreviation**  $ibounded-by_1 :: ('ekey, 'plain, 'cipher) adversary_1 \Rightarrow (security \Rightarrow nat) \Rightarrow bool$   
**where**  $ibounded-by_1 \equiv rel-envir ibounded-by_1'$

**definition**  $lossless_1' :: ('ekey, 'plain, 'cipher) adversary_1' \Rightarrow bool$   
**where**  $lossless_1' \mathcal{A} = lossless-gpv \mathcal{I}\text{-full} \mathcal{A}$

**abbreviation**  $lossless_1 :: ('ekey, 'plain, 'cipher) adversary_1 \Rightarrow bool$   
**where**  $lossless_1 \equiv pred\text{-}envir\ lossless_1'$

**lemma**  $lossless\text{-}decrypt\text{-}oracle$  [simp]:  $lossless\text{-}spmf\ (decrypt\text{-}oracle\ \eta\ \sigma\ cipher)$   
**by**(cases ( $\eta, \sigma, cipher$ ) rule:  $decrypt\text{-}oracle.cases$ ) simp-all

**lemma**  $lossless\text{-}ekey\text{-}oracle$  [simp]:  
 $lossless\text{-}spmf\ (ekey\text{-}oracle\ \eta\ \sigma\ x) \longleftrightarrow (\sigma = None \longrightarrow lossless\text{-}spmf\ (key\text{-}gen\ \eta))$   
**by**(cases ( $\eta, \sigma, x$ ) rule:  $ekey\text{-}oracle.cases$ )(auto)

**lemma**  $lossless\text{-}encrypt\text{-}oracle$  [simp]:  
 $[\ \sigma = None \implies lossless\text{-}spmf\ (key\text{-}gen\ \eta);$   
 $\ \wedge ekey\ m.\ valid\text{-}plain\ \eta\ m \implies lossless\text{-}spmf\ (encrypt\ \eta\ ekey\ m) ]$   
 $\implies lossless\text{-}spmf\ (encrypt\text{-}oracle\ b\ \eta\ \sigma\ (m0, m1)) \longleftrightarrow valid\text{-}plain\ \eta\ m0 \wedge valid\text{-}plain\ \eta\ m1$   
**apply**(cases ( $b, \eta, \sigma, (m0, m1)$ ) rule:  $encrypt\text{-}oracle.cases$ )  
**apply**(auto simp add:  $split\text{-}beta$  dest:  $lossless\text{-}spmfD\text{-}set\text{-}spmf\text{-}nonempty$  split:  $if\text{-}split\text{-}asm$ )  
**done**

### 1.3.2 Multi-user setting

**definition**  $oracle_n :: bool \Rightarrow security$   
 $\Rightarrow ('i \Rightarrow ('ekey, 'dkey, 'cipher) state\text{-}oracle, 'i \times ('plain, 'cipher) call_1, ('ekey, 'plain, 'cipher) ret_1) oracle'$   
**where**  $oracle_n\ b\ \eta = family\text{-}oracle\ (\lambda\cdot. oracle_1\ b\ \eta)$

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

**type-synonym**  $('i, 'ekey, 'plain, 'cipher) adversary_n' =$   
 $(bool, 'i \times ('plain, 'cipher) call_1, ('ekey, 'plain, 'cipher) ret_1) gpv$

**type-synonym**  $('i, 'ekey, 'plain, 'cipher) adversary_n =$   
 $security \Rightarrow ('i, 'ekey, 'plain, 'cipher) adversary_n'$

**definition**  $ind\text{-}cca2_n :: ('i, 'ekey, 'plain, 'cipher) adversary_n \Rightarrow security \Rightarrow bool\ smpf$   
**where**

$ind\text{-}cca2_n\ \mathcal{A}\ \eta = TRY\ do\ \{$   
 $\ b \leftarrow coin\text{-}spmf;$   
 $\ (guess, \sigma) \leftarrow exec\text{-}gpv\ (oracle_n\ b\ \eta)\ (\mathcal{A}\ \eta)\ (\lambda\cdot. None);$   
 $\ return\text{-}spmf\ (guess = b)$   
 $\} ELSE\ coin\text{-}spmf$

**definition**  $advantage_n :: ('i, 'ekey, 'plain, 'cipher) adversary_n \Rightarrow advantage$   
**where**  $advantage_n\ \mathcal{A}\ \eta = |smpf\ (ind\text{-}cca2_n\ \mathcal{A}\ \eta)\ True - 1/2|$

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

**abbreviation** *secure-for<sub>n</sub>* :: ('i, 'ekey, 'plain, 'cipher) adversary<sub>n</sub> ⇒ bool  
**where** *secure-for<sub>n</sub>*  $\mathcal{A} \equiv \text{negligible}(\text{advantage}_n \mathcal{A})$

**definition** *ibounded-by<sub>n</sub>'* :: ('i, 'ekey, 'plain, 'cipher) adversary<sub>n'</sub> ⇒ nat ⇒ bool  
**where** *ibounded-by<sub>n</sub>'*  $\mathcal{A} \ q = \text{interaction-any-bounded-by} \ \mathcal{A} \ q$

**abbreviation** *ibounded-by<sub>n</sub>* :: ('i, 'ekey, 'plain, 'cipher) adversary<sub>n</sub> ⇒ (security ⇒ nat) ⇒ bool  
**where** *ibounded-by<sub>n</sub>*  $\equiv \text{rel-envir } \text{ibounded-by}_n'$

**definition** *lossless<sub>n</sub>'* :: ('i, 'ekey, 'plain, 'cipher) adversary<sub>n'</sub> ⇒ bool  
**where** *lossless<sub>n</sub>'*  $\mathcal{A} = \text{lossless-gpv } \mathcal{I}\text{-full } \mathcal{A}$

**abbreviation** *lossless<sub>n</sub>* :: ('i, 'ekey, 'plain, 'cipher) adversary<sub>n</sub> ⇒ bool  
**where** *lossless<sub>n</sub>*  $\equiv \text{pred-envir } \text{lossless}_n'$

**definition** *cipher-queries* :: ('i ⇒ ('ekey, 'dkey, 'cipher) state-oracle) ⇒ 'cipher set  
**where** *cipher-queries* *ose* = ( $\bigcup_{(ciphers) \in \text{ran } \text{ose}. \text{set } \text{ciphers}}$ )

**lemma** *cipher-queriesI*:  
 $\llbracket \text{ose } n = \text{Some } (ek, dk, \text{ciphers}); x \in \text{set } \text{ciphers} \rrbracket \implies x \in \text{cipher-queries } \text{ose}$   
**by**(*auto simp add: cipher-queries-def ran-def*)

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

**lemma** *cipher-queries-updE*:  
**assumes**  $x \in \text{cipher-queries } (\text{ose}(n \mapsto (ek, dk, \text{ciphers})))$   
**obtains** (*old*)  $x \in \text{cipher-queries } \text{ose} \ x \notin \text{set } \text{ciphers} \mid$  (*new*)  $x \in \text{set } \text{ciphers}$   
**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  $\Rightarrow$  'message  $\Rightarrow$  'cipher spmf
  and decrypt :: 'key  $\Rightarrow$  'cipher  $\Rightarrow$  'message option
  and msg-predicate :: 'message  $\Rightarrow$  bool
begin

type-synonym ('message', 'cipher') adversary =
  (bool, 'message'  $\times$  'message' + 'cipher', 'cipher' option + 'message' option) gpv

definition oracle-encrypt :: 'key  $\Rightarrow$  bool  $\Rightarrow$  ('message  $\times$  'message, 'cipher option, 'cipher
set) callee
where
  oracle-encrypt k b L = ( $\lambda$ (msg1, msg0).
    (case msg-predicate msg1  $\wedge$  msg-predicate msg0 of
      True  $\Rightarrow$  do {
        c  $\leftarrow$  encrypt k (if b then msg1 else msg0);
        return-spmf (Some c, {c}  $\cup$  L)
      }
      | False  $\Rightarrow$  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  $\Rightarrow$  ('cipher, 'message option, 'cipher set) callee
where oracle-decrypt k L c = return-spmf (if c  $\in$  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  $\Rightarrow$  bool spmf
where
  game  $\mathcal{A}$  = do {
    key  $\leftarrow$  key-gen;
    b  $\leftarrow$  coin-spmf;
    (b', L')  $\leftarrow$  exec-gpv (oracle-encrypt key b  $\oplus_O$  oracle-decrypt key)  $\mathcal{A}$  {};
    return-spmf (b = b')
  }

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

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

end

end

```

```

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

```

## 1.5 The IND-CPA game for symmetric encryption schemes

```

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

```

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

```

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

```

```

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

```

```

definition ind-cpa :: ('plain, 'cipher, 'state) adversary ⇒ bool spmf
where
  ind-cpa  $\mathcal{A}$  = do {
    let ( $\mathcal{A}1$ ,  $\mathcal{A}2$ ) =  $\mathcal{A}$ ;
    key ← key-gen;
    b ← coin-spmf;
    (guess, -) ← exec-gpv (encrypt-oracle key) (do {
      ((m0, m1), σ) ←  $\mathcal{A}1$ ;
      if valid-plain m0 ∧ valid-plain m1 then do {
        cipher ← lift-spmf (encrypt key (if b then m0 else m1));
         $\mathcal{A}2$  cipher σ
      } else lift-spmf coin-spmf
    }) ();
    return-spmf (guess = b)
  }

```

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

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

**definition** *ibounded-by* :: ('plain, 'cipher, 'state) adversary  $\Rightarrow$  enat  $\Rightarrow$  bool  
**where**

*ibounded-by* =  $(\lambda(\mathcal{A}1, \mathcal{A}2) q.$   
 $(\exists q1 q2. \text{interaction-any-bounded-by } \mathcal{A}1 q1 \wedge (\forall \text{cipher } \sigma. \text{interaction-any-bounded-by}$   
 $(\mathcal{A}2 \text{ cipher } \sigma) q2) \wedge q1 + q2 \leq q))$

**lemma** *ibounded-byE* [*consumes 1, case-names ibounded-by, elim?*]:

**assumes** *ibounded-by* ( $\mathcal{A}1, \mathcal{A}2$ )  $q$   
**obtains**  $q1 q2$   
**where**  $q1 + q2 \leq q$   
**and** *interaction-any-bounded-by*  $\mathcal{A}1 q1$   
**and**  $\wedge \text{cipher } \sigma. \text{interaction-any-bounded-by } (\mathcal{A}2 \text{ cipher } \sigma) q2$   
**using** *assms by* (*auto simp add: ibounded-by-def*)

**lemma** *ibounded-byI* [*intro?*]:

$\llbracket \text{interaction-any-bounded-by } \mathcal{A}1 q1; \wedge \text{cipher } \sigma. \text{interaction-any-bounded-by } (\mathcal{A}2 \text{ cipher } \sigma) q2; q1 + q2 \leq q \rrbracket$   
 $\implies \text{ibounded-by } (\mathcal{A}1, \mathcal{A}2) q$   
**by** (*auto simp add: ibounded-by-def*)

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

**where** *lossless* =  $(\lambda(\mathcal{A}1, \mathcal{A}2). \text{lossless-gpv } \mathcal{I}\text{-full } \mathcal{A}1 \wedge (\forall \text{cipher } \sigma. \text{lossless-gpv } \mathcal{I}\text{-full}$   
 $(\mathcal{A}2 \text{ cipher } \sigma)))$

**end**

**end**

**theory** *IND-CPA-PK imports*

*CryptHOL.Computational-Model*

*CryptHOL.Negligible*

**begin**

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

**locale** *ind-cpa-pk* =

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

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

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

**fun** *ind-cpa* :: (*'pubkey, 'plain, 'cipher, 'call, 'ret, 'state*) *adversary ⇒ (bool, 'call, 'ret)*  
*gpv*

**where**

*ind-cpa* (*ℳ1, ℳ2*) = *TRY* *do* {  
 (*pk, sk*) ← *key-gen*;  
*b* ← *lift-spmf coin-spmf*;  
 ((*m0, m1*), *σ*) ← (*ℳ1 pk*);  
*assert-gpv (valid-plains m0 m1)*;  
*cipher* ← *aencrypt pk (if b then m0 else m1)*;  
*guess* ← *ℳ2 cipher σ*;  
*Done (guess = b)*  
 } *ELSE lift-spmf coin-spmf*

**definition** *advantage* :: (*'σ ⇒ 'call ⇒ ('ret × 'σ) spmf*) ⇒ *'σ ⇒ ('pubkey, 'plain, 'cipher,*  
*'call, 'ret, 'state) adversary ⇒ real*

**where** *advantage oracle σ ℳ* = |*spmf (run-gpv oracle (ind-cpa ℳ) σ) True* − 1/2|

**lemma** *advantage-nonneg: advantage oracle σ ℳ ≥ 0 by (simp add: advantage-def)*

**definition** *ibounded-by* :: (*'call ⇒ bool*) ⇒ (*'pubkey, 'plain, 'cipher, 'call, 'ret, 'state*)  
*adversary ⇒ enat ⇒ bool*

**where**

*ibounded-by consider* = ( $\lambda(\mathcal{A}1, \mathcal{A}2) q.$   
 ( $\exists q1 q2. (\forall pk. \text{interaction-bounded-by consider } (\mathcal{A}1 \text{ pk}) q1) \wedge (\forall \text{cipher } \sigma. \text{interaction-bounded-by consider } (\mathcal{A}2 \text{ cipher } \sigma) q2) \wedge q1 + q2 \leq q$ ))

**lemma** *ibounded-by'E [consumes 1, case-names ibounded-by', elim?]:*

**assumes** *ibounded-by consider* (*ℳ1, ℳ2*) *q*

**obtains** *q1 q2*

**where** *q1 + q2 ≤ q*

**and**  $\wedge pk. \text{interaction-bounded-by consider } (\mathcal{A}1 \text{ pk}) q1$

**and**  $\wedge \text{cipher } \sigma. \text{interaction-bounded-by consider } (\mathcal{A}2 \text{ cipher } \sigma) q2$

**using** *assms by (auto simp add: ibounded-by-def)*

**lemma** *ibounded-byI [intro?]:*

$\llbracket \wedge pk. \text{interaction-bounded-by consider } (\mathcal{A}1 \text{ pk}) q1; \wedge \text{cipher } \sigma. \text{interaction-bounded-by consider } (\mathcal{A}2 \text{ cipher } \sigma) q2; q1 + q2 \leq q \rrbracket$

$\implies \text{ibounded-by consider } (\mathcal{A}1, \mathcal{A}2) q$

**by** (*auto simp add: ibounded-by-def*)

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

**where**  $lossless = (\lambda (\mathcal{A}1, \mathcal{A}2). (\forall pk. lossless-gpv \mathcal{I}\text{-full} (\mathcal{A}1\ pk)) \wedge (\forall cipher\ \sigma. lossless-gpv \mathcal{I}\text{-full} (\mathcal{A}2\ cipher\ \sigma)))$

**end**

**end**

**theory** *IND-CPA-PK-Single imports*

*CryptHOL.Computational-Model*

**begin**

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

**locale** *ind-cpa =*

**fixes** *key-gen* :: ('pub-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* ( $\mathcal{A}1, \mathcal{A}2$ ) = *TRY* *do* {  
 (*pk, sk*) ← *key-gen*;  
 ((*m0, m1*),  $\sigma$ ) ←  $\mathcal{A}1\ pk$ ;  
 - :: *unit* ← *assert-spmf* (*valid-plains* *m0* *m1*);  
*b* ← *coin-spmf*;  
*cipher* ← *aencrypt* *pk* (*if* *b* *then* *m0* *else* *m1*);  
*b'* ←  $\mathcal{A}2\ cipher\ \sigma$ ;  
*return-spmf* (*b = b'*)  
 } *ELSE* *coin-spmf*

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

**definition** *advantage* :: ('pub-key, 'plain, 'cipher, 'state) *adversary* ⇒ *real*

**where** *advantage*  $\mathcal{A} = |spmf\ (ind-cpa\ \mathcal{A})\ True - 1/2|$

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

**where**

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

**lemma** *lossless-ind-cpa*:

$$\llbracket \text{lossless } \mathcal{A}; \text{lossless-spmf } (\text{key-gen}) \rrbracket \implies \text{lossless-spmf } (\text{ind-cpa } \mathcal{A})$$

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

**end**

**end**

**theory** *SUF-CMA imports*

*CryptHOL.Computational-Model*

*CryptHOL.Negligible*

*CryptHOL.Environment-Functor*

**begin**

## 1.8 Strongly existentially unforgeable signature scheme

**locale** *sig-scheme* =

**fixes** *key-gen* :: *security*  $\Rightarrow$  ('vkey  $\times$  'sigkey) *spmf*

**and** *sign* :: *security*  $\Rightarrow$  'sigkey  $\Rightarrow$  'message  $\Rightarrow$  'signature *spmf*

**and** *verify* :: *security*  $\Rightarrow$  'vkey  $\Rightarrow$  'message  $\Rightarrow$  'signature  $\Rightarrow$  *bool* — verification is deterministic

**and** *valid-message* :: *security*  $\Rightarrow$  'message  $\Rightarrow$  *bool*

**locale** *suf-cma* = *sig-scheme* +

**constrains** *key-gen* :: *security*  $\Rightarrow$  ('vkey  $\times$  'sigkey) *spmf*

**and** *sign* :: *security*  $\Rightarrow$  'sigkey  $\Rightarrow$  'message  $\Rightarrow$  'signature *spmf*

**and** *verify* :: *security*  $\Rightarrow$  'vkey  $\Rightarrow$  'message  $\Rightarrow$  'signature  $\Rightarrow$  *bool*

**and** *valid-message* :: *security*  $\Rightarrow$  'message  $\Rightarrow$  *bool*

**begin**

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

= ('vkey'  $\times$  'sigkey'  $\times$  ('message'  $\times$  'signature') *list*) *option*

**fun** *vkey-oracle* :: *security*  $\Rightarrow$  (('vkey, 'sigkey, 'message, 'signature) *state-oracle*, *unit*, 'vkey) *oracle'*

**where**

*vkey-oracle*  $\eta$  *None* - = *do* {

(*vkey*, *sigkey*)  $\leftarrow$  *key-gen*  $\eta$ ;

*return-spmf* (*vkey*, *Some* (*vkey*, *sigkey*, []))

}

|  $\wedge$  *log*. *vkey-oracle*  $\eta$  (*Some* (*vkey*, *sigkey*, *log*)) - = *return-spmf* (*vkey*, *Some* (*vkey*, *sigkey*, *log*))

**context notes** *bind-spmf-cong*[*fundef-cong*] **begin**

**function** *sign-oracle*  
 :: *security*  $\Rightarrow$  (('vkey, 'sigkey, 'message, 'signature) *state-oracle*, 'message, 'signature)  
*oracle'*  
**where**  
*sign-oracle*  $\eta$  *None* *m* = *do* { (-,  $\sigma$ )  $\leftarrow$  *vkey-oracle*  $\eta$  *None* (); *sign-oracle*  $\eta$   $\sigma$  *m* }  
|  $\wedge$  *log*. *sign-oracle*  $\eta$  (*Some* (*vkey*, *skey*, *log*)) *m* =  
(*if valid-message*  $\eta$  *m* *then do* {  
*sig*  $\leftarrow$  *sign*  $\eta$  *skey* *m*;  
*return-spmf* (*sig*, *Some* (*vkey*, *skey*, (*m*, *sig*) # *log*))  
} *else return-pmf None*)  
**by** *pat-completeness auto*  
**termination by** (*relation Wellfounded.measure* ( $\lambda$  ( $\eta$ ,  $\sigma$ , *m*). *case*  $\sigma$  *of None*  $\Rightarrow$  1 | -  $\Rightarrow$   
0)) *auto*  
**end**

**lemma** *lossless-vkey-oracle* [*simp*]:  
*lossless-spmf* (*vkey-oracle*  $\eta$   $\sigma$  *x*)  $\longleftrightarrow$  ( $\sigma = \text{None} \longrightarrow$  *lossless-spmf* (*key-gen*  $\eta$ ))  
**by** (*cases* ( $\eta$ ,  $\sigma$ , *x*) *rule: vkey-oracle.cases*) *auto*

**lemma** *lossless-sign-oracle* [*simp*]:  
 $\llbracket \sigma = \text{None} \Longrightarrow$  *lossless-spmf* (*key-gen*  $\eta$ );  
 $\wedge$  *skey* *m*. *valid-message*  $\eta$  *m*  $\Longrightarrow$  *lossless-spmf* (*sign*  $\eta$  *skey* *m*)  $\rrbracket$   
 $\Longrightarrow$  *lossless-spmf* (*sign-oracle*  $\eta$   $\sigma$  *m*)  $\longleftrightarrow$  *valid-message*  $\eta$  *m*  
**apply** (*cases* ( $\eta$ ,  $\sigma$ , *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*  $\eta$  (*Some* (*vkey*, *skey*, *log*)) *m*)  $\longleftrightarrow$  *lossless-spmf* (*sign*  $\eta$  *skey*  
*m*)  $\wedge$  *valid-message*  $\eta$  *m*  
**by** (*simp*)

### 1.8.1 Single-user setting

**type-synonym** 'message' *call*<sub>1</sub> = *unit* + 'message'  
**type-synonym** ('vkey', 'signature') *ret*<sub>1</sub> = 'vkey' + 'signature'

**definition** *oracle*<sub>1</sub> :: *security*  
 $\Rightarrow$  (('vkey, 'sigkey, 'message, 'signature) *state-oracle*, 'message *call*<sub>1</sub>, ('vkey, 'signature)  
*ret*<sub>1</sub>) *oracle'*  
**where** *oracle*<sub>1</sub>  $\eta$  = *vkey-oracle*  $\eta$   $\oplus_O$  *sign-oracle*  $\eta$

**lemma** *oracle*<sub>1</sub>-*simps* [*simp*]:  
*oracle*<sub>1</sub>  $\eta$  *s* (*Inl* *x*) = *map-spmf* (*apfst Inl*) (*vkey-oracle*  $\eta$  *s* *x*)  
*oracle*<sub>1</sub>  $\eta$  *s* (*Inr* *y*) = *map-spmf* (*apfst Inr*) (*sign-oracle*  $\eta$  *s* *y*)  
**by** (*simp-all add: oracle*<sub>1</sub>-*def*)

**type-synonym** ('vkey', 'message', 'signature') *adversary*<sub>1</sub>' =  
(('message'  $\times$  'signature'), 'message' *call*<sub>1</sub>, ('vkey', 'signature') *ret*<sub>1</sub>) *gpv*

**type-synonym** ('vkey', 'message', 'signature') adversary<sub>1</sub> =  
 security ⇒ ('vkey', 'message', 'signature') adversary<sub>1</sub>'

**definition** suf-cma<sub>1</sub> :: ('vkey', 'message', 'signature') adversary<sub>1</sub> ⇒ security ⇒ bool spmf  
**where**

```

∧ log. suf-cma1 ℳ η = do {
  ((m, sig), σ) ← exec-gpv (oracle1 η) (ℳ η) None;
  return-spmf (
    case σ of None ⇒ False
    | Some (vkey, skey, log) ⇒ verify η vkey m sig ∧ (m, sig) ∉ set log)
}

```

**definition** advantage<sub>1</sub> :: ('vkey', 'message', 'signature') adversary<sub>1</sub> ⇒ advantage  
**where** advantage<sub>1</sub> ℳ η = spmf (suf-cma<sub>1</sub> ℳ η) True

**lemma** advantage<sub>1</sub>-nonneg: advantage<sub>1</sub> ℳ η ≥ 0 **by** (simp add: advantage<sub>1</sub>-def pmf-nonneg)

**abbreviation** secure-for<sub>1</sub> :: ('vkey', 'message', 'signature') adversary<sub>1</sub> ⇒ bool  
**where** secure-for<sub>1</sub> ℳ ≡ negligible (advantage<sub>1</sub> ℳ)

**definition** ibounded-by<sub>1</sub>' :: ('vkey', 'message', 'signature') adversary<sub>1</sub>' ⇒ nat ⇒ bool  
**where** ibounded-by<sub>1</sub>' ℳ q = (interaction-any-bounded-by ℳ q)

**abbreviation** ibounded-by<sub>1</sub> :: ('vkey', 'message', 'signature') adversary<sub>1</sub> ⇒ (security ⇒ nat)  
 ⇒ bool  
**where** ibounded-by<sub>1</sub> ≡ rel-envir ibounded-by<sub>1</sub>'

**definition** lossless<sub>1</sub>' :: ('vkey', 'message', 'signature') adversary<sub>1</sub>' ⇒ bool  
**where** lossless<sub>1</sub>' ℳ = (lossless-gpv ℳ-full ℳ)

**abbreviation** lossless<sub>1</sub> :: ('vkey', 'message', 'signature') adversary<sub>1</sub> ⇒ bool  
**where** lossless<sub>1</sub> ≡ pred-envir lossless<sub>1</sub>'

## 1.8.2 Multi-user setting

**definition** oracle<sub>n</sub> :: security  
 ⇒ ('i ⇒ ('vkey', 'sigkey', 'message', 'signature') state-oracle, 'i × 'message call<sub>1</sub>, ('vkey',  
 'signature') ret<sub>1</sub>) oracle'  
**where** oracle<sub>n</sub> η = family-oracle (λ-. oracle<sub>1</sub> η)

**lemma** oracle<sub>n</sub>-apply [simp]:  
 oracle<sub>n</sub> η s (i, x) = map-spmf (apsnd (fun-upd s i)) (oracle<sub>1</sub> η (s i) x)  
**by** (simp add: oracle<sub>n</sub>-def)

**type-synonym** ('i, 'vkey', 'message', 'signature') adversary<sub>n</sub>' =  
 (('i × 'message' × 'signature'), 'i × 'message' call<sub>1</sub>, ('vkey', 'signature') ret<sub>1</sub>) gpv  
**type-synonym** ('i, 'vkey', 'message', 'signature') adversary<sub>n</sub> =  
 security ⇒ ('i, 'vkey', 'message', 'signature') adversary<sub>n</sub>'



**definition**  $\text{suf-cma}_n :: ('i, 'vkey, 'message, 'signature) \text{adversary}_n \Rightarrow \text{security} \Rightarrow \text{bool spmf}$   
**where**

```

 $\wedge \text{log. suf-cma}_n \mathcal{A} \eta = \text{do } \{$ 
   $((i, m, sig), \sigma) \leftarrow \text{exec-gpv } (\text{oracle}_n \eta) (\mathcal{A} \eta) (\lambda-. \text{None});$ 
   $\text{return-spmf } ($ 
     $\text{case } \sigma \text{ of None} \Rightarrow \text{False}$ 
     $| \text{Some } (vkey, skey, log) \Rightarrow \text{verify } \eta \text{ vkey } m \text{ sig} \wedge (m, sig) \notin \text{set log}$ 
   $)$ 

```

**definition**  $\text{advantage}_n :: ('i, 'vkey, 'message, 'signature) \text{adversary}_n \Rightarrow \text{advantage}$   
**where**  $\text{advantage}_n \mathcal{A} \eta = \text{spmf } (\text{suf-cma}_n \mathcal{A} \eta) \text{ True}$

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

**abbreviation**  $\text{secure-for}_n :: ('i, 'vkey, 'message, 'signature) \text{adversary}_n \Rightarrow \text{bool}$   
**where**  $\text{secure-for}_n \mathcal{A} \equiv \text{negligible } (\text{advantage}_n \mathcal{A})$

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

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

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

**abbreviation**  $\text{lossless}_n :: ('i, 'vkey, 'message, 'signature) \text{adversary}_n \Rightarrow \text{bool}$   
**where**  $\text{lossless}_n \equiv \text{pred-envir } \text{lossless}_n'$

**end**

**end**

**theory** *Pseudo-Random-Function* **imports**

*CryptHOL.Computational-Model*

**begin**

## 1.9 Pseudo-random function

**locale** *random-function* =

**fixes**  $p :: 'a \text{ spmf}$

**begin**

**type-synonym**  $(b, 'a) \text{ dict} = b \rightarrow 'a$

**definition**  $\text{random-oracle} :: (b, 'a) \text{ dict} \Rightarrow b \Rightarrow ('a \times (b, 'a) \text{ dict}) \text{ spmf}$   
**where**

*random-oracle*  $\sigma x =$   
 (case  $\sigma x$  of Some  $y \Rightarrow$  *return-spmf* ( $y, \sigma$ )  
 | None  $\Rightarrow p \gg\equiv (\lambda y. \text{return-spmf } (y, \sigma(x \mapsto y)))$ )

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

**lemma** *weight-random-oracle* [*simp*]:  
*weight-spmf*  $p = 1 \implies \text{weight-spmf } (\text{random-oracle } \sigma x) = 1$   
**by** (*simp add: random-oracle-def weight-bind-spmf o-def split: option.split*)

**lemma** *lossless-random-oracle* [*simp*]:  
*lossless-spmf*  $p \implies \text{lossless-spmf } (\text{random-oracle } \sigma x)$   
**by** (*simp add: lossless-spmf-def*)

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

**lemma** *card-dom-random-oracle*:  
**assumes** *interaction-any-bounded-by*  $\mathcal{A} q$   
**and**  $(y, \sigma') \in \text{set-spmf } (\text{exec-gpv } \text{random-oracle } \mathcal{A} \sigma)$   
**and** *fin: finite* ( $\text{dom } \sigma$ )  
**shows**  $\text{card } (\text{dom } \sigma') \leq q + \text{card } (\text{dom } \sigma)$   
**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* ::  $'key \text{ spmf}$   
**and** *prf* ::  $'key \Rightarrow 'domain \Rightarrow 'range$   
**and** *rand* ::  $'range \text{ spmf}$   
**begin**

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

**definition** *prf-oracle* ::  $'key \Rightarrow \text{unit} \Rightarrow 'domain \Rightarrow ('range \times \text{unit}) \text{ spmf}$   
**where** *prf-oracle*  $\text{key } \sigma x = \text{return-spmf } (\text{prf } \text{key } x, ())$

**type-synonym**  $( 'domain', 'range') \text{ adversary} = (\text{bool}, 'domain', 'range') \text{ gpv}$

**definition** *game-0* ::  $( 'domain, 'range) \text{ adversary} \Rightarrow \text{bool} \text{ spmf}$   
**where**  
*game-0*  $\mathcal{A} = \text{do } \{$   
    $\text{key} \leftarrow \text{key-gen};$

```

    (b, -) ← exec-gpv (prf-oracle key)  $\mathcal{A}$  ();
    return-spmf b
  }

```

**definition** *game-1* :: ('domain, 'range) adversary  $\Rightarrow$  bool spmf  
**where**

```

game-1  $\mathcal{A}$  = do {
  (b, -) ← exec-gpv random-oracle  $\mathcal{A}$  Map.empty;
  return-spmf b
}

```

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

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

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

**abbreviation** (*input*) *ibounded-by* :: ('domain, 'range) adversary  $\Rightarrow$  enat  $\Rightarrow$  bool  
**where** *ibounded-by*  $\equiv$  *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  $\rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  ('b  $\times$  ('a  $\rightarrow$  'b)) spmf

**where**

*random-permutation*  $\sigma$  *x* =

(case  $\sigma$  *x* of Some *y*  $\Rightarrow$  *return-spmf* (*y*,  $\sigma$ ))

| None  $\Rightarrow$  *spmf-of-set* (*A* - *ran*  $\sigma$ )  $\gg\equiv$  ( $\lambda y. \text{return-spmf } (y, \sigma(x \mapsto y))$ ))

**lemma** *weight-random-oracle* [*simp*]:

$\llbracket \text{finite } A; A - \text{ran } \sigma \neq \{\} \rrbracket \Longrightarrow \text{weight-spmf } (\text{random-permutation } \sigma \ x) = 1$

**by** (*simp add: random-permutation-def weight-bind-spmf o-def split. option.split*)

**lemma** *lossless-random-oracle* [*simp*]:

$\llbracket \text{finite } A; A - \text{ran } \sigma \neq \{\} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{random-permutation } \sigma \ x)$

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

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

**lemma** *card-dom-random-oracle:*

**assumes** *interaction-any-bounded-by*  $\mathcal{A}$  *q*  
**and**  $(y, \sigma') \in \text{set-spmf } (\text{exec-gpv } \text{random-permutation } \mathcal{A} \ \sigma)$   
**and** *fin: finite* (*dom*  $\sigma$ )  
**shows**  $\text{card } (\text{dom } \sigma') \leq q + \text{card } (\text{dom } \sigma)$   
**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* ::  $('c-o \times 'c-a \times 's)$  *spmf*  
**and** *oracle* ::  $'c-o \Rightarrow 's \Rightarrow 'call \Rightarrow ('ret \times 's)$  *spmf*  
**and** *eval* ::  $'c-o \Rightarrow 'c-a \Rightarrow 's \Rightarrow 'guess \Rightarrow \text{bool}$  *spmf*

**begin**

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

**definition** *game-single* ::  $('c-a, 'guess, 'call, 'ret)$  *adversary-single*  $\Rightarrow \text{bool}$  *spmf*

**where**

*game-single*  $\mathcal{A} = \text{do } \{$   
   $(c-o, c-a, s) \leftarrow \text{init};$   
   $(guess, s') \leftarrow \text{exec-gpv } (\text{oracle } c-o) (\mathcal{A} \ c-a) \ s;$   
   $\text{eval } c-o \ c-a \ s' \ \text{guess}$   
 $\}$

**definition** *advantage-single* ::  $('c-a, 'guess, 'call, 'ret)$  *adversary-single*  $\Rightarrow \text{real}$

**where** *advantage-single*  $\mathcal{A} = \text{spmf } (\text{game-single } \mathcal{A}) \ \text{True}$

**type-synonym**  $('c-a', 'guess', 'call', 'ret)$  *adversary-many* =  $'c-a' \Rightarrow (\text{unit}, 'call' + 'guess', 'ret' + \text{unit})$  *gpv*

**definition** *eval-oracle* :: 'c-o  $\Rightarrow$  'c-a  $\Rightarrow$  bool  $\times$  's  $\Rightarrow$  'guess  $\Rightarrow$  (unit  $\times$  (bool  $\times$  's)) spmf  
**where**  
*eval-oracle* c-o c-a = ( $\lambda(b, s')$  guess. map-spmf ( $\lambda b'. (((), (b \vee b', s'))$ )) (eval c-o c-a s' guess))

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

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

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

**primrec** *process-call* :: 'guess reduction-state  $\Rightarrow$  'call  $\Rightarrow$  ('ret option  $\times$  'guess reduction-state, 'call, 'ret) gpv  
**where**  
*process-call* (Inr j) x = do {  
ret  $\leftarrow$  Pause x Done;  
Done (Some ret, Inr j)  
}  
| *process-call* (Inl guess) x = Done (None, Inl guess)

**primrec** *process-guess* :: 'guess reduction-state  $\Rightarrow$  'guess  $\Rightarrow$  (unit option  $\times$  '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  $\Rightarrow$  'call + 'guess  $\Rightarrow$  (('ret + unit) option  $\times$  ('guess + nat), 'call, 'ret) gpv  
**where** *reduction-oracle*  $\equiv$  plus-intercept-stop process-call process-guess

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

```

  Done (projl s)
}

```

**lemma many-single-reduction:**

**assumes** bound:  $\bigwedge c-a\ c-o\ s. (c-o, c-a, s) \in \text{set-spmf init} \implies \text{interaction-bounded-by (Not } \circ \text{isl) } (\mathcal{A}\ c-a)\ q$

**and** lossless-oracle:  $\bigwedge c-a\ c-o\ s\ s'x. (c-o, c-a, s) \in \text{set-spmf init} \implies \text{lossless-spmf (oracle } c-o\ s'x)$

**and** lossless-eval:  $\bigwedge c-a\ c-o\ s\ s'\ \text{guess}. (c-o, c-a, s) \in \text{set-spmf init} \implies \text{lossless-spmf (eval } c-o\ c-a\ s'\ \text{guess)}$

**shows** advantage-multi  $\mathcal{A} \leq \text{advantage-single (reduction } q\ \mathcal{A}) * q$

**including** lifting-syntax

**proof** –

**define** eval-oracle'

**where** eval-oracle' =  $(\lambda c-o\ c-a\ ((id, occ :: \text{nat option}), s')\ \text{guess}.$

map-spmf  $(\lambda b'. \text{case } occ \text{ of Some } j_0 \Rightarrow ((), (Suc\ id, Some\ j_0), s')$

| None  $\Rightarrow ((), (Suc\ id, (if\ b' \text{ then Some } id \text{ else None})), s')$ )

(eval c-o c-a s' guess))

**let** ?multi'-body =  $\lambda c-o\ c-a\ s. \text{exec-gpv } (\dagger(\text{oracle } c-o) \oplus_O \text{eval-oracle}'\ c-o\ c-a)\ (\mathcal{A}\ c-a)$   
 $(((), None), s)$

**define** game-multi' **where** game-multi' =  $(\lambda c-o\ c-a\ s. \text{do } \{$

$(-, ((id, j_0), s' :: 's)) \leftarrow ?\text{multi}'\text{-body } c-o\ c-a\ s;$

return-spmf  $(j_0 \neq \text{None}) \}$ )

**define** initialize ::  $('c-o \Rightarrow 'c-a \Rightarrow 's \Rightarrow \text{nat} \Rightarrow \text{bool spmf}) \Rightarrow \text{bool spmf}$  **where**

initialize body = do {

$(c-o, c-a, s) \leftarrow \text{init};$

$j_s \leftarrow \text{spmf-of-set } \{..<q\};$

body c-o c-a s  $j_s$  } **for** body

**define** body2 **where** body2 c-o c-a s  $j_s$  = do {

$(-, (id, j_0), s') \leftarrow ?\text{multi}'\text{-body } c-o\ c-a\ s;$

return-spmf  $(j_0 = \text{Some } j_s)$  } **for** c-o c-a s  $j_s$

**let** ?game2 = initialize body2

**define** stop-oracle **where** stop-oracle =  $(\lambda c-o.$

$(\lambda (idgs, s)\ x. \text{case } idgs \text{ of Inr } - \Rightarrow \text{map-spmf } (\lambda (y, s). (\text{Some } y, (idgs, s))) (\text{oracle } c-o\ s\ x) \mid \text{Inl } - \Rightarrow \text{return-spmf } (\text{None}, (idgs, s)))$

$\oplus_O^S$

$(\lambda (idgs, s)\ \text{guess} :: 'guess. \text{return-spmf } (\text{case } idgs \text{ of Inr } 0 \Rightarrow (\text{None}, \text{Inl } (\text{guess}, s), s) \mid \text{Inr } (\text{Suc } i) \Rightarrow (\text{Some } (), \text{Inr } i, s) \mid \text{Inl } - \Rightarrow (\text{None}, idgs, s))))$

**define** body3 **where** body3 c-o c-a s  $j_s$  = do {

$(- :: \text{unit option}, idgs, -) \leftarrow \text{exec-gpv-stop } (\text{stop-oracle } c-o)\ (\mathcal{A}\ c-a)\ (\text{Inr } j_s, s);$

$(b' :: \text{bool}) \leftarrow \text{case } idgs \text{ of Inr } - \Rightarrow \text{return-spmf } \text{False} \mid \text{Inl } (g, s') \Rightarrow \text{eval } c-o\ c-a\ s'\ g;$

return-spmf  $b'$  } **for** c-o c-a s  $j_s$

**let** ?game3 = initialize body3

{ **define** S ::  $\text{bool} \Rightarrow \text{nat} \times \text{nat option} \Rightarrow \text{bool}$  **where** S  $\equiv \lambda b' (id, occ). b' \longleftrightarrow (\exists j_0. \text{occ} = \text{Some } j_0)$

**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 ==> (=)) (λ b. b) result by (simp add: rel-fun-def result-def
S-def)
have [transfer-rule]: S False initial by (simp add: S-def initial-def)

have eval-oracle'[transfer-rule]:
  ((=) ==> (=) ==> ?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-refl
split: option.split)

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

have bound-occ:  $j_s < q$  if that: (((), (id, Some  $j_s$ ),  $s'$ ) ∈ 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,  $j_0$ ),  $s'$ ).  $j_0 \neq \text{None}$ }
  by (auto simp add: game-multi'-def map-spmf-conv-bind-spmf[symmetric] split-def
spmf-conv-measure-spmf measure-map-spmf vimage-def)
  also have {(u, (id,  $j_0$ ),  $s'$ ).  $j_0 \neq \text{None}$ } =
  {((), (id, Some  $j_s$ ),  $s'$ ) |  $j_s$   $s'$  id.  $j_s < q$ } ∪ {((), (id, Some  $j_s$ ),  $s'$ ) |  $j_s$   $s'$  id.  $j_s \geq q$ }
  (is - = ?A ∪ -) by auto

```

**also have**  $?M \dots = ?M ?A$   
**by** (rule *measure-spmf.measure-zero-union*)(*auto simp add: measure-spmf-zero-iff*  
*dest: bound-occ*)  
**also have**  $\dots = \text{measure } (\text{measure-spmf } (\text{pair-spmf } (\text{spmf-of-set } \{.. < q\})) (?multi^l\text{-body}$   
*c-o c-a s*))  
 $\{ (j_s, ()), (id, j_0), s^{\wedge} | j_s j_0 s^{\wedge} id. j_0 = \text{Some } j_s \} * q$   
*(is - = measure ?M' ?B \* -)*  
**proof** –  
**have**  $?B = \{ (j_s, ()), (id, j_0), s^{\wedge} | j_s j_0 s^{\wedge} id. j_0 = \text{Some } j_s \wedge j_s < q \} \cup$   
 $\{ (j_s, ()), (id, j_0), s^{\wedge} | j_s j_0 s^{\wedge} id. j_0 = \text{Some } j_s \wedge j_s \geq q \}$  *(is - = ?Set1  $\cup$  ?Set2)*  
**by** *auto*  
**then have** *measure ?M' ?B = measure ?M' (?Set1  $\cup$  ?Set2)* **by** *simp*  
**also have**  $\dots = \text{measure } ?M' ?Set1$   
**by** (rule *measure-spmf.measure-zero-union*) (*auto simp add: measure-spmf-zero-iff*)  
**also have**  $\dots = (\sum_{j \in \{0.. < q\}}. \text{measure } ?M' (\{j\} \times \{ ((), (id, \text{Some } j), s^{\wedge}) | s^{\wedge} id. \text{True} \}))$   
**by** (*subst measure-spmf.finite-measure-finite-Union[symmetric]*)  
*(auto intro!: arg-cong2[where f=measure] simp add: disjoint-family-on-def)*  
**also have**  $\dots = (\sum_{j \in \{0.. < q\}}. 1 / q * \text{measure } (\text{measure-spmf } (?multi^l\text{-body } c-o c-a$   
*s*))  $\{ ((), (id, \text{Some } j), s^{\wedge}) | s^{\wedge} id. \text{True} \}$   
**by** (*simp add: measure-pair-spmf-times-spmf-conv-measure-spmf[symmetric] spmf-of-set*)  
**also have**  $\dots = 1 / q * \text{measure } (\text{measure-spmf } (?multi^l\text{-body } c-o c-a s)) \{ ((), (id,$   
*Some } j\_s), s^{\wedge}) | j\_s s^{\wedge} 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**  $?B = (\lambda (j_s, -, (-, j_0), -). j_0 = \text{Some } j_s) - ' \{ \text{True} \}$   
**by** (*auto simp add: vimage-def*)  
**also have** *rw2: measure ?M'  $\dots = \text{spmf } (\text{bind-spmf } (\text{spmf-of-set } \{.. < q\})) (\text{body2 } c-o$*   
*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 ( $\lambda (c-a, c-o, s). \text{game-multi}^l c-a c-o s)) \text{True} = \text{spmf } ?\text{game2}$*   
*True \* q*  
**unfolding** *initialize-def-spmf-bind[where p=init]*  
**by** (*auto intro!: integral-cong-AE simp del: integral-mult-left-zero simp add: inte-*  
*gral-mult-left-zero[symmetric]*)*

**moreover**

**have** *ord-spmf ( $\longrightarrow$ ) (body2 c-o c-a s j\_s) (body3 c-o c-a s j\_s)*

**if** *init: (c-o, c-a, s)  $\in$  set-spmf init and  $j_s: j_s < \text{Suc } q$  for c-o c-a s  $j_s$*

**proof** –

**define** *oracle2'* **where** *oracle2'  $\equiv \lambda (b, (id, gs), s) \text{ guess. if } id = j_s \text{ then do } \{$*

*b' :: bool  $\leftarrow$  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**  $?R = \lambda((id1, j_0), s1) (b', (id2, gs), s2). s1 = s2 \wedge id1 = id2 \wedge (j_0 = \text{Some } j_s \longrightarrow b' = \text{Some True}) \wedge (id2 \leq j_s \longrightarrow b' = \text{None})$   
**from** *init* **have**  $rel\text{-}spmf (rel\text{-}prod (=) ?R)$   
 $(exec\text{-}gpv (extend\text{-}state\text{-}oracle (oracle\ c\text{-}o) \oplus_O eval\text{-}oracle' c\text{-}o\ c\text{-}a) (\mathcal{A}\ c\text{-}a) ((0, \text{None}), s))$   
 $(exec\text{-}gpv (extend\text{-}state\text{-}oracle (extend\text{-}state\text{-}oracle (oracle\ c\text{-}o)) \oplus_O oracle2') (\mathcal{A}\ c\text{-}a) (\text{None}, (0, \text{None}), s))$   
**by**(*intro*  $exec\text{-}gpv\text{-}oracle\text{-}bisim$ [**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-refl split: option.split-asm plus-oracle-split if-split-asm*)  
**then** **have**  $rel\text{-}spmf (\longrightarrow) (body2\ c\text{-}o\ c\text{-}a\ s\ j_s)$   
 $(do \{$   
 $(-, b', -, -) \leftarrow exec\text{-}gpv (\dagger\dagger(oracle\ c\text{-}o) \oplus_O oracle2') (\mathcal{A}\ c\text{-}a) (\text{None}, (0, \text{None}), s);$   
 $return\text{-}spmf (b' = \text{Some True}) \}$   
 $(is\ rel\text{-}spmf\ -\ -\ ?body2')$   
— 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**  $body2\text{-}def$  **by**(*rule*  $rel\text{-}spmf\text{-}bind1$ ) *clarsimp*  
**also**  
**let**  $?guess\text{-}oracle = \lambda((id, gs), s) guess. return\text{-}spmf ((), (Suc\ id, if\ id = j_s\ then\ Some (guess, s)\ else\ gs), s)$   
**let**  $?I = \lambda(idgs, s). case\ idgs\ of\ (-, \text{None}) \Rightarrow False \mid (i, \text{Some } -) \Rightarrow j_s < i$   
**interpret**  $I: callee\text{-}invariant\text{-}on\ \dagger(oracle\ c\text{-}o) \oplus_O ?guess\text{-}oracle\ ?I\ \mathcal{I}\text{-full}$   
**by**(*simp*)(*unfold-locales; auto split: option.split*)  
  
**let**  $?f = \lambda s. case\ snd\ (fst\ s)\ of\ \text{None} \Rightarrow return\text{-}spmf\ False \mid \text{Some } a \Rightarrow eval\ c\text{-}o\ c\text{-}a\ (snd\ a)\ (fst\ a)$   
**let**  $?X = \lambda j_s (b1, (id1, gs1), s1) (b2, (id2, gs2), s2). b1 = b2 \wedge id1 = id2 \wedge gs1 = gs2 \wedge s1 = s2 \wedge (b2 = \text{None} \longleftrightarrow gs2 = \text{None}) \wedge (id2 \leq j_s \longrightarrow b2 = \text{None})$   
**have**  $?body2' = do \{$   
 $(a, r, s) \leftarrow exec\text{-}gpv (\lambda(r, s) x. do \{$   
 $(y, s') \leftarrow (\dagger(oracle\ c\text{-}o) \oplus_O ?guess\text{-}oracle) s\ x;$   
 $if\ ?I\ s' \wedge r = \text{None}\ then\ map\text{-}spmf\ (\lambda r. (y, \text{Some } r, s'))\ (?f\ s')\ else\ return\text{-}spmf$   
 $(y, r, s')$   
 $\})$   
 $(\mathcal{A}\ c\text{-}a) (\text{None}, (0, \text{None}), s);$   
 $case\ r\ of\ \text{None} \Rightarrow ?f\ s \gg\gg return\text{-}spmf \mid \text{Some } r' \Rightarrow return\text{-}spmf\ r' \}$   
**unfolding**  $oracle2'\text{-}def\ spmf\text{-}rel\text{-}eq$ [*symmetric*]  
**by**(*rule*  $rel\text{-}spmf\text{-}bind1$ [*OF*  $exec\text{-}gpv\text{-}oracle\text{-}bisim'$ [**where**  $X=?X\ j_s$ ]])  
 $(auto\ simp\ add: bind\text{-}map\text{-}spmf\ o\text{-}def\ spmf.\ map\text{-}comp\ split\text{-}beta\ conj\text{-}comms\ map\text{-}spmf\text{-}conv\text{-}bind\text{-}spmf$ [*symmetric*]  
 $spmf\text{-}rel\text{-}map\ rel\text{-}spmf\text{-}refl\ cong: conj\text{-}cong\ split: plus\text{-}oracle\text{-}split$ )  
**also** **have**  $\dots = do \{$   
 $us' \leftarrow exec\text{-}gpv (\dagger(oracle\ c\text{-}o) \oplus_O ?guess\text{-}oracle) (\mathcal{A}\ c\text{-}a) ((0, \text{None}), s);$   
 $(b' :: bool) \leftarrow ?f (snd\ us');$   
 $return\text{-}spmf\ b' \}$   
 $(is\ - = ?body2'')$   
**by**(*rule*  $I.exec\text{-}gpv\text{-}bind\text{-}materialize$ [*symmetric*])(*auto split: plus-oracle-split-asm option.split-asm*)

```

also have ... = do {
  us' ← exec-gpv-stop (lift-stop-oracle (†(oracle c-o) ⊕O ?guess-oracle)) (ℳ 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 ≤ js)
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))) ⊕OS
(λ((id, gs), s) guess. return-spmf (if id ≥ js then None else Some (), (Suc id, if id
= js then Some (guess, s) else gs), s)))
(ℳ c-a) ((0, None), s) >>=
(λus'. 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-refl)
also let ?X = λ((id, gs), s1) (idgs, s2). s1 = s2 ∧ (case (gs, idgs) of (None, Inr id') ⇒
id' = js - id ∧ id ≤ js | (Some gs, Inl gs') ⇒ gs = gs' ∧ id > js | - ⇒ False)
have ... = body3 c-o c-a s js 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-refl)
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-refl)
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 ℳ))
unfolding body3-def stop-oracle-def game-single-def reduction-def split-def initial-
ize-def
apply(clarsimp simp add: bind-map-spmf exec-gpv-bind exec-gpv-inline intro!: rel-spmf-bind-refl)
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-refl rel-spmf-return-spmf1 lossless-eval split: nat.split)
done
finally have ord-spmf (→) ?game2 (game-single (reduction q ℳ))
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 ℳ)) 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-hash* *key*)  $\oplus_O$  *oracle-flag* *key*

**definition** *game* :: ('x, 'hash) *adversary*  $\Rightarrow$  bool *spmf*

**where**

*game*  $\mathcal{A}$  = do {

*key*  $\leftarrow$  *key-gen*;

  (-, (*flag'*, *L*))  $\leftarrow$  *exec-gpv* (*oracle* *key*)  $\mathcal{A}$  (*False*, {});

  return-spmf *flag'*

}

**definition** *advantage* :: ('x, 'hash) *adversary*  $\Rightarrow$  real

**where** *advantage*  $\mathcal{A}$  = *spmf* (*game*  $\mathcal{A}$ ) *True*

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

**definition** *game1* :: ('x, 'hash) adversary1  $\Rightarrow$  bool spmf

**where**

```
game1  $\mathcal{A}$  = do {  
  key  $\leftarrow$  key-gen;  
  ((m, h), L)  $\leftarrow$  exec-gpv (oracle-hash key)  $\mathcal{A}$  {};  
  return-spmf (h = hash key m  $\wedge$  m  $\notin$  L)  
}
```

**definition** *advantage1* :: ('x, 'hash) adversary1  $\Rightarrow$  real

**where** *advantage1*  $\mathcal{A}$  = spmf (game1  $\mathcal{A}$ ) True

**lemma** *advantage-advantage1*:

**assumes** *bound*: interaction-bounded-by (Not  $\circ$  isl)  $\mathcal{A}$  *q*

**shows** *advantage*  $\mathcal{A}$   $\leq$  *advantage1* (guessing-many-one.reduction *q* ( $\lambda$ - :: unit.  $\mathcal{A}$ ) ())

\* *q*

**proof** –

**let** ?*init* = map-spmf ( $\lambda$ key. (key, (), {})) key-gen

**let** ?*oracle* =  $\lambda$ key . oracle-hash key

**let** ?*eval* =  $\lambda$ key (- :: unit) L (x, h). return-spmf (h = hash key x  $\wedge$  x  $\notin$  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*  $\mathcal{A}$  = *game-multi* ( $\lambda$ - .  $\mathcal{A}$ )

**by** (*auto* *simp* add: *game-multi-def* *game-def* *bind-map-spmf* *intro!*: *bind-spmf-cong*[*OF refl*])

**hence** *advantage*  $\mathcal{A}$  = *advantage-multi* ( $\lambda$ - .  $\mathcal{A}$ ) **by** (*simp* add: *advantage-def* *advantage-multi-def*)

**also have** ...  $\leq$  *advantage-single* (*reduction* *q* ( $\lambda$ - .  $\mathcal{A}$ )) \* *q* **using** *bound*

**by** (*rule* *many-single-reduction*)(*auto* *simp* add: *oracle-hash-def*)

**also have** *advantage-single* (*reduction* *q* ( $\lambda$ - .  $\mathcal{A}$ )) = *advantage1* (*reduction* *q* ( $\lambda$ - .  $\mathcal{A}$ )) **for**  $\mathcal{A}$

**unfolding** *advantage1-def* *advantage-single-def*

**by** (*auto* *simp* add: *game1-def* *game-single-def* *bind-map-spmf* *o-def* *intro!*: *bind-spmf-cong*[*OF refl*] *arg-cong2*[**where** *f*=*spmf*])

**finally show** ?*thesis* .

**qed**

**end**

**end**

**theory** *Security-Spec* **imports**

*Diffie-Hellman*

*IND-CCA2*

*IND-CCA2-sym*

*IND-CPA*

*IND-CPA-PK*  
*IND-CPA-PK-Single*  
*SUF-CMA*  
*Pseudo-Random-Function*  
*Pseudo-Random-Permutation*  
*Unpredictable-Function*  
**begin**  
  
**end**

## 2 Cryptographic constructions and their security

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

### 2.1 Elgamal encryption scheme

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

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

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

**lemma** *key-gen-alt*:  
 $\text{key-gen} = \text{map-spmf } (\lambda x. (\mathbf{g} [\wedge] x, x)) (\text{sample-uniform } (\text{order } \mathcal{G}))$   
**by** (*simp add: map-spmf-conv-bind-spmf key-gen-def*)

**definition** *aencrypt* :: *'grp pub-key*  $\Rightarrow$  *'grp*  $\Rightarrow$  *'grp cipher* *spmf*  
**where**  
*aencrypt*  $\alpha$  *msg* = *do* {  
 $y \leftarrow \text{sample-uniform}(\text{order } \mathcal{G});$   
 $\text{return-spmf } (\mathbf{g} [\wedge] y, (\alpha [\wedge] y) \otimes \text{msg})$   
}

**lemma** *aencrypt-alt*:

$aencrypt\ \alpha\ msg = map\text{-}spmf\ (\lambda y. (\mathbf{g} [\wedge] y, (\alpha [\wedge] y) \otimes msg))\ (sample\text{-}uniform\ (order\ \mathcal{G}))$   
**by** (*simp add: map-spmf-conv-bind-spmf aencrypt-def*)

**definition** *adecrypt* :: 'grp priv-key  $\Rightarrow$  'grp cipher  $\Rightarrow$  'grp option

**where**

$adecrypt\ x = (\lambda(\beta, \zeta). Some\ (\zeta \otimes (inv\ (\beta [\wedge] x))))$

**abbreviation** *valid-plains* :: 'grp  $\Rightarrow$  'grp  $\Rightarrow$  bool

**where** *valid-plains* *msg1* *msg2*  $\equiv msg1 \in carrier\ \mathcal{G} \wedge msg2 \in carrier\ \mathcal{G}$

**sublocale** *ind-cpa*: *ind-cpa* *key-gen* *aencrypt* *adecrypt* *valid-plains* .

**sublocale** *ddh*: *ddh*  $\mathcal{G}$  .

**fun** *elgamal-adversary* :: ('grp pub-key, 'grp plain, 'grp cipher, 'state) *ind-cpa.adversary*  
 $\Rightarrow$  'grp *ddh.adversary*

**where**

*elgamal-adversary* ( $\mathcal{A}1, \mathcal{A}2$ )  $\alpha\ \beta\ \gamma = TRY\ do\ \{\$   
   $b \leftarrow coin\text{-}spmf;$   
   $((msg1, msg2), \sigma) \leftarrow \mathcal{A}1\ \alpha;$   
  — have to check that the attacker actually sends two elements from the group; otherwise  
  flip a coin  
   $- :: unit \leftarrow assert\text{-}spmf\ (valid\text{-}plains\ msg1\ msg2);$   
   $guess \leftarrow \mathcal{A}2\ (\beta, \gamma \otimes (if\ b\ then\ msg1\ else\ msg2))\ \sigma;$   
   $return\text{-}spmf\ (guess = b)$   
   $\}\ ELSE\ coin\text{-}spmf$

**end**

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

**begin**

**theorem** *advantage-elgamal*: *ind-cpa.advantage*  $\mathcal{A} = ddh.advantage\ (elgamal\text{-}adversary\ \mathcal{A})$

**including** *monad-normalisation*

**proof** —

**obtain**  $\mathcal{A}1$  **and**  $\mathcal{A}2$  **where**  $\mathcal{A} = (\mathcal{A}1, \mathcal{A}2)$  **by** (*cases*  $\mathcal{A}$ )

**note** [*simp*] = *this order-gt-0-iff-finite finite-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*  $\mathcal{A}$ ) = *TRY* *do*  $\{\$

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

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

$((msg1, msg2), \sigma) \leftarrow \mathcal{A}1\ (\mathbf{g} [\wedge] x);$

$- :: unit \leftarrow assert\text{-}spmf\ (valid\text{-}plains\ msg1\ msg2);$

$b \leftarrow coin\text{-}spmf;$

$z \leftarrow map\text{-}spmf\ (\lambda z. \mathbf{g} [\wedge] z \otimes (if\ b\ then\ msg1\ else\ msg2))\ (sample\text{-}uniform\ (order\ \mathcal{G}));$

$guess \leftarrow \mathcal{A}2\ (\mathbf{g} [\wedge] y, z)\ \sigma;$

$return\text{-}spmf\ (guess \longleftrightarrow b)$   
 $\}$

```

    } ELSE coin-spmf
  by(simp add: ddh.ddh-1-def)
also have ... = TRY do {
  x ← sample-uniform (order  $\mathcal{G}$ );
  y ← sample-uniform (order  $\mathcal{G}$ );
  ((msg1, msg2),  $\sigma$ ) ←  $\mathcal{A}1$  ( $\mathbf{g}^{[\wedge]} x$ );
  - :: unit ← assert-spmf (valid-plains msg1 msg2);
  z ← map-spmf ( $\lambda z. \mathbf{g}^{[\wedge]} z$ ) (sample-uniform (order  $\mathcal{G}$ ));
  guess ←  $\mathcal{A}2$  ( $\mathbf{g}^{[\wedge]} y, z$ )  $\sigma$ ;
  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  $\mathcal{A}$ ) = ind-cpa.ind-cpa  $\mathcal{A}$ 
by(simp add: ddh.ddh-0-def IND-CPA-PK-Single.ind-cpa.ind-cpa-def key-gen-def 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  $\mathcal{G}$  :: security  $\Rightarrow$  'grp cyclic-group
  assumes elgamal:  $\bigwedge \eta. \text{elgamal}(\mathcal{G} \ \eta)$ 
begin

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

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

end

context elgamal-base begin

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

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

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

```

**end**

**end**

## 2.2 Hashed Elgamal in the Random Oracle Model

**theory** *Hashed-Elgamal* **imports**

*CryptHOL.GPV-Bisim*

*CryptHOL.Cyclic-Group-SPMF*

*CryptHOL.List-Bits*

*IND-CPA-PK*

*Diffie-Hellman*

**begin**

**type-synonym** *bitstring* = *bool list*

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

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

**definition** *oracle* :: *'a state*  $\Rightarrow$  *'a*  $\Rightarrow$  (*bitstring*  $\times$  *'a state*) *spmf*

**where**

*oracle*  $\sigma$  *x* =

(*case*  $\sigma$  *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*,  $\sigma$ ))

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

**inductive** *invariant* :: *'a state*  $\Rightarrow$  *bool*

**where**

*invariant*:  $\llbracket \text{finite } (\text{dom } \sigma); \text{length } \text{ran } \sigma \subseteq \{len\} \rrbracket \Longrightarrow \text{invariant } \sigma$

**lemma** *invariant-initial* [*simp*]: *invariant initial*

**by** (*rule invariant.intros*) *auto*

**lemma** *invariant-update* [*simp*]:  $\llbracket \text{invariant } \sigma; \text{length } bs = len \rrbracket \Longrightarrow \text{invariant } (\sigma(x \mapsto bs))$

**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* ( $\lambda \sigma. x \in \text{dom } \sigma$ )

**by** *unfold-locales* (*simp-all add: oracle-def split: option.split-asm*)

**lemma** *lossless-oracle* [*simp*]: *lossless-spmf* (*oracle*  $\sigma$  *x*)



```

by(simp add: oracle-def split: option.split)

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

end

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

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

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

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

  definition aencrypt :: 'grp pub-key  $\Rightarrow$  plain  $\Rightarrow$  ('grp cipher, 'grp, bitstring) gpv
  where
    aencrypt  $\alpha$  msg = do {
      y  $\leftarrow$  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* ℒ (aencrypt α msg) ⇔ 0 < order ℒ  
**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* ℒ .

**fun** *elgamal-adversary*

```

:: ('grp pub-key, plain, 'grp cipher, 'grp, bitstring, 'state) ind-cpa.adversary
⇒ 'grp lcdh.adversary

```

**where**

```

elgamal-adversary (ℒ1, ℒ2) α β = do {
  ((msg1, msg2), σ), s ← exec-gpv hash.oracle (ℒ1 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 (ℒ2 (β, h') σ) s;
    return-spmf (dom s')
  } ELSE return-spmf (dom s)
}

```

**end**

**locale** *elgamal* = *elgamal-base* +

**assumes** *cyclic-group*: *cyclic-group* ℒ

**begin**

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

**lemma** *advantage-elgamal*:

**includes** *lifting-syntax*  
**assumes** *lossless: ind-cpa.lossless*  $\mathcal{A}$   
**shows** *ind-cpa.advantage hash.oracle hash.initial*  $\mathcal{A} \leq \text{lcdh.advantage (elgamal-adversary } \mathcal{A})$   
**proof** –  
**note** [*cong del*] = *if-weak-cong* **and** [*split del*] = *if-split*  
**and** [*simp*] = *map-lift-spmf gpv.map-id lossless-weight-spmfD map-spmf-bind-spmf bind-spmf-const*  
**obtain**  $\mathcal{A}1 \ \mathcal{A}2$  **where**  $\mathcal{A}$  [*simp*]:  $\mathcal{A} = (\mathcal{A}1, \mathcal{A}2)$  **by**(*cases*  $\mathcal{A}$ )  
  
**interpret** *cyclic-group: cyclic-group*  $\mathcal{G}$  **by**(*rule cyclic-group*)  
**from** *finite-carrier* **have** [*simp*]: *order*  $\mathcal{G} > 0$  **using** *order-gt-0-iff-finite* **by**(*simp*)  
  
**from** *lossless* **have** *lossless1* [*simp*]:  $\bigwedge pk. \text{lossless-gpv } \mathcal{I}\text{-full } (\mathcal{A}1 \ pk)$   
**and** *lossless2* [*simp*]:  $\bigwedge \sigma \text{ cipher. lossless-gpv } \mathcal{I}\text{-full } (\mathcal{A}2 \ \sigma \ \text{cipher})$   
**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'* =  $(\lambda \sigma x. \text{do } \{$   
 $h \leftarrow \text{hash } x;$   
 $\text{Done } (h, \text{insert } x \ \sigma)$   
 $\})$   
**have** [*simp*]: *lossless-gpv*  $\mathcal{I}\text{-full } (\text{hash-oracle}' \ \sigma \ x)$  **for**  $\sigma \ x$  **by**(*simp add: hash-oracle'-def*)  
**have** [*simp*]: *lossless-gpv*  $\mathcal{I}\text{-full } (\text{inline } \text{hash-oracle}' \ (\mathcal{A}1 \ \alpha) \ s)$  **for**  $\alpha \ s$   
**by**(*rule lossless-inline[where*  $\mathcal{I} = \mathcal{I}\text{-full}$ ]) *simp-all*  
**define** *game0* **where** *game0* = *TRY* **do** {  
 $(pk, -) \leftarrow \text{lift-spmf } \text{key-gen};$   
 $b \leftarrow \text{lift-spmf } \text{coin-spmf};$   
 $((\text{msg1}, \text{msg2}), \sigma), s \leftarrow \text{inline } \text{hash-oracle}' \ (\mathcal{A}1 \ pk) \ \{\};$   
 $\text{assert-gpv } (\text{valid-plains } \text{msg1 } \text{msg2});$   
 $\text{cipher} \leftarrow \text{aencrypt } pk \ (\text{if } b \ \text{then } \text{msg1} \ \text{else } \text{msg2});$   
 $(\text{guess}, s') \leftarrow \text{inline } \text{hash-oracle}' \ (\mathcal{A}2 \ \text{cipher } \sigma) \ s;$   
 $\text{Done } (\text{guess} = b)$   
 $\}$  *ELSE* *lift-spmf coin-spmf*  
**{ define** *cr* **where** *cr* =  $(\lambda - :: \text{unit. } \lambda - :: 'a \ \text{set. } \text{True})$   
**have** [*transfer-rule*]: *cr* () {} **by**(*simp add: cr-def*)  
**have** [*transfer-rule*]:  $((=) \implies (=) \implies \text{rel-gpv } (\text{rel-prod } (=) \ \text{cr}) \ (=))$  *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*  $\mathcal{A} = \text{game0}$  **unfolding** *game0-def*  $\mathcal{A}$  *ind-cpa-pk.ind-cpa.simps*  
**by**(*transfer fixing:*  $\mathcal{G}$  *len-plain*  $\mathcal{A}1 \ \mathcal{A}2$ )(*simp add: bind-map-gpv o-def ind-cpa-pk.ind-cpa.simps*  
*split-def*) }  
**note** *game0* = *this*  
**have** *game0-alt-def*: *game0* = **do** {  
 $x \leftarrow \text{lift-spmf } (\text{sample-uniform } (\text{order } \mathcal{G}));$   
 $b \leftarrow \text{lift-spmf } \text{coin-spmf};$

```

  (((msg1, msg2),  $\sigma$ ), s)  $\leftarrow$  inline hash-oracle' ( $\mathcal{A}1$  ( $\mathbf{g} [\wedge] x$ )) {};
  TRY do {
    - :: unit  $\leftarrow$  assert-gpv (valid-plains msg1 msg2);
    cipher  $\leftarrow$  aencrypt ( $\mathbf{g} [\wedge] x$ ) (if b then msg1 else msg2);
    (guess, s')  $\leftarrow$  inline hash-oracle' ( $\mathcal{A}2$  cipher  $\sigma$ ) 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'' = ( $\lambda(s, \sigma) (x :: 'a). do \{$ 
  ( $h, \sigma'$ )  $\leftarrow$  case  $\sigma$  x of
    None  $\Rightarrow$  bind-spmf (spmf-of-set (nlists UNIV len-plain)) ( $\lambda bs. return-spmf (bs, \sigma(x$ 
 $\mapsto bs))$ )
    | Some ( $bs :: bitstring$ )  $\Rightarrow$  return-spmf ( $bs, \sigma$ );
  return-spmf ( $h, insert x s, \sigma'$ )
})
have *: exec-gpv hash.oracle (inline hash-oracle'  $\mathcal{A} s$ )  $\sigma =$ 
  map-spmf ( $\lambda(a, b, c). ((a, b), c)$ ) (exec-gpv hash-oracle''  $\mathcal{A} (s, \sigma)$ ) for  $\mathcal{A} \sigma s$ 
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'' ( $\mathcal{A}1 \alpha$ ) s) for s  $\alpha$ 
by(rule lossless-exec-gpv[where  $\mathcal{I} = \mathcal{I}$ -full]) simp-all
have [simp]: lossless-spmf (exec-gpv hash-oracle'' ( $\mathcal{A}2 \sigma$  cipher) s) for  $\sigma$  cipher s
by(rule lossless-exec-gpv[where  $\mathcal{I} = \mathcal{I}$ -full]) simp-all

let ?sample =  $\lambda f. bind-spmf$  (sample-uniform (order  $\mathcal{G}$ )) ( $\lambda x. bind-spmf$  (sample-uniform
(order  $\mathcal{G}$ )) ( $f x$ ))
define game1 where game1 = ( $\lambda(x :: nat) (y :: nat). do \{$ 
  b  $\leftarrow$  coin-spmf;
  (((msg1, msg2),  $\sigma$ ), (s, s-h))  $\leftarrow$  exec-gpv hash-oracle'' ( $\mathcal{A}1$  ( $\mathbf{g} [\wedge] x$ )) ({}, hash.initial);
  TRY do {
    - :: unit  $\leftarrow$  assert-spmf (valid-plains msg1 msg2);
    ( $h, s-h'$ )  $\leftarrow$  hash.oracle s-h ( $\mathbf{g} [\wedge] (x * y)$ );
    let cipher = ( $\mathbf{g} [\wedge] y, h [\oplus]$ ) (if b then msg1 else msg2);
    (guess, (s', s-h'))  $\leftarrow$  exec-gpv hash-oracle'' ( $\mathcal{A}2$  cipher  $\sigma$ ) (s, s-h');
    return-spmf (guess = b,  $\mathbf{g} [\wedge] (x * y) \in s'$ )
  } ELSE do {
    b  $\leftarrow$  coin-spmf;
    return-spmf (b,  $\mathbf{g} [\wedge] (x * y) \in s$ )
  }
})
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-distrib * try-bind-assert-gpv try-bind-assert-spmf loss-
less-inline[where  $\mathcal{I} = \mathcal{I}$ -full] bind-rpv-def nat-pow-pow del: bind-spmf-const)
including monad-normalisation by(simp add: bind-rpv-def nat-pow-pow)

```

```

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

{ fix  $x y :: \text{nat}$ 
  let  $?bad = \lambda(s, s-h). \mathbf{g} [\wedge] (x * y) \in s$ 
  let  $?X = \lambda(s, s-h) (s', s-h'). s = \text{dom } s-h \wedge s' = s \wedge s-h = s-h' (\mathbf{g} [\wedge] (x * y) := \text{None})$ 
  have bisim:
     $\text{rel-spmf} (\lambda(a, s1') (b, s2'). ?bad \ s1' = ?bad \ s2' \wedge (\neg ?bad \ s2' \longrightarrow a = b \wedge ?X \ s1' \ s2'))$ 
    (hash-oracle'' s1 plain) (hash-oracle'' s2 plain)
    if  $?X \ s1 \ s2$  for  $s1 \ s2$  plain using that
    by (auto split: prod.splits intro!: rel-spmf-bind-refl 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  $\text{rel-spmf} (\lambda(\text{win}, \text{bad}) (\text{win}', \text{bad}'). \text{bad} = \text{bad}' \wedge (\neg \text{bad}' \longrightarrow \text{win} = \text{win}'))$ 
    (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-refl if-distrib
intro!: rel-spmf-bind-refl simp del: bind-spmf-const)
    apply (rule rel-spmf-try-spmf)
    subgoal for  $b \ \text{msg1} \ \text{msg2} \ \sigma \ s \ s-h$ 
    apply (rule rel-spmf-bind-refl)
    apply (drule inv''.exec-gpv-invariant; clarsimp)
    apply (cases s-h (g [^] (x * y)))
    subgoal — case None
    apply (clarsimp intro!: rel-spmf-bind-refl)
    apply (rule rel-spmf-bindI)
    apply (rule exec-gpv-oracle-bisim-bad-full[OF - - bisim inv inv, where R= $\lambda(x, s1)$ 
(y, s2). ?bad s1 = ?bad s2  $\wedge$  ( $\neg ?bad s2 \longrightarrow x = y$ )]; clarsimp simp add: fun-upd-idem;

```

```

fail)
  apply clarsimp
  done
  subgoal by(auto intro!: rel-spmf-bindI1 rel-spmf-bindI2 lossless-exec-gpv[where
 $\mathcal{I}=\mathcal{I}$ -full] dest!: callee-invariant-on.exec-gpv-invariant[OF in-encrypt-oracle])
  done
  subgoal by(rule rel-spmf-refl) simp
  done }
  hence rel-spmf ( $\lambda$ (win, bad) (win', bad'). (bad  $\longleftrightarrow$  bad')  $\wedge$  ( $\neg$  bad'  $\longrightarrow$  win  $\longleftrightarrow$  win'))
  (?sample game2) (?sample game1)
  by(intro rel-spmf-bind-refl)
  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 = ( $\lambda$ f :: -  $\Rightarrow$  -  $\Rightarrow$  -  $\Rightarrow$  bitstring spmf  $\Rightarrow$  - spmf.  $\lambda$ (x :: nat) (y :: nat). do {
    b  $\leftarrow$  coin-spmf;
    ((msg1, msg2),  $\sigma$ ), (s, s-h)  $\leftarrow$  exec-gpv hash-oracle'' ( $\mathcal{A}1$  ( $\mathbf{g}$  [^] 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 = ( $\mathbf{g}$  [^] y, h');
      (guess, (s', s-h'))  $\leftarrow$  exec-gpv hash-oracle'' ( $\mathcal{A}2$  cipher  $\sigma$ ) (s, s-h);
      return-spmf (guess = b,  $\mathbf{g}$  [^] (x * y)  $\in$  s')
    } ELSE do {
      b  $\leftarrow$  coin-spmf;
      return-spmf (b,  $\mathbf{g}$  [^] (x * y)  $\in$  s)
    }
  })
  let ?f =  $\lambda$ b msg1 msg2. map-spmf ( $\lambda$ 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 ( $\lambda$ - - 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 = (λ σ (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-refl)
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''' (A1 (g [^] x)) hash.initial;
  TRY do {
    - :: unit ← assert-spmf (valid-plains msg1 msg2);
    h' ← spmf-of-set (nlists UNIV len-plain);
    let cipher = (g [^] y, h');
    (guess, s') ← exec-gpv hash-oracle''' (A2 cipher σ) s;
    return-spmf (guess = b, g [^] (x * y) ∈ dom s')
  } ELSE do {
    b ← coin-spmf;
    return-spmf (b, g [^] (x * y) ∈ dom s)
  }
} for x y
unfolding game3-def Map-empty-def [symmetric] split-def fst-conv snd-conv prod.collapse
by(transfer fixing: A1 G len-plain x y A2) simp
moreover have map-spmf snd (... x y) = do {
  zs ← elgamal-adversary A (g [^] x) (g [^] y);
  return-spmf (g [^] (x * y) ∈ zs)
} for x y
by(simp add: o-def split-def hash-oracle'''-def map-try-spmf map-scale-spmf)
(simp add: o-def map-try-spmf map-scale-spmf map-spmf-conv-bind-spmf [symmetric]
spmf.map-comp map-const-spmf-of-set)
ultimately have map-spmf snd (?sample (game3 (λ - - x. x))) = lcdh.lcdh (elgamal-adversary
A)
by(simp add: o-def lcdh.lcdh-def Let-def nat-pow-pow) }
then have game2-snd: map-spmf snd (?sample game2) = lcdh.lcdh (elgamal-adversary
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'' (A1 (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'' (A2 (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  $\mathcal{A} = |$ spmf (map-spmf fst (?sample
game1)) True - 1 / 2|
using game0 by(simp add: ind-cpa-pk.advantage-def game01 o-def)
also have ... = |1 / 2 - spmf (map-spmf fst (?sample game1)) True|
by(simp add: abs-minus-commute)
also have 1 / 2 = spmf (map-spmf fst (?sample game2)) True
by(simp add: game23 o-def game3-fst spmf-of-set)
also note hop23 also note game2-snd
finally show ?thesis by(simp add: lcdh.advantage-def)
qed

end

context elgamal-base begin

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

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

end

end

```

## 2.3 The random-permutation random-function switching lemma

**theory** RP-RF **imports**

Pseudo-Random-Function

Pseudo-Random-Permutation

CryptHOL.GPV-Bisim

**begin**

**lemma** rp-resample:

**assumes**  $B \subseteq A \cup C$   $A \cap C = \{\}$   $C \subseteq B$  **and** finB: finite B

**shows** bind-spmf (spmf-of-set B) (λx. 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) (λx. if x ∈ A then spmf-of-set C else return-spmf x) =
    bind-spmf (spmf-of-set B) (λx. 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-eql)
    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-notinI[OF A'C] indicator-single-Some sum-mult-indicator[of C
λ-. 1 :: real λ-. - λx. 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-notinI 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 spm where
  game b ℳ = run-gpv (if b then rp.random-permutation else rf.random-oracle) ℳ
  Map.empty

```

```

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

```

```

definition advantage :: 'a adversary ⇒ real where
  advantage ℳ = |spm (prp-game ℳ) True - spmf (prf-game ℳ) True|

```

```

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

```

```

lemma advantage-le-1: advantage ℳ ≤ 1

```

**by**(*auto simp add: advantage-def intro!: abs-leI*)(*metis diff-0-right diff-left-mono order-trans pmf-le-1 pmf-nonneg*) +

**context includes**  $\mathcal{I}$ .*lifting begin*

**lift-definition**  $\mathcal{I} :: ('a, 'a) \mathcal{I}$  **is**  $(\lambda x. \text{if } x \in A \text{ then } A \text{ else } \{\})$  .

**lemma** *outs-* $\mathcal{I}$ - $\mathcal{I}$  [*simp*]: *outs-* $\mathcal{I}$   $\mathcal{I} = A$  **by** *transfer auto*

**lemma** *responses-* $\mathcal{I}$ - $\mathcal{I}$  [*simp*]: *responses-* $\mathcal{I}$   $\mathcal{I} x = (\text{if } x \in A \text{ then } A \text{ else } \{\})$  **by** *transfer simp*

**lifting-update**  $\mathcal{I}$ .*lifting*

**lifting-forget**  $\mathcal{I}$ .*lifting*

**end**

**lemma** *rp-rf*:

**assumes** *bound: interaction-any-bounded-by*  $\mathcal{A} q$

**and** *lossless: lossless-gpv*  $\mathcal{I} \mathcal{A}$

**and** *WT:  $\mathcal{I} \vdash_g \mathcal{A} \checkmark$*

**shows** *advantage*  $\mathcal{A} \leq q * q / \text{card } A$

**including** *lifting-syntax*

**proof** –

**let** *?run* =  $\lambda b. \text{exec-gpv } (\text{if } b \text{ then } \text{rp.random-permutation} \text{ else } \text{rf.random-oracle}) \mathcal{A}$   
*Map.empty*

**define** *rp-bad* ::  $\text{bool} \times ('a \rightarrow 'a) \Rightarrow 'a \Rightarrow ('a \times (\text{bool} \times ('a \rightarrow 'a))) \text{pmf}$

**where** *rp-bad* =  $(\lambda (\text{bad}, \sigma) x. \text{case } \sigma x \text{ of } \text{Some } y \Rightarrow \text{return-spmf } (y, (\text{bad}, \sigma))$

| *None*  $\Rightarrow \text{bind-spmf } (\text{spmof-of-set } A) (\lambda y. \text{if } y \in \text{ran } \sigma \text{ then } \text{map-spmf } (\lambda y'. (y', (\text{True}, \sigma(x \mapsto y')))) (\text{spmof-of-set } (A - \text{ran } \sigma)) \text{ else } \text{return-spmf } (y, (\text{bad}, (\sigma(x \mapsto y)))))$

**have** *rp-bad-simps: rp-bad*  $(\text{bad}, \sigma) x = (\text{case } \sigma x \text{ of } \text{Some } y \Rightarrow \text{return-spmf } (y, (\text{bad}, \sigma))$

| *None*  $\Rightarrow \text{bind-spmf } (\text{spmof-of-set } A) (\lambda y. \text{if } y \in \text{ran } \sigma \text{ then } \text{map-spmf } (\lambda y'. (y', (\text{True}, \sigma(x \mapsto y')))) (\text{spmof-of-set } (A - \text{ran } \sigma)) \text{ else } \text{return-spmf } (y, (\text{bad}, (\sigma(x \mapsto y)))))$

**for** *bad*  $\sigma$  *x* **by**(*simp add: rp-bad-def*)

**let** *?S* = *rel-prod2* (=)

**define** *init* ::  $\text{bool} \times ('a \rightarrow 'a)$  **where** *init* =  $(\text{False}, \text{Map.empty})$

**have** *rp: rp.random-permutation* =  $(\lambda \sigma x. \text{case } \sigma x \text{ of } \text{Some } y \Rightarrow \text{return-spmf } (y, \sigma)$

| *None*  $\Rightarrow \text{bind-spmf } (\text{bind-spmf } (\text{spmof-of-set } A) (\lambda y. \text{if } y \in \text{ran } \sigma \text{ then } \text{spmof-of-set } (A - \text{ran } \sigma) \text{ else } \text{return-spmf } y)) (\lambda y. \text{return-spmf } (y, \sigma(x \mapsto y)))$

**by**(*subst rp-resample*)(*auto simp add: finite-A rp.random-permutation-def[abs-def]*)

**have** [*transfer-rule*]:  $(?S \text{ ===== } (=) \text{ ===== } \text{rel-spmf } (\text{rel-prod } (=) ?S)) \text{rp.random-permutation}$   
*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-refl*)

**have** [*transfer-rule*]:  $?S \text{Map.empty}$  *init* **by**(*simp add: init-def*)

**have** *spmof* (*prp-game*  $\mathcal{A}$ ) *True* = *spmof* (*run-gpv rp-bad*  $\mathcal{A}$  *init*) *True*

**unfolding** *vimage-def game-def if-True* **by** *transfer-prover*

**moreover** {

**define** *collision* ::  $('a \rightarrow 'a) \Rightarrow \text{bool}$  **where** *collision*  $m \iff \neg \text{inj-on } m (\text{dom } m)$  **for**  $m$

**have** [*simp*]:  $\neg \text{collision } \text{Map.empty}$  **by**(*simp add: collision-def*)

**have** [*simp*]:  $\llbracket \text{collision } m; m x = \text{None} \rrbracket \implies \text{collision } (m(x := y))$  **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*:  $\llbracket m\ x = \text{None}; y \in \text{ran } m \rrbracket \implies \text{collision } (m(x \mapsto y))$  **for**  $m\ x\ y$   
**by**(*auto simp add: collision-def ran-def intro: rev-image-eqI*)  
**have** *collision-map-upd-iff*:  $\neg \text{collision } m \implies \text{collision } (m(x \mapsto y)) \iff y \in \text{ran } m \wedge m\ x \neq \text{Some } y$  **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**  $?bad1 = \text{collision}$  **and**  $?bad2 = \text{fst}$   
**and**  $?X = \lambda \sigma 1\ (bad, \sigma 2). \sigma 1 = \sigma 2 \wedge \neg \text{collision } \sigma 1 \wedge \neg bad$   
**and**  $?I1 = \lambda \sigma 1. \text{dom } \sigma 1 \subseteq A \wedge \text{ran } \sigma 1 \subseteq A$   
**and**  $?I2 = \lambda (bad, \sigma 2). \text{dom } \sigma 2 \subseteq A \wedge \text{ran } \sigma 2 \subseteq A$   
**let**  $?X\text{-bad} = \lambda \sigma 1\ s2. ?I1\ \sigma 1 \wedge ?I2\ s2$   
**have** [*simp*]:  $\mathcal{S} \vdash c\ \text{rf.random-oracle } s1 \checkmark$  **if**  $\text{ran } s1 \subseteq A$  **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*  $?I1\ \mathcal{S}$   
**by**(*unfold-locales*)(*auto simp add: rf.random-oracle-def finite-A split: option.split-asm*)  
**then interpret** *rf: callee-invariant-on* *rf.random-oracle*  $?I1\ \mathcal{S}$  .  
**have** [*simp*]:  $\mathcal{S} \vdash c\ \text{rp-bad } s2 \checkmark$  **if**  $\text{ran } (\text{snd } s2) \subseteq A$  **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*  $(\lambda \sigma 1. ?bad1\ \sigma 1 \wedge ?I1\ \sigma 1)\ \mathcal{S}$   
**by**(*unfold-locales*)(*clarsimp simp add: rf.random-oracle-def finite-A split: option.split-asm*) +  
**have** [*simp*]: *callee-invariant-on* *rp-bad*  $(\lambda s2. ?I2\ s2)\ \mathcal{S}$   
**by**(*unfold-locales*)(*auto 4 3 simp add: rp-bad-simps finite-A split: option.splits if-split-asm iff del: domIff*)  
**have** [*simp*]: *callee-invariant-on* *rp-bad*  $(\lambda s2. ?bad2\ s2 \wedge ?I2\ s2)\ \mathcal{S}$   
**by**(*unfold-locales*)(*auto 4 3 simp add: rp-bad-simps finite-A split: option.splits if-split-asm iff del: domIff*)  
**have** [*simp*]:  $\mathcal{S} \vdash c\ \text{rp-bad } (bad, \sigma 2) \checkmark$  **if**  $\text{ran } \sigma 2 \subseteq A$  **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*  $(\text{rp-bad } (b, \sigma 2)\ x)$  **if**  $x \in A\ \text{dom } \sigma 2 \subseteq A\ \text{ran } \sigma 2 \subseteq A$  **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*  $(\lambda (b1, \sigma 1)\ (b2, \text{state2}). (?bad1\ \sigma 1 \iff ?bad2\ \text{state2}) \wedge (\text{if } ?bad2\ \text{state2} \text{ then } ?X\text{-bad } \sigma 1\ \text{state2} \text{ else } b1 = b2 \wedge ?X\ \sigma 1\ \text{state2}))$   
 $((\text{if } \text{False} \text{ then } \text{rp.random-permutation} \text{ else } \text{rf.random-oracle})\ s1\ x)\ (\text{rp-bad } s2\ x)$   
**if**  $?X\ s1\ s2\ x \in \text{outs-}\mathcal{S}\ \mathcal{S}\ ?I1\ s1\ ?I2\ s2$  **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-spmfI collision-map-updI dom-subset-ran-iff eq-None-iff-not-dom collision-map-upd-iff intro!: rel-spmf-bind-refl*)  
**with** - - **have** *rel-spmf*  
 $(\lambda (b1, \sigma 1)\ (b2, \text{state2}). (?bad1\ \sigma 1 \iff ?bad2\ \text{state2}) \wedge (\text{if } ?bad2\ \text{state2} \text{ then } ?X\text{-bad } \sigma 1\ \text{state2} \text{ else } b1 = b2 \wedge ?X\ \sigma 1\ \text{state2}))$   
 $(?run\ \text{False})\ (\text{exec-gpv } \text{rp-bad } \mathcal{S}\ \text{init})$   
**by**(*rule exec-gpv-oracle-bisim-bad-invariant[where*  $\mathcal{S} = \mathcal{S}$  **and**  $?II.0 = ?II$  **and**

```

?I2.0=?I2])(auto simp add: init-def WT lossless finite-A nonempty-A)
  then have |spmf (map-spmf fst (?run False)) True - spmf (run-gpv rp-bad  $\mathcal{A}$  init)
True|  $\leq$  spmf (map-spmf (?bad1  $\circ$  snd) (?run False)) True
  unfolding spmf-conv-measure-spmf measure-map-spmf vimage-def
  by(intro fundamental-lemma[where ?bad2.0= $\lambda(-, s2). ?bad2 s2]$ )(auto simp add:
split-def elim: rel-spmf-mono)
  also have ennreal ...  $\leq$  ennreal (q / card A) * (enat q) unfolding if-False using bound
----- WT
  by(rule rf.interaction-bounded-by-exec-gpv-bad-count[where count= $\lambda s. card (dom
s)]$ )
  (auto simp add: rf.random-oracle-def finite-A nonempty-A card-insert-if finite-subset[OF
-finite-A] map-spmf-conv-bind-spmf[symmetric] spmf.map-comp o-def collision-map-upd-iff
map-mem-spmf-of-set card-gt-0-iff card-mono field-simps Int-absorb2 intro: card-ran-le-dom[OF
finite-subset, OF - finite-A, THEN order-trans] split: option.splits)
  hence spmf (map-spmf (?bad1  $\circ$  snd) (?run False)) True  $\leq$  q * q / card A
  by(simp add: ennreal-of-nat-eq-real-of-nat ennreal-times-divide ennreal-mult''[symmetric])
  finally have |spmf (run-gpv rp-bad  $\mathcal{A}$  init) True - spmf (run-gpv rf.random-oracle  $\mathcal{A}$ 
Map.empty) True|  $\leq$  q * q / card A
  by simp }
  ultimately show ?thesis by(simp add: advantage-def game-def)
qed

end

end

```

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

This example is taken from [19, §4.2].

**theory PRF-UHF imports**

*CryptHOL.GPV-Bisim*

*Pseudo-Random-Function*

**begin**

**locale** hash =

**fixes** seed-gen :: 'seed spmf

**and** hash :: 'seed  $\Rightarrow$  'domain  $\Rightarrow$  'range

**begin**

**definition** game-hash :: 'domain  $\Rightarrow$  'domain  $\Rightarrow$  bool spmf

**where**

game-hash w w' = do {

seed  $\leftarrow$  seed-gen;

return-spmf (hash seed w = hash seed w'  $\wedge$  w  $\neq$  w')

}

**definition** game-hash-set :: 'domain set  $\Rightarrow$  bool spmf

**where**

```

game-hash-set W = do {
  seed ← seed-gen;
  return-spmf (¬ inj-on (hash seed) W)
}

```

**definition**  $\varepsilon\text{-uh} :: \text{real}$   
**where**  $\varepsilon\text{-uh} = (\text{SUP } w \ w'. \text{spmfmf } (\text{game-hash } w \ w') \ \text{True})$

**lemma**  $\varepsilon\text{-uh-nonneg} : \varepsilon\text{-uh} \geq 0$   
**by**(*auto* 4 3 *intro!*: *cSUP-upper2 bdd-aboveI2*[**where**  $M=1$ ] *cSUP-least pmf-le-1 pmf-nonneg simp add:  $\varepsilon\text{-uh-def}$* )

**lemma** *hash-ineq-card*:

**assumes** *finite W*

**shows**  $\text{spmfmf } (\text{game-hash-set } W) \ \text{True} \leq \varepsilon\text{-uh} * \text{card } W * \text{card } W$

**proof** –

**let**  $?M = \text{measure } (\text{measure-spmfmf } \text{seed-gen})$

**have** *bound*:  $?M \{x. \text{hash } x \ w = \text{hash } x \ w' \wedge w \neq w'\} \leq \varepsilon\text{-uh}$  **for**  $w \ w'$

**proof** –

**have**  $?M \{x. \text{hash } x \ w = \text{hash } x \ w' \wedge w \neq w'\} = \text{spmfmf } (\text{game-hash } w \ w') \ \text{True}$

**by**(*simp add: game-hash-def spmf-conv-measure-spmfmf-map-spmfmf-conv-bind-spmfmf*[*symmetric*]

*measure-map-spmfmf vimage-def*)

**also** **have**  $\dots \leq \varepsilon\text{-uh}$  **unfolding**  $\varepsilon\text{-uh-def}$

**by**(*auto intro!*: *cSUP-upper2 bdd-aboveI*[**where**  $M=1$ ] *cSUP-least simp add: pmf-le-1*)

**finally** **show** *?thesis* .

**qed**

**have**  $\text{spmfmf } (\text{game-hash-set } W) \ \text{True} = ?M \{x. \exists xa \in W. \exists y \in W. \text{hash } x \ xa = \text{hash } x \ y \wedge xa \neq y\}$

**by**(*auto simp add: game-hash-set-def inj-on-def map-spmfmf-conv-bind-spmfmf*[*symmetric*] *spmfmf-conv-measure-spmfmf measure-map-spmfmf vimage-def*)

**also** **have**  $\{x. \exists xa \in W. \exists y \in W. \text{hash } x \ xa = \text{hash } x \ y \wedge xa \neq y\} = (\bigcup (w, w') \in W \times W. \{x. \text{hash } x \ w = \text{hash } x \ w' \wedge w \neq w'\})$

**by**(*auto*)

**also** **have**  $?M \dots \leq (\sum (w, w') \in W \times W. ?M \{x. \text{hash } x \ w = \text{hash } x \ w' \wedge w \neq w'\})$

**by**(*auto intro!*: *measure-spmfmf.finite-measure-subadditive-finite simp add: split-def assms*)

**also** **have**  $\dots \leq (\sum (w, w') \in W \times W. \varepsilon\text{-uh})$  **by**(*rule sum-mono*)(*clarsimp simp add: bound*)

**also** **have**  $\dots = \varepsilon\text{-uh} * \text{card}(W) * \text{card}(W)$  **by**(*simp add: card-cartesian-product*)

**finally** **show** *?thesis* .

**qed**

**end**

**locale** *prf-hash* =

**fixes**  $f :: 'key \Rightarrow 'a \Rightarrow 'b$

**and**  $h :: 'seed \Rightarrow 'a \Rightarrow 'b$

**and**  $\text{key-gen} :: 'key \ \text{spmfmf}$

**and**  $\text{seed-gen} :: 'seed \ \text{spmfmf}$

**and**  $\text{range-f} :: ' \gamma \text{ set}$   
**assumes**  $\text{lossless-seed-gen}: \text{lossless-spmf seed-gen}$   
**and**  $\text{range-f-finite}: \text{finite range-f}$   
**and**  $\text{range-f-nonempty}: \text{range-f} \neq \{\}$   
**begin**

**definition**  $\text{rand} :: ' \gamma \text{ spmf}$   
**where**  $\text{rand} = \text{spmof-of-set range-f}$

**lemma**  $\text{lossless-rand [simp]}: \text{lossless-spmf rand}$   
**by**  $(\text{simp add}: \text{rand-def range-f-finite range-f-nonempty})$

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

**interpretation**  $\text{prf}: \text{prf key-gen f rand} .$   
**interpretation**  $\text{hash}: \text{hash seed-gen h} .$

**fun**  $f' :: ' \text{key} \times ' \text{seed} \Rightarrow ' \beta \Rightarrow ' \gamma$   
**where**  $f' (\text{key}, \text{seed}) x = f \text{key} (\text{h seed } x)$

**interpretation**  $\text{prf}': \text{prf key-seed-gen f' rand} .$

**definition**  $\text{reduction-oracle} :: ' \text{seed} \Rightarrow \text{unit} \Rightarrow ' \beta \Rightarrow (' \gamma \times \text{unit}, ' \alpha, ' \gamma) \text{ gpv}$   
**where**  $\text{reduction-oracle seed } x \text{ b} = \text{Pause} (\text{h seed } b) (\lambda x. \text{Done} (x, ()))$

**definition**  $\text{prf}'\text{-reduction} :: (' \beta, ' \gamma) \text{ prf}' . \text{adversary} \Rightarrow (' \alpha, ' \gamma) \text{ prf} . \text{adversary}$   
**where**  
 $\text{prf}'\text{-reduction } \mathcal{A} = \text{do} \{$   
 $\quad \text{seed} \leftarrow \text{lift-spmf seed-gen};$   
 $\quad (b, \sigma) \leftarrow \text{inline} (\text{reduction-oracle seed}) \mathcal{A} ();$   
 $\quad \text{Done } b$   
 $\}$

**theorem**  $\text{prf-prf}'\text{-advantage}:$   
**assumes**  $\text{prf}' . \text{lossless } \mathcal{A}$   
**and**  $\text{bounded}: \text{prf}' . \text{ibounded-by } \mathcal{A} \text{ } q$   
**shows**  $\text{prf}' . \text{advantage } \mathcal{A} \leq \text{prf} . \text{advantage} (\text{prf}'\text{-reduction } \mathcal{A}) + \text{hash} . \epsilon\text{-uh} * q * q$   
**including**  $\text{lifting-syntax}$

**proof** –  
**let**  $? \mathcal{A} = \text{prf}'\text{-reduction } \mathcal{A}$

**{ define**  $\text{cr}$  **where**  $\text{cr} = (\lambda - :: \text{unit} \times \text{unit}. \lambda - :: \text{unit}. \text{True})$   
**have**  $[\text{transfer-rule}]: \text{cr} ((), ()) ()$  **by**  $(\text{simp add}: \text{cr-def})$

```

have prf.game-0 ? $\mathcal{A}$  = prf'.game-0  $\mathcal{A}$ 
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 = ( $\lambda$  seed :: 'seed.  $\lambda$  ( $\sigma$  :: 'α → 'β ×
'γ, b :: bool).  $\lambda$  x.
  case  $\sigma$  (h seed x) of Some (a, y) ⇒ return-spmf (y, ( $\sigma$ , a ≠ x ∨ b))
  | None ⇒ bind-spmf rand ( $\lambda$  y. return-spmf (y, ( $\sigma$ (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)  $\mathcal{A}$  (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 = ( $\lambda$  (- :: unit,  $\sigma$ ) ( $\sigma'$  :: 'α ⇒ ('β × 'γ) option, - :: bool).  $\sigma$  = map-option
snd ∘  $\sigma'$ )
  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))
  ( $\lambda$  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-refl)
  have prf.game-1 ? $\mathcal{A}$  = 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)  $\mathcal{A}$  (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* ( $\lambda \sigma. \neg \text{inj-on } (h \text{ seed}) (\text{dom } \sigma)$ ) **for** *seed*  
**by** *unfold-locales(auto simp add: prf.random-oracle-def split: option.split-asm)*

**define** *invar*  
**where** *invar* = ( $\lambda \text{seed } (\sigma 1, b) (\sigma 2 :: 'b \Rightarrow 'c \text{ option}). \neg b \wedge \text{dom } \sigma 1 = h \text{ seed } \text{' dom } \sigma 2 \wedge$   
 $(\forall x \in \text{dom } \sigma 2. \sigma 1 (h \text{ seed } x) = \text{map-option } (\text{Pair } x) (\sigma 2 x))$ )

**have** *rel-spmf-oracle-adv*:  
 $\text{rel-spmf } (\lambda (x, s1) (y, s2). \text{snd } s1 \neq \text{inj-on } (h \text{ seed}) (\text{dom } s2) \wedge (\text{inj-on } (h \text{ seed}) (\text{dom } s2) \longrightarrow x = y \wedge \text{invar seed } s1 \text{ } s2))$   
 $(\text{exec-gpv } (\text{semi-forgetful-RO seed}) \mathcal{A} (\text{Map.empty}, \text{False}))$   
 $(\text{exec-gpv } \text{prf.random-oracle } \mathcal{A} \text{Map.empty})$   
**if** *seed*  $\in \text{set-spmf seed-gen}$  **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* =  $\lambda (a, s1) (b, s2 :: 'b \Rightarrow 'c \text{ option}).$   
 $\text{snd } s1 = (\neg \text{inj-on } (h \text{ seed}) (\text{dom } s2)) \wedge$   
 $(\neg \neg \text{inj-on } (h \text{ seed}) (\text{dom } s2) \longrightarrow a = b \wedge \text{invar seed } s1 \text{ } s2)$

**have** *step*: *rel-spmf ?R* (*semi-forgetful-RO seed*  $\sigma 1 b x$ ) (*prf.random-oracle* *s2 s*)  
**if** *X*: *invar seed*  $\sigma 1 b s2$  **for** *s2*  $\sigma 1 b x$   
**proof** –  
**obtain**  $\sigma 1 b$  **where** [*simp*]:  $\sigma 1 b = (\sigma 1, b)$  **by**(*cases*  $\sigma 1 b$ )  
**from** *X* **have** *not-b*:  $\neg b$   
**and** *dom*:  $\text{dom } \sigma 1 = h \text{ seed } \text{' dom } s2$   
**and** *eq*:  $\forall x \in \text{dom } s2. \sigma 1 (h \text{ seed } x) = \text{map-option } (\text{Pair } x) (s2 x)$   
**by**(*simp-all add: invar-def*)  
**from** *X* **have** *inj*: *inj-on* (*h seed*) (*dom s2*) **by**(*rule invarD-inj*)

**have** *not-in-image*:  $h \text{ seed } x \notin h \text{ seed } \text{' (dom } s2 - \{x\})$  **if**  $\sigma 1 (h \text{ seed } x) = \text{None}$   
**proof** (*rule notI*)  
**assume**  $h \text{ seed } x \in h \text{ seed } \text{' (dom } s2 - \{x\})$   
**then obtain** *y* **where**  $y \in \text{dom } s2$  **and** *hx-hy*:  $h \text{ seed } x = h \text{ seed } y$  **by** (*auto*)  
**then have**  $\sigma 1 (h \text{ seed } y) = \text{None}$  **using** *that* **by** (*auto*)  
**then have**  $h \text{ seed } y \notin h \text{ seed } \text{' dom } s2$  **using** *dom* **by** (*auto*)  
**then have**  $y \notin \text{dom } s2$  **by** (*auto*)  
**then show** *False* **using**  $\langle y \in \text{dom } s2 \rangle$  **by**(*auto*)  
**qed**

**show** *?thesis*



```

proof(cases  $\sigma 1$  (h seed x))
  case  $\sigma 1$ : None
  hence s2: s2 x = None using dom by(auto)
  have insert (h seed x) (dom  $\sigma 1$ ) = insert (h seed x) (h seed ' dom s2) by(simp add:
dom)
  then have invar-update: invar seed ( $\sigma 1$ (h seed x  $\mapsto$  (x, bs)), False) (s2(x  $\mapsto$  bs)) for
bs
    using inj not-b not-in-image  $\sigma 1$  dom
    by(auto simp add: invar-def domIff eq) (metis domI domIff imageI)
    with  $\sigma 1$  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-refl)
  next
  case  $\sigma 1$ : (Some by)
  show ?thesis
  proof(cases s2 x)
    case s2: (Some z)
    with eq  $\sigma 1$  have by = (x, z) by(auto simp add: domIff)
    thus ?thesis using  $\sigma 1$  inj not-b s2 X
    by(simp add: semi-forgetful-RO-def prf.random-oracle-def split-beta)
  next
  case s2: None
  from  $\sigma 1$  dom obtain y where y: y  $\in$  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  $\sigma 1$  have by: by = (y, z) by(auto simp add: * domIff)
  from y s2 have xny: x  $\neq$  y by auto
  with y * have h seed x  $\in$  h seed ' (dom s2 - {x}) by auto
  then show ?thesis using  $\sigma 1$  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 =  $\lambda \sigma. \neg$ 
inj-on (h seed) (dom  $\sigma$ )])
  (simp-all add: assms)
qed

define game-A where game-A = do {
  seed :: 'seed  $\leftarrow$  seed-gen;
  (b,  $\sigma$ )  $\leftarrow$  exec-gpv prf.random-oracle  $\mathcal{A}$  Map.empty;
  return-spmf (b,  $\neg$  inj-on (h seed) (dom  $\sigma$ ))
}

let ?bad1 =  $\lambda x. \text{snd } (\text{snd } x)$  and ?bad2 = snd
have hop3: rel-spmf ( $\lambda x xa. (?bad1 x \longleftrightarrow ?bad2 xa) \wedge (\neg ?bad2 xa \longrightarrow \text{fst } x \longleftrightarrow \text{fst } xa)$ )
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-refl simp del: bind-return-spmf)
  (rule rel-spmf-bindI[OF rel-spmf-oracle-adv]; auto)
have bad1-bad2: spmf (map-spmf (snd ∘ snd) game-semi-forgetful) True = spmf (map-spmf
snd game-A) True
using fundamental-lemma-bad[OF hop3] by(simp add: measure-map-spmf spmf-conv-measure-spmf
vimage-def)
have bound-bad1-event: |spmf (map-spmf fst game-semi-forgetful) True - spmf (map-spmf
fst game-A) True| ≤ spmf (map-spmf (snd ∘ snd) game-semi-forgetful) True
using fundamental-lemma[OF hop3] by(simp add: measure-map-spmf spmf-conv-measure-spmf
vimage-def)

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

define game-B where game-B = do {
  (b, σ) ← exec-gpv prf.random-oracle ℳ Map.empty;
  hash.game-hash-set (dom σ)
}

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

have game-B-bound : spmf game-B True ≤ hash.ε-uh * q * q unfolding game-B-def
proof(rule spmf-bind-leI, clarify)
  fix b σ
  assume *: (b, σ) ∈ set-spmf (exec-gpv prf.random-oracle ℳ Map.empty)
  have finite (dom σ) by(rule prf.finite.exec-gpv-invariant[OF *]; simp-all)
  then have spmf (hash.game-hash-set (dom σ)) True ≤ hash.ε-uh * (card (dom σ) *
card (dom σ))
    using hash.hash-ineq-card[of dom σ] by simp
  also have p1: card (dom σ) ≤ q + card (dom (Map.empty :: 'β ⇒ 'γ option))
    by(rule prf.card-dom-random-oracle[OF bounded *]; simp)
  then have card (dom σ) * card (dom σ) ≤ q * q using mult-le-mono by auto
  finally show spmf (hash.game-hash-set (dom σ)) True ≤ hash.ε-uh * q * q
    by(simp add: hash.ε-uh-nonneg mult-left-mono)
qed(simp add: hash.ε-uh-nonneg)

have hop4: prf'.game-1 ℳ = 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 ℳ ≤ |spmf (prf.game-0 ?ℳ) True - spmf (prf'.game-1 ℳ) True|
using hop1 by(simp add: prf'.advantage-def)
also have ... ≤ prf.advantage ?ℳ + |spmf (prf.game-1 ?ℳ) True - spmf (prf'.game-1
ℳ) 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*:  $\bigwedge xs\ ys. \llbracket \text{length } xs = \text{len}; \text{length } ys = \text{len} \rrbracket \implies \text{length } (f\ xs\ ys) = \text{len}$

**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*  $\longleftrightarrow$  *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*:  
 $\llbracket \text{length key} = \text{len}; \text{length plain} = \text{len} \rrbracket$   
 $\implies \text{encrypt key plain} \gg\equiv (\lambda \text{cipher. return-spmf (decrypt key cipher)}) = \text{return-spmf}$   
*(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*  $\Rightarrow$  *plain*  $\Rightarrow$  (*cipher*  $\times$  *unit*, *plain*, *plain*) *gpv*  
**where**

*prf-encrypt-oracle* *x plain* = *do* {  
 $r \leftarrow \text{lift-spmf (spmof-of-set (nlists UNIV len))}$ ;  
 $\text{Pause } r (\lambda \text{pad. Done ((r, xor-list plain pad), ()))$   
}

**lemma** *interaction-bounded-by-prf-encrypt-oracle* [*interaction-bound*]:

*interaction-any-bounded-by (prf-encrypt-oracle  $\sigma$  plain) 1*

**unfolding** *prf-encrypt-oracle-def* **by** *simp*

**lemma** *lossless-prf-encrypt-oracle* [*simp*]: *lossless-gpv  $\mathcal{I}$ -top (prf-encrypt-oracle *s x*)*

**by** (*simp add: prf-encrypt-oracle-def*)

**definition** *prf-adversary* :: (*plain*, *cipher*, 'state) *ind-cpa.adversary*  $\Rightarrow$  (*plain*, *plain*) *prf.adversary*  
**where**

*prf-adversary*  $\mathcal{A}$  = *do* {  
 $\text{let } (\mathcal{A}1, \mathcal{A}2) = \mathcal{A}$ ;  
 $((p1, p2), \sigma), n \leftarrow \text{inline prf-encrypt-oracle } \mathcal{A}1 ()$ ;  
 $\text{if valid-plain } p1 \wedge \text{valid-plain } p2 \text{ then do}$  {  
 $b \leftarrow \text{lift-spmf coin-spmf}$ ;  
 $\text{let } pb = (\text{if } b \text{ then } p1 \text{ else } p2)$ ;  
 $r \leftarrow \text{lift-spmf (spmof-of-set (nlists UNIV len))}$ ;  
 $\text{pad} \leftarrow \text{Pause } r \text{ Done}$ ;  
 $\text{let } c = (r, \text{xor-list } pb \text{ pad})$ ;  
 $(b', -) \leftarrow \text{inline prf-encrypt-oracle } (\mathcal{A}2 \text{ } c \text{ } \sigma) \text{ } n$ ;  
 $\text{Done } (b' = b)$   
}  $\text{else lift-spmf coin-spmf}$   
}

**theorem** *prf-encrypt-advantage*:

**assumes** *ind-cpa.ibounded-by*  $\mathcal{A} \ q$

**and** *lossless-gpv  $\mathcal{I}$ -full (fst  $\mathcal{A}$ )*

**and**  $\bigwedge \text{cipher } \sigma. \text{lossless-gpv } \mathcal{I}\text{-full (snd } \mathcal{A} \text{ cipher } \sigma)$

**shows** *ind-cpa.advantage*  $\mathcal{A} \leq \text{prf.advantage (prf-adversary } \mathcal{A}) + q / 2^{\wedge} \text{len}$

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

let ?oracle1 =  $\lambda \text{key } (s', s) \ y. \text{ map-spmf } (\lambda ((x, s'), s). (x, (), ())) (\text{exec-gpv } (\text{prf.prf-oracle}$ 
key) (prf.encrypt-oracle () y) ())
have  $\text{bisim1}$ :  $\bigwedge \text{key}. \text{ rel-spmf } (\lambda (x, -) (y, -). x = y)$ 
  (exec-gpv (ind-cpa.encrypt-oracle key)  $\mathcal{A}1$  ())
  (exec-gpv (?oracle1 key)  $\mathcal{A}1$  ((), ()))
  using True1
  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  $\text{bisim2}$ :  $\bigwedge \text{key } \text{cipher } \sigma. \text{ rel-spmf } (\lambda (x, -) (y, -). x = y)$ 
  (exec-gpv (ind-cpa.encrypt-oracle key) ( $\mathcal{A}2 \ \text{cipher } \sigma$ ) ())
  (exec-gpv (?oracle1 key) ( $\mathcal{A}2 \ \text{cipher } \sigma$ ) ((), ()))
  using True1
  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  $\text{ind-cpa-0}$ : rel-spmf (=) (ind-cpa.ind-cpa  $\mathcal{A}$ ) (prf.game-0 (prf.adversary  $\mathcal{A}$ ))
unfolding IND-CPA.ind-cpa.ind-cpa-def  $\mathcal{A}$  key-gen-def Let-def prf.adversary-def Pseudo-Random-Function.prf.gam
apply(simp)
apply(rewrite in bind-spmf -  $\square$  bind-commute-spmf)
apply(rule rel-spmf-bind-refl)
apply(rule rel-spmf-bindI[OF bisim1])
apply(clarsimp simp add: if-distrib bind-coin-spmf-eq-const')
apply(auto intro: rel-spmf-bindI[OF bisim2] intro!: rel-spmf-bind-refl simp add: en
crypt-def prf.prf-oracle-def cong del: if-cong)
done

define rf-encrypt where rf-encrypt = ( $\lambda s \ \text{plain}. \text{ bind-spmf } (\text{spmf-of-set } (\text{nlists UNIV}$ 
len)) ( $\lambda r :: \text{bool list}.$ 
  bind-spmf (prf.random-oracle  $s \ r$ ) ( $\lambda (\text{pad}, s')$ 

```

```

    return-spmf ((r, xor-list plain pad), s'))
  )
interpret rf-finite: callee-invariant-on rf-encrypt  $\lambda s$ . finite (dom s)  $\mathcal{A}$ -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),  $\sigma$ ), s1)  $\leftarrow$  exec-gpv rf-encrypt  $\mathcal{A}1$  Map.empty;
  if valid-plain p0  $\wedge$  valid-plain p1 then do {
    b  $\leftarrow$  coin-spmf;
    let pb = (if b then p0 else p1);
    (cipher, s2)  $\leftarrow$  rf-encrypt s1 pb;
    (b', s3)  $\leftarrow$  exec-gpv rf-encrypt ( $\mathcal{A}2$  cipher  $\sigma$ ) s2;
    return-spmf (b' = b)
  } else coin-spmf
}

let ?oracle2 =  $\lambda (s', s)$  y. map-spmf ( $\lambda ((x, s'), s)$ . (x, (), s)) (exec-gpv prf.random-oracle
(prf-encrypt-oracle () y) s)
let ?I =  $\lambda (x, -, s)$  (y, s'). x = y  $\wedge$  s = s'
have bisim1: rel-spmf ?I (exec-gpv ?oracle2  $\mathcal{A}1$  ((), Map.empty)) (exec-gpv rf-encrypt
 $\mathcal{A}1$  Map.empty)
by(rule exec-gpv-oracle-bisim[where X= $\lambda (-, s)$  s'. s = s'])
(auto simp add: rf-encrypt-def prf-encrypt-oracle-def intro!: rel-spmf-bind-refl)
have bisim2:  $\bigwedge$  cipher  $\sigma$  s. rel-spmf ?I (exec-gpv ?oracle2 ( $\mathcal{A}2$  cipher  $\sigma$ ) ((), s))
(exec-gpv rf-encrypt ( $\mathcal{A}2$  cipher  $\sigma$ ) s)
by(rule exec-gpv-oracle-bisim[where X= $\lambda (-, s)$  s'. s = s'])
(auto simp add: prf-encrypt-oracle-def rf-encrypt-def intro!: rel-spmf-bind-refl)
have game1-2 [unfolded spmf-rel-eq]: rel-spmf (=) (prf.game-1 (prf-adversary  $\mathcal{A}$ ))
game2
unfolding prf.game-1-def game2-def prf-adversary-def
by(rewrite in if-then  $\exists$  else - rf-encrypt-def)
(auto simp add: Let-def  $\mathcal{A}$  if-distrib intro!: rel-spmf-bindI[OF bisim2] rel-spmf-bind-refl
rel-spmf-bindI[OF bisim1])

define game2-a where game2-a = do {
  r  $\leftarrow$  spmf-of-set (nlists UNIV len);
  (((p0, p1),  $\sigma$ ), s1)  $\leftarrow$  exec-gpv rf-encrypt  $\mathcal{A}1$  Map.empty;
  let bad = r  $\in$  dom s1;
  if valid-plain p0  $\wedge$  valid-plain p1 then do {
    b  $\leftarrow$  coin-spmf;
    let pb = (if b then p0 else p1);
    (pad, s2)  $\leftarrow$  prf.random-oracle s1 r;
    let cipher = (r, xor-list pb pad);
    (b', s3)  $\leftarrow$  exec-gpv rf-encrypt ( $\mathcal{A}2$  cipher  $\sigma$ ) s2;
    return-spmf (b' = b, bad)
  } else coin-spmf  $\gg$  ( $\lambda b$ . 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  $\mathcal{A}1$  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 ( $\mathcal{A}2$  cipher σ) (s1(r ↦ pad));
    return-spmf (b' = b, bad)
  } else coin-spmf  $\gg\equiv$  ( $\lambda b.$  return-spmf (b, bad))
}

have game2 = do {
  r ← spmf-of-set (nlists UNIV len);
  ((p0, p1), σ), s1 ← exec-gpv rf-encrypt  $\mathcal{A}1$  Map.empty;
  if valid-plain p0 ∧ valid-plain p1 then do {
    b ← coin-spmf;
    let pb = (if b then p0 else p1);
    (pad, s2) ← prf.random-oracle s1 r;
    let cipher = (r, xor-list pb pad);
    (b', s3) ← exec-gpv rf-encrypt ( $\mathcal{A}2$  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-distrib 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-distrib split-def restrict-return-spmf
prf.random-oracle-def intro!: rel-spmf-bind-refl 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 ⊘ 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:  $\bigwedge B :: \text{bool list set}. \text{card} (B \cap \text{nlists UNIV len}) \leq \text{card} (\text{nlists UNIV len} :: \text{bool list set})$ 
by(rule card-mono) simp-all
then have spmf game2-a-bad True =  $\int^+ x. \text{card} (\text{dom} (\text{snd } x) \cap \text{nlists UNIV len}) / 2^{\text{len}} \partial \text{measure-spmf} (\text{exec-gpv rf-encrypt } \text{⊘} 1 \text{ Map.empty})$ 
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 ⊘ 1 Map.empty)
hence finite (dom s) by(rule rf-finite.exec-gpv-invariant) simp-all
hence 1:  $\text{card} (\text{dom } s \cap \text{nlists UNIV len}) \leq \text{card} (\text{dom } s)$  by(intro card-mono) simp-all
moreover from q1 *
have  $\text{card} (\text{dom } s) \leq q1 + \text{card} (\text{dom} (\text{Map.empty} :: (\text{plain}, \text{plain}) \text{prf.dict}))$ 
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  $\text{card} (\text{dom } s \cap \text{nlists UNIV len}) \leq q1$  by(simp) }
then have ... ≤  $\int^+ x. q1 / 2^{\text{len}} \partial \text{measure-spmf} (\text{exec-gpv rf-encrypt } \text{⊘} 1 \text{ Map.empty})$ 
by(intro nn-integral-mono-AE)(clarsimp simp add: field-simps)
also have ... ≤  $q1 / 2^{\text{len}}$ 
by(simp add: measure-spmf.emmeasure-eq-measure field-simps mult-left-le weight1)
finally have game2a-bad-bound: spmf game2-a-bad True ≤  $q1 / 2^{\text{len}}$  by simp

define rf-encrypt-bad
where rf-encrypt-bad = ( $\lambda \text{secret} (s :: (\text{plain}, \text{plain}) \text{prf.dict}, \text{bad}) \text{plain}. \text{bind-spmf} (\text{spmf-of-set} (\text{nlists UNIV len})) (\lambda r. \text{bind-spmf} (\text{prf.random-oracle } s r) (\lambda (\text{pad}, s'). \text{return-spmf} ((r, \text{xor-list plain pad}), (s', \text{bad} \vee r = \text{secret}))))))$ )
have rf-encrypt-bad-sticky [simp]:  $\bigwedge s. \text{callee-invariant} (\text{rf-encrypt-bad } s) \text{snd}$ 
by(unfold-locales)(auto simp add: rf-encrypt-bad-def)
have lossless-rf-encrypt [simp]:  $\bigwedge \text{challenge } s \text{ plain}. \text{lossless-spmf} (\text{rf-encrypt-bad challenge } s \text{ plain})$ 
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 ⊘ 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-refl)

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-distrib intro!:
rel-spmf-bind-refl 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-refl
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-distrib intro!*:  
*rel-spmf-bind-refl 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-distrib split-def restrict-return-spmf intro!*: *rel-spmf-bind-refl 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  $\sigma$  and challenge :: plain and s**

**have** *card (nlists UNIV len  $\cap$  ( $\lambda x. x = \text{challenge}$ ) - ' {True}) ≤ card {challenge}*

**by**(*rule card-mono*) *auto*

**then have** *spmf (map-spmf (snd  $\circ$  snd) (exec-gpv (rf-encrypt-bad challenge) ( $\mathcal{A}2$  cipher  $\sigma$ ) (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** ( $\int^+ x. \text{ennreal (indicator \{True\} x) } \partial \text{measure-spmf (map-spmf (snd } \circ \text{ snd) (exec-gpv (rf-encrypt-bad challenge) (\mathcal{A}2 \text{ cipher } \sigma) (s, False)))} \leq (1 / 2 ^ \text{len}) * q2$ )

**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 ≤*

$\int^+ (r :: \text{plain}). \int^+ (((p0, p1), \sigma), s). (\text{if valid-plain } p0 \wedge \text{valid-plain } p1 \text{ then } \int^+ b. \int^+ (\text{pad} :: \text{plain}). q2 / 2 ^ \text{len} \partial \text{measure-spmf (spmf-of-set (nlists UNIV len))} \partial \text{measure-spmf coin-spmf} \text{ else } 0)$

$\partial \text{measure-spmf (exec-gpv rf-encrypt } \mathcal{A}1 \text{ Map.empty)} \partial \text{measure-spmf (spmf-of-set (nlists UNIV len))}$

**unfolding** *game2-d-def*

**by**(*simp add: ennreal-spmf-bind o-def split-def Let-def if-distrib if-distrib[where  $f = \lambda x. \text{ennreal (spmf } x \text{ -)}$ ] indicator-single-Some nn-integral-mono if-mono-cong del: nn-integral-const cong: if-cong*)

**also have**  $\dots \leq \int^+ (r :: \text{plain}). \int^+ (((p0, p1), \sigma), s). (\text{if valid-plain } p0 \wedge \text{valid-plain } p1 \text{ then } \text{ennreal (} q2 / 2 ^ \text{len) else } q2 / 2 ^ \text{len}) \partial \text{measure-spmf (exec-gpv rf-encrypt } \mathcal{A}1 \text{ Map.empty)} \partial \text{measure-spmf (spmf-of-set (nlists UNIV len))}$

**unfolding** *split-def*

**by**(*intro nn-integral-mono if-mono-cong*)(*auto simp add: measure-spmf.emeasure-eq-measure*)

**also have**  $\dots \leq q2 / 2 ^ \text{len}$  **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),  $\sigma$ ), s1)  $\leftarrow$  exec-gpv rf-encrypt  $\mathcal{A}1$  Map.empty;
  if valid-plain p0  $\wedge$  valid-plain p1 then do {
    b  $\leftarrow$  coin-spmf;
    let pb = (if b then p0 else p1);
    r  $\leftarrow$  spmf-of-set (nlists UNIV len);
    pad  $\leftarrow$  spmf-of-set (nlists UNIV len);
    let cipher = (r, xor-list pb pad);
    (b', s2)  $\leftarrow$  exec-gpv rf-encrypt ( $\mathcal{A}2$  cipher  $\sigma$ ) s1;
    return-spmf (b' = b)
  } else coin-spmf
}
have bisim2d-3:  $\bigwedge$  cipher  $\sigma$  s r. rel-spmf ( $\lambda(x, -) (y, -). x = y$ )
  (exec-gpv (rf-encrypt-bad r) ( $\mathcal{A}2$  cipher  $\sigma$ ) (s, False))
  (exec-gpv rf-encrypt ( $\mathcal{A}2$  cipher  $\sigma$ ) s)
by (rule exec-gpv-oracle-bisim[where X= $\lambda(s1, -) s2. s1 = s2$ ])(auto simp add: rf-encrypt-bad-def
rf-encrypt-def intro!: rel-spmf-bind-refl)
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-distrib cong: if-cong intro!: rel-spmf-bind-refl
rel-spmf-bindI[OF bisim2d-3])

have |spmf game2 True - 1 / 2|  $\leq$ 
  |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 ...  $\leq$  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|  $\leq$ 
  |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 ...  $\leq$  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|  $\leq$  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),  $\sigma$ ), s1)  $\leftarrow$  exec-gpv rf-encrypt  $\mathcal{A}1$  Map.empty;
  if valid-plain p0  $\wedge$  valid-plain p1 then do {
    b  $\leftarrow$  coin-spmf;
    let pb = (if b then p0 else p1);
    r  $\leftarrow$  spmf-of-set (nlists UNIV len);
    pad  $\leftarrow$  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*  $\mathcal{A}$   $q$   
**shows** *prf.ibounded-by* (*prf-adversary*  $\mathcal{A}$ )  $(1 + q)$   
**proof** –  
**fix**  $\eta$   
**from** *assms* **have** *ind-cpa.ibounded-by*  $\mathcal{A}$   $q$  **by** *blast*  
**then obtain**  $q1$   $q2$  **where**  $q: q1 + q2 \leq q$   
**and** [*interaction-bound*]: *interaction-any-bounded-by* (*fst*  $\mathcal{A}$ )  $q1$   
 $\wedge x \sigma. \text{interaction-any-bounded-by}$  (*snd*  $\mathcal{A}$   $x$   $\sigma$ )  $q2$   
**unfolding** *ind-cpa.ibounded-by-def* **by** (*auto simp add: split-beta iadd-le-enat-iff*)  
**show** *prf.ibounded-by* (*prf-adversary*  $\mathcal{A}$ )  $(1 + q)$  **using**  $q$   
**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*  $\mathcal{A} \implies \text{prf.lossless}$  (*prf-adversary*  $\mathcal{A}$ )  
**by** (*fastforce simp add: prf-adversary-def Let-def split-def ind-cpa.lossless-def intro: lossless-inline*)

**end**

**locale** *otp- $\eta$*  =  
**fixes**  $f :: \text{security} \Rightarrow \text{key} \Rightarrow \text{bool list} \Rightarrow \text{bool list}$   
**and**  $len :: \text{security} \Rightarrow \text{nat}$   
**assumes** *length-f*:  $\wedge \eta \text{ xs ys. } [\text{length xs} = \text{len } \eta; \text{length ys} = \text{len } \eta] \implies \text{length} (f \ \eta \ \text{xs} \ \text{ys}) = \text{len } \eta$   
**and** *negligible-len* [*negligible-intros*]: *negligible*  $(\lambda \eta. 1 / 2 ^ (\text{len } \eta))$   
**begin**

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

**lemma** *prf-encrypt-secure-for*:  
**assumes** [*negligible-intros*]: *negligible*  $(\lambda \eta. \text{prf.advantage } \eta \ (\text{prf-adversary } \eta \ (\mathcal{A} \ \eta)))$   
**and**  $q: \wedge \eta. \text{ind-cpa.ibounded-by}$   $(\mathcal{A} \ \eta) \ (q \ \eta)$  **and** [*negligible-intros*]: *polynomial*  $q$   
**and** *lossless*:  $\wedge \eta. \text{ind-cpa.lossless}$   $(\mathcal{A} \ \eta)$   
**shows** *negligible*  $(\lambda \eta. \text{ind-cpa.advantage } \eta \ (\mathcal{A} \ \eta))$   
**proof** (*rule negligible-mono*)  
**show** *negligible*  $(\lambda \eta. \text{prf.advantage } \eta \ (\text{prf-adversary } \eta \ (\mathcal{A} \ \eta)) + q \ \eta / 2 ^ \text{len } \eta)$   
**by** (*intro negligible-intros*)  
**{ fix**  $\eta$   
**from**  $\langle \text{ind-cpa.ibounded-by} \ - \ - \rangle$  **have** *ind-cpa.ibounded-by*  $(\mathcal{A} \ \eta) \ (q \ \eta)$  **by** *blast*  
**moreover from** *lossless* **have** *ind-cpa.lossless*  $(\mathcal{A} \ \eta)$  **by** *blast*  
**hence** *lossless-gpv*  $\mathcal{I}$ -*full* (*fst*  $(\mathcal{A} \ \eta)$ )  $\wedge$  *cipher*  $\sigma. \text{lossless-gpv}$   $\mathcal{I}$ -*full* (*snd*  $(\mathcal{A} \ \eta)$  *cipher*  $\sigma$ )

```

    by(auto simp add: ind-cpa.lossless-def)
    ultimately have ind-cpa.advantage  $\eta$  ( $\mathcal{A}$   $\eta$ )  $\leq$  prf.advantage  $\eta$  (prf-adversary  $\eta$  ( $\mathcal{A}$ 
 $\eta$ )) +  $q \eta / 2^{\text{len } \eta}$ 
    by(rule prf-encrypt-advantage) }
    hence eventually ( $\lambda \eta. |\text{ind-cpa.advantage } \eta$  ( $\mathcal{A}$   $\eta$ )|  $\leq 1 * |\text{prf.advantage } \eta$  (prf-adversary
 $\eta$  ( $\mathcal{A}$   $\eta$ )) +  $q \eta / 2^{\text{len } \eta}$ ) at-top
    by(simp add: always-eventually ind-cpa.advantage-nonneg prf.advantage-nonneg)
    then show ( $\lambda \eta. \text{ind-cpa.advantage } \eta$  ( $\mathcal{A}$   $\eta$ ))  $\in O(\lambda \eta. \text{prf.advantage } \eta$  (prf-adversary
 $\eta$  ( $\mathcal{A}$   $\eta$ )) +  $q \eta / 2^{\text{len } \eta}$ )
    by(intro bigO[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  $\Rightarrow$  bitstring  $\Rightarrow$  bitstring
  and prf-domain :: bitstring set
  and prf-range :: bitstring set
  and prf-dlen :: nat
  and prf-clen :: nat
  and upf-key-gen :: 'upf-key spmf
  and upf-fun :: 'upf-key  $\Rightarrow$  bitstring  $\Rightarrow$  'hash
  +
  assumes prf-domain-finite: finite prf-domain
  assumes prf-domain-nonempty: prf-domain  $\neq$  {}
  assumes prf-domain-length:  $x \in \text{prf-domain} \implies \text{length } x = \text{prf-dlen}$ 
  assumes prf-codomain-length:
     $\llbracket \text{key-prf} \in \text{set-spmf } \text{prf-key-gen}; m \in \text{prf-domain} \rrbracket \implies \text{length } (\text{prf-fun } \text{key-prf } m) =$ 
    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* ≫≡ (*λc. 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 λm. length m = prf-clen .*  
**interpretation** *ind-cca'*: *ind-cca key-gen encrypt λ - . None λm. length m = prf-clen .*

**definition** *intercept-upf-enc*

$:: 'prf\text{-}key \Rightarrow bool \Rightarrow 'hash\text{ cipher-text set} \times 'hash\text{ cipher-text set} \Rightarrow bitstring \times bitstring$   
 $\Rightarrow ('hash\text{ cipher-text option} \times ('hash\text{ cipher-text set} \times 'hash\text{ cipher-text set}),$   
 $bitstring + (bitstring \times 'hash), 'hash + unit) gpv$

**where**

$intercept\text{-}upf\text{-}enc\ k\ b = (\lambda(L, D)\ (m1, m0).$   
 $(case\ (length\ m1 = prf\text{-}clen \wedge length\ m0 = prf\text{-}clen)\ of$   
 $False \Rightarrow Done\ (None, L, D)$   
 $| True \Rightarrow do\ \{$   
 $x \leftarrow lift\text{-}spmf\ (spmf\text{-}of\text{-}set\ prf\text{-}domain);$   
 $let\ c = prf\text{-}fun\ k\ x\ [\oplus]\ (if\ b\ then\ m1\ else\ m0);$   
 $t \leftarrow Pause\ (Inl\ (x\ @\ c))\ Done;$   
 $Done\ ((Some\ (x, c, proj1\ t)), (insert\ (x, c, proj1\ t)\ L, D))$   
 $\})$

**definition** *intercept-upf-dec*

$:: 'hash\text{ cipher-text set} \times 'hash\text{ cipher-text set} \Rightarrow 'hash\text{ cipher-text}$   
 $\Rightarrow (bitstring\ option \times ('hash\text{ cipher-text set} \times 'hash\text{ cipher-text set}),$   
 $bitstring + (bitstring \times 'hash), 'hash + unit) gpv$

**where**

$intercept\text{-}upf\text{-}dec = (\lambda(L, D)\ (x, c, t).$   
 $if\ (x, c, t) \in L \vee length\ x \neq prf\text{-}dlen\ then\ Done\ (None, (L, D))\ else\ do\ \{$   
 $Pause\ (Inr\ (x\ @\ c, t))\ Done;$   
 $Done\ (None, (L, insert\ (x, c, t)\ D))$   
 $\})$

**definition** *intercept-upf* ::

$'prf\text{-}key \Rightarrow bool \Rightarrow 'hash\text{ cipher-text set} \times 'hash\text{ cipher-text set} \Rightarrow bitstring \times bitstring$   
 $+ 'hash\text{ cipher-text}$   
 $\Rightarrow (('hash\text{ cipher-text option} + bitstring\ option) \times ('hash\text{ cipher-text set} \times 'hash\text{ cipher-text set}),$   
 $bitstring + (bitstring \times 'hash), 'hash + unit) gpv$

**where**

$intercept\text{-}upf\ k\ b = plus\text{-}intercept\ (intercept\text{-}upf\text{-}enc\ k\ b)\ intercept\text{-}upf\text{-}dec$

**lemma** *intercept-upf-simps* [simp]:

$intercept\text{-}upf\ k\ b\ (L, D)\ (Inr\ (x, c, t)) =$   
 $(if\ (x, c, t) \in L \vee length\ x \neq prf\text{-}dlen\ then\ Done\ (Inr\ None, (L, D))\ else\ do\ \{$   
 $Pause\ (Inr\ (x\ @\ c, t))\ Done;$   
 $Done\ (Inr\ None, (L, insert\ (x, c, t)\ D))$   
 $\})$   
 $intercept\text{-}upf\ k\ b\ (L, D)\ (Inl\ (m1, m0)) =$   
 $(case\ (length\ m1 = prf\text{-}clen \wedge length\ m0 = prf\text{-}clen)\ of$   
 $False \Rightarrow Done\ (Inl\ None, L, D)$   
 $| True \Rightarrow do\ \{$   
 $x \leftarrow lift\text{-}spmf\ (spmf\text{-}of\text{-}set\ prf\text{-}domain);$   
 $let\ c = prf\text{-}fun\ k\ x\ [\oplus]\ (if\ b\ then\ m1\ else\ m0);$   
 $t \leftarrow Pause\ (Inl\ (x\ @\ c))\ Done;$   
 $Done\ (Inl\ (Some\ (x, c, proj1\ t)), (insert\ (x, c, proj1\ t)\ L, D))$   
 $\})$



**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*  $\circ$  *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*  $\circ$  *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*  $\circ$  *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*  $\implies$  *interaction-bounded-by* (*Not*  $\circ$  *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*)  $\subseteq$  *responses- $\mathcal{I}$*  ( *$\mathcal{I}$ -full*  $\oplus_{\mathcal{I}}$   *$\mathcal{I}$ -full*) *x*  $\times$  *UNIV*

**by**(*cases x*)(*auto simp add: intercept-upf-def*)

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

$\implies$  (*bitstring*, *'hash*) *UPF.adversary*

**where** *reduction-upf*  *$\mathcal{A}$*  = *do* {

*k*  $\leftarrow$  *lift-spmf prf-key-gen*;

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

(*-*, (*L*, *D*))  $\leftarrow$  *inline* (*intercept-upf* *k b*)  *$\mathcal{A}$*  (*{}*, *{}*);

*Done* (*()*) }

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

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

**lemma** *round-1*:

**assumes** *lossless-gpv* ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full)  $\mathcal{A}$   
**shows**  $|\text{spmf}$  (*ind-cca.game*  $\mathcal{A}$ ) *True*  $- \text{spmf}$  (*ind-cca'.game*  $\mathcal{A}$ ) *True*  $|\leq \text{UPF.}\text{advantage}$  (*reduction-upf*  $\mathcal{A}$ )

**proof** –

**define** *oracle-decrypt0'* **where** *oracle-decrypt0'*  $\equiv (\lambda \text{key } (\text{bad}, L) (x', c', t'). \text{return-spmf}$   
 ( $\text{if } (x', c', t') \in L \vee \text{length } x' \neq \text{prf-dlen}$  then (*None*, (*bad*, *L*))

$\text{else } (\text{decrypt key } (x', c', t'), (\text{bad} \vee \text{upf-fun } (\text{snd key}) (x' @ c') = t', L)))$

**have** *oracle-decrypt0'-simps*:

*oracle-decrypt0' key* (*bad*, *L*) ( $x', c', t'$ ) = *return-spmf* ( $\text{if } (x', c', t') \in L \vee \text{length } x' \neq \text{prf-dlen}$  then (*None*, (*bad*, *L*))  
 $\text{else } (\text{decrypt key } (x', c', t'), (\text{bad} \vee \text{upf-fun } (\text{snd key}) (x' @ c') = t', L)))$

**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'* =  $(\lambda (\text{key} :: \text{'prf-key} \times \text{'upf-key}) (\text{bad}, L) (x', c', t').$

*return-spmf* (*None* :: *bitstring option*,  
 $(\text{bad} \vee \text{upf-fun } (\text{snd key}) (x' @ c') = t' \wedge (x', c', t') \notin L \wedge \text{length } x' = \text{prf-dlen}, L))$

**have** *oracle-decrypt1'-simps*:

*oracle-decrypt1' key* (*bad*, *L*) ( $x', c', t'$ ) =  
*return-spmf* (*None*,  
 $(\text{bad} \vee \text{upf-fun } (\text{snd key}) (x' @ c') = t' \wedge (x', c', t') \notin L \wedge \text{length } x' = \text{prf-dlen}, L))$

**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'* =  $(\lambda (\text{decrypt} :: \text{'prf-key} \times \text{'upf-key} \Rightarrow (\text{bitstring} \times \text{bitstring} \times \text{'hash},$   
*bitstring option, bool} \times (\text{bitstring} \times \text{bitstring} \times \text{'hash} \text{ set}) \text{ callee}) \mathcal{A}. \text{do} {*

$\text{key} \leftarrow \text{key-gen};$

$b \leftarrow \text{coin-spmf};$

$(b', (bad', L')) \leftarrow exec\text{-}gpv (\dagger(ind\text{-}cca.oracle\text{-}encrypt\ key\ b) \oplus_O decrypt\ key) \mathcal{A} (False, \{\});$

$return\text{-}spmf\ (b = b', bad')$

**let**  $?game0' = game01'\ oracle\text{-}decrypt0'$

**let**  $?game1' = game01'\ oracle\text{-}decrypt1'$

**have**  $game0'\text{-}eq: ind\text{-}cca.game\ \mathcal{A} = map\text{-}spmf\ fst\ (?game0'\ \mathcal{A})$  (**is**  $?game0$ )

**and**  $game1'\text{-}eq: ind\text{-}cca'.game\ \mathcal{A} = map\text{-}spmf\ fst\ (?game1'\ \mathcal{A})$  (**is**  $?game1$ )

**proof** –

**let**  $?S = rel\text{-}prod2\ (=)$

**define**  $initial$  **where**  $initial = (False, \{\} :: 'hash\ cipher\text{-}text\ set)$

**have**  $[transfer\text{-}rule]: ?S\ \{\} initial$  **by**  $(simp\ add: initial\text{-}def)$

**have**  $[transfer\text{-}rule]:$

$((=) \implies ?S \implies (=) \implies rel\text{-}spmf\ (rel\text{-}prod\ (=)\ ?S))$

$ind\text{-}cca.oracle\text{-}decrypt\ oracle\text{-}decrypt0'$

**unfolding**  $ind\text{-}cca.oracle\text{-}decrypt\text{-}def[abs\text{-}def]\ oracle\text{-}decrypt0'\text{-}def[abs\text{-}def]$

**by**  $(simp\ add: rel\text{-}spmf\text{-}return\text{-}spmf1\ rel\text{-}fun\text{-}def)$

**have**  $[transfer\text{-}rule]:$

$((=) \implies ?S \implies (=) \implies rel\text{-}spmf\ (rel\text{-}prod\ (=)\ ?S))$

$ind\text{-}cca'.oracle\text{-}decrypt\ oracle\text{-}decrypt1'$

**unfolding**  $ind\text{-}cca'.oracle\text{-}decrypt\text{-}def[abs\text{-}def]\ oracle\text{-}decrypt1'\text{-}def[abs\text{-}def]$

**by**  $(simp\ add: rel\text{-}spmf\text{-}return\text{-}spmf1\ rel\text{-}fun\text{-}def)$

**note**  $[transfer\text{-}rule] = extend\text{-}state\text{-}oracle\text{-}transfer$

**show**  $?game0\ ?game1$  **unfolding**  $game01'\text{-}def\ ind\text{-}cca.game\text{-}def\ ind\text{-}cca'.game\text{-}def\ initial\text{-}def[symmetric]$

**by**  $(simp\text{-}all\ add: map\text{-}spmf\text{-}bind\text{-}spmf\ o\text{-}def\ split\text{-}def)\ transfer\text{-}prover+$

**qed**

**have**  $*$ :  $rel\text{-}spmf\ (\lambda(b'1, (bad1, L1)) (b'2, (bad2, L2)). bad1 = bad2 \wedge (\neg bad2 \longrightarrow b'1 = b'2))$

$(exec\text{-}gpv (\dagger(ind\text{-}cca.oracle\text{-}encrypt\ k\ b) \oplus_O oracle\text{-}decrypt1'\ k) \mathcal{A} (False, \{\}))$

$(exec\text{-}gpv (\dagger(ind\text{-}cca.oracle\text{-}encrypt\ k\ b) \oplus_O oracle\text{-}decrypt0'\ k) \mathcal{A} (False, \{\}))$

**for**  $k\ b$

**by**  $(cases\ k; rule\ exec\text{-}gpv\text{-}oracle\text{-}bisim\text{-}bad[where\ X=(=)\ and\ ?bad1.0=fst\ and\ ?bad2.0=fst\ and\ \mathcal{I} = \mathcal{I}\text{-}full\ \oplus_{\mathcal{I}}\ \mathcal{I}\text{-}full])$

$(auto\ intro: rel\text{-}spmf\ refl\ callee\text{-}invariant\text{-}extend\text{-}state\text{-}oracle\text{-}const'\ simp\ add: spmf\text{-}rel\text{-}map1\ spmf\text{-}rel\text{-}map2\ oracle\text{-}decrypt0'\text{-}sims\ oracle\text{-}decrypt1'\text{-}sims\ assms\ split: plus\text{-}oracle\text{-}split)$

— We cannot get rid of the losslessness assumption on  $\mathcal{A}$  in this step, because if it were not, then the bad event might still occur, but the adversary does not terminate in the case of  $game01'\ oracle\text{-}decrypt1'$ . 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**  $|measure\ (measure\text{-}spmf\ (?game1'\ \mathcal{A}))\ \{(b, bad). b\} - measure\ (measure\text{-}spmf\ (?game0'\ \mathcal{A}))\ \{(b, bad). b\}|$

$\leq measure\ (measure\text{-}spmf\ (?game1'\ \mathcal{A}))\ \{(b, bad). bad\}$

**by**  $(rule\ fundamental\text{-}lemma[where\ ?bad2.0=snd])(auto\ intro!: rel\text{-}spmf\ bind\text{-}refl)$

```

rel-spmf-bindI[OF *] simp add: game01'-def
also have ... = spmf (map-spmf snd (?game1'  $\mathcal{A}$ )) True
by (simp add: spmf-conv-measure-spmf measure-map-spmf split-def vimage-def)
also have map-spmf snd (?game1'  $\mathcal{A}$ ) = UPF.game (reduction-upf  $\mathcal{A}$ )
proof -
note [split del] = if-split
have map-spmf ( $\lambda x. \text{fst} (\text{snd } x)$ ) (exec-gpv ( $\dagger(\text{ind-cca.oracle-encrypt} (k\text{-prf}, k\text{-upf}) b)$ 
 $\oplus_{\mathcal{O}}$  oracle-decrypt1' (k-prf, k-upf))  $\mathcal{A}$  (False, { $\}$ )) =
  map-spmf ( $\lambda x. \text{fst} (\text{snd } x)$ ) (exec-gpv (UPF.oracle k-upf) (inline (intercept-upf k-prf
  b)  $\mathcal{A}$  ({ $\}$ , { $\}$ )) (False, { $\}$ ))
  (is map-spmf ?fl ?lhs = map-spmf ?fr ?rhs is map-spmf - (exec-gpv ?oracle-normal -
  ?init-normal) = -)
for k-prf k-upf b
proof(rule map-spmf-eq-map-spmfI)
define oracle-intercept
where [simp]: oracle-intercept = ( $\lambda (s', s) y. \text{map-spmf} (\lambda ((x, s'), s). (x, s', s))$ 
  (exec-gpv (UPF.oracle k-upf) (intercept-upf k-prf b s' y) s))
let ?I = ( $\lambda ((L, D), (\text{flg}, \text{Li})).$ 
  ( $\forall (x, c, t) \in L. \text{upf-fun } k\text{-upf} (x @ c) = t \wedge \text{length } x = \text{prf-dlen}$ )  $\wedge$ 
  ( $\forall e \in \text{Li}. \exists (x, c, -) \in L. e = x @ c$ )  $\wedge$ 
  ( $(\exists (x, c, t) \in D. \text{upf-fun } k\text{-upf} (x @ c) = t) \longleftrightarrow \text{flg}$ ))
interpret callee-invariant-on oracle-intercept ?I  $\mathcal{I}$ -full
apply(unfold-locales)
subgoal for s x y s'
apply(cases s; cases s'; cases x)
apply(clarsimp simp add: set-spmf-of-set-finite[OF prf-domain-finite]
  UPF.oracle-hash-def prf-domain-length exec-gpv-bind Let-def split: bool.splits)
apply(force simp add: exec-gpv-bind UPF.oracle-flag-def split: if-split-asm)
done
subgoal by simp
done

define S :: bool  $\times$  'hash cipher-text set  $\Rightarrow$  ('hash cipher-text set  $\times$  'hash cipher-text
set)  $\times$  bool  $\times$  bitstring set  $\Rightarrow$  bool
where S = ( $\lambda (\text{bad}, L1) ((L2, D), -). \text{bad} = (\exists (x, c, t) \in D. \text{upf-fun } k\text{-upf} (x @ c) =$ 
t)  $\wedge L1 = L2$ )  $\uparrow$  ( $\lambda -. \text{True}$ )  $\otimes$  ?I
define initial :: ('hash cipher-text set  $\times$  'hash cipher-text set)  $\times$  bool  $\times$  bitstring set
where initial = (({ $\}$ , { $\}$ ), (False, { $\}$ ))
have [transfer-rule]: S ?init-normal initial by(simp add: S-def initial-def)
have [transfer-rule]: (S  $\text{====}$   $\Rightarrow$  (=)  $\text{====}$   $\Rightarrow$  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-refl
  rel-pmf-bind-refl)
have rel-spmf (rel-prod (=) S) ?lhs (exec-gpv oracle-intercept  $\mathcal{A}$  initial)

```

```

    by(transfer-prover)
  then show rel-spmf (λx 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 = (λkey 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  $\mathcal{A} \equiv$  do {

```

```

key ← key-gen;
b ← coin-spmf;
(b', D) ← exec-gpv
  (oracle-encrypt2 key b ⊕O oracle-decrypt2 key)  $\mathcal{A}$  Map-empty;
return-spmf (b = b')
}

```

**fun** *intercept-prf* ::

```

'upf-key ⇒ bool ⇒ unit ⇒ (bitstring × bitstring) + 'hash cipher-text
⇒ (('hash cipher-text option + bitstring option) × unit, bitstring, bitstring) gpv

```

**where**

```

intercept-prf - - (Inr -) = Done (Inr None, ())
| intercept-prf k b - (Inl (m1, m0)) = (case (length m1) = prf-clen ∧ (length m0) = prf-clen
of
  False ⇒ Done (Inl None, ())
| True ⇒ do {
  x ← lift-spmf (spmf-of-set prf-domain);
  p ← Pause x Done;
  let c = p [⊕] (if b then m1 else m0);
  let t = upf-fun k (x @ c);
  Done (Inl (Some (x, c, t)), ())
})

```

**definition** *reduction-prf*

```

:: (bitstring, 'hash cipher-text) ind-cca.adversary ⇒ (bitstring, bitstring) PRF.adversary

```

**where**

```

reduction-prf  $\mathcal{A}$  = do {
  k ← lift-spmf upf-key-gen;
  b ← lift-spmf coin-spmf;
  (b', -) ← inline (intercept-prf k b)  $\mathcal{A}$  ();
  Done (b' = b)
}

```

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

**proof** –

**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) ← exec-gpv (oracle-encrypt1'' key b ⊕O oracle-decrypt2 key)  $\mathcal{A}$  ();
return-spmf (b = b')

have ind-cca'.game  $\mathcal{A}$  = game1''
proof –
  define S where S = ( $\lambda$ (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-refl rel-pmf-bind-refl 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  $\mathcal{A}$ )
proof –
  { fix k-prf k-upf b
    define oracle-normal
    where oracle-normal = oracle-encrypt1'' (k-prf, k-upf) b ⊕O oracle-decrypt2 (k-prf,
k-upf)
    define oracle-intercept
    where oracle-intercept = ( $\lambda$ (s', s :: unit) y. map-spmf ( $\lambda$ ((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 = ( $\lambda$ (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-refl rel-spmf-refl)
    have map-spmf ( $\lambda$ x. b = fst x) (exec-gpv oracle-normal  $\mathcal{A}$  initial) =
map-spmf ( $\lambda$ x. b = fst (fst x)) (exec-gpv (PRF.prf-oracle k-prf) (inline (intercept-prf
k-upf b)  $\mathcal{A}$  ()))
    by(transfer fixing: b  $\mathcal{A}$  prf-fun k-prf prf-domain prf-clen upf-fun k-upf)
    (auto simp add: map-spmf-eq-map-spmf-iff exec-gpv-inline spmf-rel-map ora-
cle-intercept-def split-def intro: rel-spmf-refl) }
  then show ?thesis unfolding game1''-def PRF.game-0-def key-gen-def reduction-prf-def
by (auto simp add: exec-gpv-bind-lift-spmf exec-gpv-bind map-spmf-conv-bind-spmf
split-def eq-commute intro!: bind-spmf-cong[OF refl])
qed

```

```

also have game2  $\mathcal{A} = \text{PRF.game-1 (reduction-prf } \mathcal{A})$ 
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 = ( $\lambda(s', s) y. \text{map-spmf } (\lambda((x, s'), s). (x, s', s)) (\text{exec-gpv } \text{PRF.random-oracle } (\text{intercept-prf } k\text{-upf } b \ s' \ y) \ s))$ )
    define S
      where S = ( $\lambda(s2 :: \text{unit}, s2') (s1 :: (\text{bitstring}, \text{bitstring}) \ \text{PRF.dict}). s2' = s1$ )

    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-refl rel-spmf-refl)

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

```

```

definition oracle-encrypt3 ::
  ('prf-key  $\times$  'upf-key)  $\Rightarrow$  bool  $\Rightarrow$  (bool  $\times$  (bitstring, bitstring) PRF.dict)  $\Rightarrow$ 
  bitstring  $\times$  bitstring  $\Rightarrow$  ('hash cipher-text option  $\times$  (bool  $\times$  (bitstring, bitstring)
  PRF.dict)) spmf
where
  oracle-encrypt3 = ( $\lambda(k\text{-prf}, k\text{-upf}) b (\text{bad}, D) (\text{msg1}, \text{msg0}).$ 
    (case (length msg1 = prf-clen  $\wedge$  length msg0 = prf-clen) of
      False  $\Rightarrow$  return-spmf (None, (bad, D))
      | True  $\Rightarrow$  do {
        x  $\leftarrow$  spmf-of-set prf-domain;
        P  $\leftarrow$  spmf-of-set (nlists UNIV prf-clen);
        let (p, F) = (case D x of Some r  $\Rightarrow$  (P, True) | None  $\Rightarrow$  (P, False));
        let c = p [ $\oplus$ ] (if b then msg1 else msg0);
        let t = upf-fun k-upf (x @ c);

```



return-spmf (Some (x, c, t), (bad  $\vee$  F, D(x  $\mapsto$  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  $\Rightarrow$  (bool  $\times$  bool) spmf

**where**

*game3*  $\mathcal{A} \equiv$  do {  
 key  $\leftarrow$  key-gen;  
 b  $\leftarrow$  coin-spmf;  
 (b', (bad, D))  $\leftarrow$  exec-gpv (oracle-encrypt3 key b  $\oplus_O$  oracle-decrypt2 key)  $\mathcal{A}$  (False,  
 Map-empty);  
 return-spmf (b = b', bad)  
 }

**lemma** *round-3*:

**assumes** *lossless-gpv* ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full)  $\mathcal{A}$

**shows** |measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). b} - spmf (game2  $\mathcal{A}$ ) True|  
 $\leq$  measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). bad}

**proof** -

**define** *oracle-encrypt2'*

**where** *oracle-encrypt2'* = ( $\lambda$ (k-prf :: 'prf-key, k-upf) b (bad, D) (msg1, msg0).

case length msg1 = prf-clen  $\wedge$  length msg0 = prf-clen of

False  $\Rightarrow$  return-spmf (None, (bad, D))

| True  $\Rightarrow$  do {

x  $\leftarrow$  spmf-of-set prf-domain;

P  $\leftarrow$  spmf-of-set (nlists UNIV prf-clen);

let (p, F) = (case D x of Some r  $\Rightarrow$  (r, True) | None  $\Rightarrow$  (P, False));

let c = p [ $\oplus$ ] (if b then msg1 else msg0);

let t = upf-fun k-upf (x @ c);

return-spmf (Some (x, c, t), (bad  $\vee$  F, D(x  $\mapsto$  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'* = ( $\lambda$  $\mathcal{A}$ . do {

```

    key ← key-gen;
    b ← coin-spmf;
    (b', (bad, D)) ← exec-gpv (oracle-encrypt2' key b ⊕O oracle-decrypt2 key)  $\mathcal{A}$  (False,
Map-empty);
    return-spmf (b = b', bad))

have game2'-eq: game2  $\mathcal{A}$  = map-spmf fst (game2'  $\mathcal{A}$ )
proof –
  define S where S = ( $\lambda$ (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-refl 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 ( $\lambda$ (b'1, bad1, L1) (b'2, bad2, L2). (bad1  $\longleftrightarrow$  bad2)  $\wedge$  ( $\neg$ 
bad2  $\longrightarrow$  b'1  $\longleftrightarrow$  b'2))
    (exec-gpv (oracle-encrypt3 key b ⊕O oracle-decrypt2 key)  $\mathcal{A}$  (False, Map-empty))
    (exec-gpv (oracle-encrypt2' key b ⊕O oracle-decrypt2 key)  $\mathcal{A}$  (False, Map-empty))
  for key b
  apply(rule exec-gpv-oracle-bisim-bad[where X=(=) and X-bad =  $\lambda$ - . True and
?bad1.0=fst and ?bad2.0=fst and  $\mathcal{I} = \mathcal{I}$ -full ⊕ $\mathcal{I}$   $\mathcal{I}$ -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-refl rel-spmf-refl)
  done
  have |measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). b} – measure (measure-spmf
(game2'  $\mathcal{A}$ )) {(b, bad). b}| ≤
    measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). bad}
  unfolding game2'-def game3-def
  by(rule fundamental-lemma[where ?bad2.0=snd])(intro rel-spmf-bind-refl rel-spmf-bindI[OF
*]; clarsimp)
  ultimately show ?thesis by(simp add: spmf-conv-measure-spmf measure-map-spmf vim-
age-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 = ( $\lambda(k\text{-prf} :: \text{'prf-key}, k\text{-upf}) (s :: \text{unit}) (msg1 :: \text{bitstring}, msg0 :: \text{bitstring}).$ 
    case length msg1 = prf-clen  $\wedge$  length msg0 = prf-clen of
      False  $\Rightarrow$  return-spmf (None, s)
    | True  $\Rightarrow$  do {
      x  $\leftarrow$  spmf-of-set prf-domain;
      P  $\leftarrow$  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  $\leftarrow$  key-gen;
  (b', -)  $\leftarrow$  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 = ( $\lambda(- :: \text{unit}) (- :: \text{bool} \times (\text{bitstring}, \text{bitstring}) \text{PRF.dict}). \text{True}$ )
  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))
    ( $\lambda key b. \text{oracle-encrypt4 } key \text{ oracle-encrypt3}$ )
  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  $\sqsupset = - \text{one-time-pad[symmetric]}$ , 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**  $\dots = \text{coin-spmf}$

**by**(*simp add: map-eq-const-coin-spmf game4-def bind-spmf-const split-def lossless-exec-gpv[OF assms] lossless-weight-spmfD*)

**finally show** *?thesis* .

**qed**

**lemma** *game3-bad*:

**assumes** *interaction-bounded-by isl  $\mathcal{A}$  q*

**shows**  $\text{measure} (\text{measure-spmf} (\text{game3 } \mathcal{A})) \{(b, \text{bad}). \text{bad}\} \leq q / \text{card } \text{prf-domain} * q$

**proof** –

**have**  $\text{measure} (\text{measure-spmf} (\text{game3 } \mathcal{A})) \{(b, \text{bad}). \text{bad}\} = \text{spmf} (\text{map-spmf } \text{snd} (\text{game3 } \mathcal{A})) \text{True}$

**by** (*simp add: spmf-conv-measure-spmf measure-map-spmf vimage-def snd-def*)

**also**

**have**  $\text{spmf} (\text{map-spmf} (\text{fst} \circ \text{snd}) (\text{exec-gpv} (\text{oracle-encrypt3 } k b \oplus_O \text{oracle-decrypt2 } k) \mathcal{A} (\text{False}, \text{Map.empty}))) \text{True} \leq q / \text{card } \text{prf-domain} * q$

(*is spmf (map-spmf - (exec-gpv ?oracle - -)) - ≤ -*)

**if**  $k: k \in \text{set-spmf } \text{key-gen}$  **for**  $k b$

**proof**(*rule callee-invariant-on.interaction-bounded-by'-exec-gpv-bad-count*)

**obtain**  $k\text{-prf } k\text{-upf}$  **where**  $k: k = (k\text{-prf}, k\text{-upf})$  **by**(*cases k*)

**let**  $?I = \lambda(\text{bad}, D). \text{finite} (\text{dom } D) \wedge \text{dom } D \subseteq \text{prf-domain}$

**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**  $?count = \lambda(\text{bad}, D). \text{card} (\text{dom } D)$

**show**  $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (?oracle s x); ?I s; \text{isl } x \rrbracket \implies ?count s' \leq \text{Suc} (?count s)$

**by**(*clarsimp simp add: isl-def oracle-encrypt3-def split-def Let-def card-insert-if split: bool.splits*)

**show**  $\llbracket (y, s') \in \text{set-spmf} (?oracle s x); ?I s; \neg \text{isl } x \rrbracket \implies ?count s' \leq ?count s$  **for**  $s x y s'$

**by**(*cases x*)(*simp-all add: oracle-decrypt2-def*)

**show**  $\text{spmf} (\text{map-spmf} (\text{fst} \circ \text{snd}) (?oracle s' x)) \text{True} \leq q / \text{card } \text{prf-domain}$

**if**  $?I s'$  **and**  $\text{bad}: \neg \text{fst } s'$  **and**  $\text{count}: ?count s' < q + ?count (\text{False}, \text{Map.empty})$

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

**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**  $(auto\ elim:\ isE)$   
**have**  $*$ :  $(case\ D\ x\ of\ None \Rightarrow False \mid Some\ x \Rightarrow True) \longleftrightarrow x \in dom\ D$  **for**  $x$   
**by**  $(auto\ split:\ option.split)$   
**show**  $?thesis$   
**proof**  $(cases\ length\ m1 = prf-clen \wedge length\ m0 = prf-clen)$   
**case**  $True$   
**with**  $bad$   
**have**  $spmf\ (map\ spmf\ (fst \circ snd)\ (?oracle\ s'\ x))\ True = pmf\ (bernoulli\ pmf\ (card\ (dom\ D \cap prf\ domain) / card\ prf\ domain))\ True$   
**by**  $(simp\ add:\ spmf.map\ comp\ o\ def\ oracle\ encrypt3\ def\ k * bool.case\ distrib [where\ h = \lambda p.\ spmf\ (map\ spmf\ -\ p)\ -] option.case\ distrib [where\ h = snd] map\ spmf\ bind\ spmf\ Let\ def\ split\ beta\ bind\ spmf\ const\ cong:\ bool.case\ cong\ option.case\ cong\ split\ del:\ if\ split\ split:\ bool.split)$   
 $(simp\ add:\ map\ spmf\ conv\ bind\ spmf [symmetric] map\ mem\ spmf\ of\ set\ prf\ domain\ finite\ prf\ domain\ nonempty)$   
**also** **have**  $\dots = card\ (dom\ D \cap prf\ domain) / card\ prf\ domain$   
**by**  $(rule\ pmf\ bernoulli\ True) (auto\ simp\ add:\ field\_simps\ prf\ domain\ finite\ prf\ domain\ nonempty\ card\ gt\ 0\ iff\ card\ mono)$   
**also** **have**  $dom\ D \cap prf\ domain = dom\ D$  **using**  $I$  **by**  $auto$   
**also** **have**  $card\ (dom\ D) \leq q$  **using**  $count$  **by**  $simp$   
**finally** **show**  $?thesis$  **by**  $(simp\ add:\ divide\ right\ mono\ o\ def)$   
**next**  
**case**  $False$   
**thus**  $?thesis$  **using**  $bad$   
**by**  $(auto\ simp\ add:\ spmf.map\ comp\ o\ def\ oracle\ encrypt3\ def\ k\ split:\ bool.split)$   
**qed**  
**qed**  
**qed**  $(auto\ split:\ plus\ oracle\ split\ asm\ simp\ add:\ oracle\ decrypt2\ def\ assms)$   
**then** **have**  $spmf\ (map\ spmf\ snd\ (game3\ \mathcal{A}))\ True \leq q / card\ prf\ domain * q$   
**by**  $(auto\ 4\ 3\ simp\ add:\ game3\ def\ map\ spmf\ bind\ spmf\ o\ def\ split\ def\ map\ spmf\ conv\ bind\ spmf\ intro:\ spmf\ bind\ leI)$   
**finally** **show**  $?thesis$  .  
**qed**

**theorem**  $security$ :

**assumes**  $lossless$ :  $lossless\ gpv\ (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})\ \mathcal{A}$   
**and**  $bound$ :  $interaction\ bounded\ by\ isl\ \mathcal{A}\ q$   
**shows**  $ind\ cca.\ advantage\ \mathcal{A} \leq$   
 $PRF.\ advantage\ (reduction\ prf\ \mathcal{A}) + UPF.\ advantage\ (reduction\ upf\ \mathcal{A}) +$   
 $real\ q / real\ (card\ prf\ domain) * real\ q$  **(is**  $?LHS \leq -)$   
**proof** –  
**have**  $?LHS \leq |spmf\ (ind\ cca.\ game\ \mathcal{A})\ True - spmf\ (ind\ cca'.\ game\ \mathcal{A})\ True| + |spmf\ (ind\ cca'.\ game\ \mathcal{A})\ True - 1 / 2|$   
**(is**  $- \leq ?round1 + ?rest)$  **using**  $abs\ triangle\ ineq$  **by**  $(simp\ add:\ ind\ cca.\ advantage\ def)$   
**also** **have**  $?round1 \leq UPF.\ advantage\ (reduction\ upf\ \mathcal{A})$   
**using**  $lossless$  **by**  $(rule\ round1)$   
**also** **have**  $?rest \leq |spmf\ (ind\ cca'.\ game\ \mathcal{A})\ True - spmf\ (game2\ \mathcal{A})\ True| + |spmf$

$(\text{game2 } \mathcal{A}) \text{ True} - 1 / 2|$   
**(is -  $\leq$  ?round2 + ?rest) using abs-triangle-ineq by simp**  
**also have ?round2 = PRF.advantage (reduction-prf  $\mathcal{A}$ ) by (rule round-2)**  
**also have ?rest  $\leq$  |measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). b} - spmf (game2  $\mathcal{A}$ ) True| +**  
 $| \text{measure (measure-spmf (game3 } \mathcal{A}) ) \{(b, \text{bad}). b\} - 1 / 2|$   
**(is -  $\leq$  ?round3 + -) using abs-triangle-ineq by simp**  
**also have ?round3  $\leq$  measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). bad}**  
**using round-3[OF lossless].**  
**also have ...  $\leq$  q / card prf-domain \* q using bound by (rule game3-bad)**  
**also have measure (measure-spmf (game3  $\mathcal{A}$ )) {(b, bad). b} = spmf coin-spmf True**  
**using round-4[OF lossless, symmetric]**  
**by (simp add: spmf-conv-measure-spmf measure-map-spmf vimage-def fst-def)**  
**also have |... - 1 / 2| = 0 by (simp add: spmf-of-set)**  
**finally show ?thesis by (simp)**  
**qed**

**theorem security1:**

**assumes lossless: lossless-gpv ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full)  $\mathcal{A}$**   
**assumes q: interaction-bounded-by isl  $\mathcal{A}$  q**  
**and q': interaction-bounded-by (Not  $\circ$  isl)  $\mathcal{A}$  q'**  
**shows ind-cca.advantage  $\mathcal{A}$   $\leq$**   
 $\text{PRF.advantage (reduction-prf } \mathcal{A}) +$   
 $\text{UPF.advantage1 (guessing-many-one.reduction } q' (\lambda \cdot \text{. reduction-upf } \mathcal{A}) ()) * q' +$   
 $\text{real } q * \text{real } q / \text{real (card prf-domain)}$   
**proof -**  
**have ind-cca.advantage  $\mathcal{A}$   $\leq$**   
 $\text{PRF.advantage (reduction-prf } \mathcal{A}) + \text{UPF.advantage (reduction-upf } \mathcal{A}) +$   
 $\text{real } q / \text{real (card prf-domain)} * \text{real } q$   
**using lossless q by (rule security)**  
**also note q'[interaction-bound]**  
**have interaction-bounded-by (Not  $\circ$  isl) (reduction-upf  $\mathcal{A}$ ) q'**  
**unfolding reduction-upf-def by (interaction-bound) (simp-all add: SUP-le-iff)**  
**then have UPF.advantage (reduction-upf  $\mathcal{A}$ )  $\leq$  UPF.advantage1 (guessing-many-one.reduction**  
 $q' (\lambda \cdot \text{. reduction-upf } \mathcal{A}) ()) * q'$   
**by (rule UPF.advantage-advantage1)**  
**finally show ?thesis by (simp)**  
**qed**

**end**

**end**

**locale simple-cipher' =**

**fixes prf-key-gen :: security  $\Rightarrow$  'prf-key spmf**  
**and prf-fun :: security  $\Rightarrow$  'prf-key  $\Rightarrow$  bitstring  $\Rightarrow$  bitstring**  
**and prf-domain :: security  $\Rightarrow$  bitstring set**  
**and prf-range :: security  $\Rightarrow$  bitstring set**  
**and prf-dlen :: security  $\Rightarrow$  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.advantageI η (guessing-many-one.reduction (q' η) (λ-. reduc-
tion-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.advantageI 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: securityI[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].<sup>1</sup> 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.

---

<sup>1</sup>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](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](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.<sup>2</sup> We return to asymptotic security statements in §6.

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

---

<sup>2</sup>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  $\mathcal{G}$  be a group of order  $q$  with generator  $\mathbf{g}$ . The List Computational Diffie-Hellman (LCDH) assumption holds for  $\mathcal{G}$  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  $\{0, \dots, q - 1\}$ .
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  $\mathcal{G}$  using Isabelle’s module system based on locales. The locale *list-cdh* fixes  $\mathcal{G}$  to be a finite cyclic group that has elements of type *'grp* and comes with a generator  $\mathbf{g}_{\mathcal{G}}$ . Basic facts about finite groups are formalized in the CryptHOL theory *CryptHOL.Cyclic-Group*.<sup>3</sup>

```
locale list-cdh = cyclic-group  $\mathcal{G}$ 
for  $\mathcal{G} :: 'grp$  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

---

<sup>3</sup>The syntax directive **structure** tells Isabelle that all group operations in the context of the locale refer to the group  $\mathcal{G}$  unless stated otherwise. For example,  $\mathbf{g}_{\mathcal{G}}$  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  $\mathcal{G}$  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.<sup>4</sup>

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

The LCDH game itself is expressed as a function from the adversary  $\mathcal{A}$  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  $\{0, \dots, n - 1\}$ .
- *order*  $\mathcal{G}$  denotes the order of  $\mathcal{G}$  and  $([\wedge]) :: 'grp \Rightarrow nat \Rightarrow 'grp$  is the group exponentiation operator.

The LCDH game formalizes the challenger’s behavior against an adversary  $\mathcal{A}$ . In the following definition, the challenger randomly (and independently) picks two natural numbers  $x$  and  $y$  that are between 0 and  $\mathcal{G}$ ’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  $\mathcal{G}$ . The game finally returns a *boolean* that indicates whether the adversary produced a right guess. Formally, *game*  $\mathcal{A}$  is a *boolean* random variable.

**definition** *game* :: *'grp adversary ⇒ bool spmf* **where**

```

game  $\mathcal{A}$  = do {
   $x \leftarrow$  sample-uniform (order  $\mathcal{G}$ );
   $y \leftarrow$  sample-uniform (order  $\mathcal{G}$ );
   $zs \leftarrow$   $\mathcal{A}$  ( $\mathbf{g} [\wedge] x$ ) ( $\mathbf{g} [\wedge] y$ );
  return-spmf ( $\mathbf{g} [\wedge] (x * y) \in zs$ )
}
```

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*  $\mathcal{A}$  = *spmf* (*game*  $\mathcal{A}$ ) *True*

---

<sup>4</sup>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 parameters. 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  $( 's, 'a, 'b ) \textit{oracle}$  abbreviates  $'s \Rightarrow 'a \Rightarrow ('b \times 's) \textit{spmf}$ .

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*  $\rightarrow$  *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*  $\rightarrow$  *bitstring*

**definition** *oracle* :: *'a state*  $\Rightarrow$  *'a*  $\Rightarrow$  (*bitstring*  $\times$  *'a state*) *spmf*

**where**

*oracle*  $\sigma$  *x* = (case  $\sigma$  *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*,  $\sigma$ ))

Initially, the state of a random oracle is the empty map *empty*, 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-of-set* (*nlists UNIV len*);  
 *return-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 *spmf*s. 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  $key\text{-}gen$  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 plain-texts. (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'*  $\Rightarrow$  ((*'plain'*  $\times$  *'plain'*)  $\times$  *'state', 'query', 'response'*) *gpv*)  
 $\times$  (*'cipher'*  $\Rightarrow$  *'state'*  $\Rightarrow$  (*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*  
 $\Rightarrow$  (*bool, 'query', 'response'*) *gpv*

**where**

*game* ( $\mathcal{A}_1, \mathcal{A}_2$ ) = *TRY* *do* {  
 ( $pk, sk$ )  $\leftarrow$  *key-gen*;  
 ( $(m_0, m_1), \sigma$ )  $\leftarrow$   $\mathcal{A}_1\ pk$ ;  
*assert-gpv* (*valid-plains*  $m_0\ m_1$ );  
 $b \leftarrow$  *lift-spmf coin-spmf*;

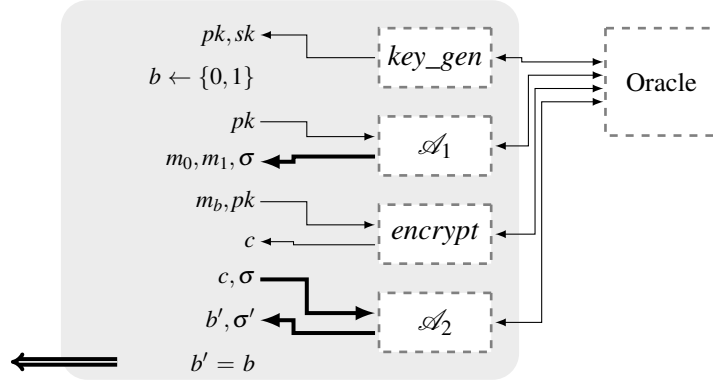


Figure 1: Graphic representation of the generic IND-CPA game.

```

cipher ← encrypt pk (if b then m0 else m1);
b' ← A2 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  $\sigma :: 's$  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) A = |spmf (run-gpv oracle (game 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  $\mathcal{G}$  +
  ro: random-oracle len-plain
  for  $\mathcal{G}$  :: '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  $(\alpha, \beta)$ , 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.<sup>5</sup> For readability, we introduce meaningful abbreviations for these concepts.

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

<sup>5</sup>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* =  $'grp' \times \textit{bitstring}$

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**  $\textit{hash} :: 'grp \Rightarrow (\textit{bitstring}, 'grp, \textit{bitstring}) \textit{gpv}$   
**where**  
 $\textit{hash } x \equiv \textit{Pause } x \textit{ Done}$

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

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

**definition**  $\textit{decrypt} :: 'grp \textit{priv-key} \Rightarrow 'grp \textit{cipher} \Rightarrow (\textit{plain}, 'grp, \textit{bitstring}) \textit{gpv}$   
**where**  
 $\textit{decrypt } x = (\lambda (\beta, \zeta). \textit{do} \{$   
 $\quad h \leftarrow \textit{hash} (\beta [\wedge] x);$   
 $\quad \textit{Done} (\zeta [\oplus] h)$   
 $\})$

**definition**  $\textit{valid-plains} :: \textit{plain} \Rightarrow \textit{plain} \Rightarrow \textit{bool}$   
**where**  
 $\textit{valid-plains } \textit{msg1 } \textit{msg2} \longleftrightarrow \textit{length } \textit{msg1} = \textit{len-plain} \wedge \textit{length } \textit{msg2} = \textit{len-plain}$

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-

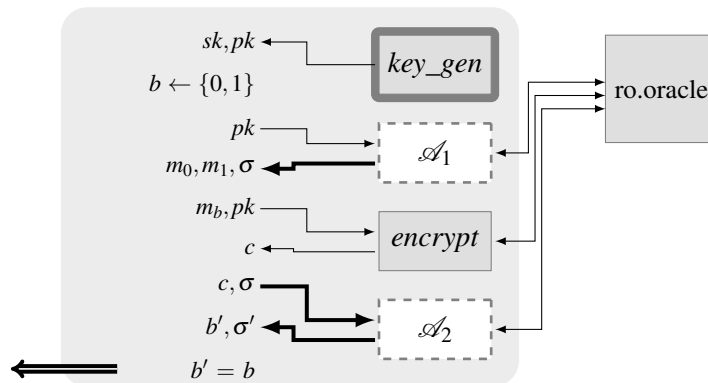


Figure 2: The IND-CPA game instantiated with the Hashed-ElGamal encryption scheme and accessing a random oracle.

gama 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-elgama`'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 Elgama 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 Elgama that depends on another adversary's advantage in the LCDH game `game` of the

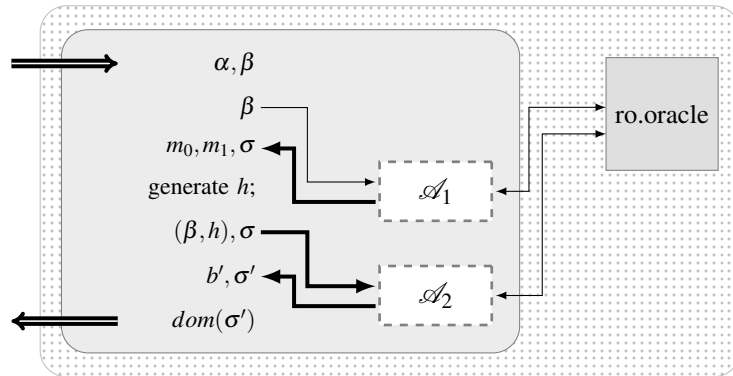


Figure 3: The reduction for the Elgamal security proof.

underlying group. The reduction transforms every IND-CPA adversary  $\mathcal{A}$  into a LCDH adversary *elgamal-reduction*  $\mathcal{A}$ , using  $\mathcal{A}$  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  $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$  (Figure 3 visualizes the reduction as the dotted box):

1. It receives two group elements  $\alpha$  and  $\beta$  from the LCDH challenger.
2. The reduction passes  $\alpha$  to the adversary as the public key and runs  $\mathcal{A}_1$  to get messages  $m_1$  and  $m_2$ . The adversary is given access to the random oracle with the initial state *empty*.
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  $(\beta, h)$  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 ( $\mathcal{A}_1, \mathcal{A}_2$ )  $\alpha \beta = do$  {
    (( $m_1, m_2$ ),  $\sigma$ ),  $s$  ← exec-gpv ro.oracle ( $\mathcal{A}_1 \alpha$ ) 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 ( $\mathcal{A}_2 (\beta, h) \sigma$ )  $s$ ;
      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  $\mathcal{A}$ , the advantage in one game is bounded by a function of the transformed adversary's advantage in a different game.<sup>6</sup>

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

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*  $\mathcal{A}$   
**shows**  $ind-cpa.advantage (ro.oracle, ro.initial) \mathcal{A} \leq lcdh.advantage (elgamal-reduction \mathcal{A})$

Such a statement captures the essence of a concrete security proof. For if there was a feasible adversary  $\mathcal{A}$  with non-negligible advantage against the *game*, then *elgamal-reduction*  $\mathcal{A}$  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*  $\mathcal{A}$  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.

---

<sup>6</sup>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**  
 $lossless = (\lambda (\mathcal{A}_1, \mathcal{A}_2). (\forall pk. lossless-gpv \mathcal{I}\text{-full} (\mathcal{A}_1 pk))$   
 $\wedge (\forall cipher \sigma. lossless-gpv \mathcal{I}\text{-full} (\mathcal{A}_2 cipher \sigma)))$

So now let's start with the proof.

**proof** –

As a preparatory step, we split the adversary  $\mathcal{A}$  into its two phases  $\mathcal{A}_1$  and  $\mathcal{A}_2$ . 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  $\mathcal{A}$ .<sup>7</sup>

**obtain**  $\mathcal{A}_1 \mathcal{A}_2$  **where**  $\mathcal{A}$  [simp]:  $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$  **by** (cases  $\mathcal{A}$ )  
**from** *lossless* **have** *lossless1* [simp]:  $\wedge pk. lossless-gpv \mathcal{I}\text{-full} (\mathcal{A}_1 pk)$   
**and** *lossless2* [simp]:  $\wedge \sigma cipher. lossless-gpv \mathcal{I}\text{-full} (\mathcal{A}_2 \sigma cipher)$   
**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**  
 $interceptor \sigma x = (do \{$   
 $h \leftarrow hash x;$   
 $Done (h, insert x \sigma)$   
 $\})$  **for**  $\sigma x$

We integrate this interceptor into the *game* using the *inline* function as illustrated in Fig. 4 and name the result *game<sub>0</sub>*.

**define** *game<sub>0</sub>* **where**

<sup>7</sup>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.



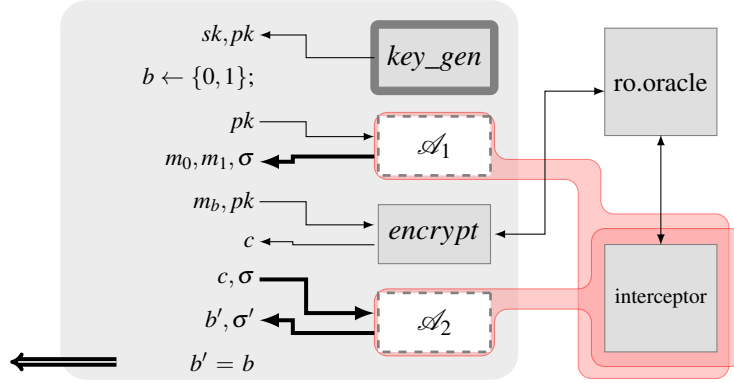


Figure 4: The IND-CPA game after expanding the key generation algorithm’s definition and inlining the query-recording hash oracle. The red boxes represent the inline operator.

```

game0 = TRY do {
  (pk, -) ← lift-spmf key-gen;
  ((m1, m2), σ), s ← inline interceptor (A1 pk) {};
  assert-gpv (valid-plains m1 m2);
  b ← lift-spmf coin-spmf;
  c ← encrypt pk (if b then m1 else m2);
  (b', s') ← inline interceptor (A2 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*  $\mathcal{A}$  and *game*<sub>0</sub> have identical structure. They differ only in that *game*<sub>0</sub> uses the adversary  $(\lambda pk. inline\ interceptor\ (\mathcal{A}_1\ pk)\ \emptyset, \lambda cipher\ \sigma. inline\ interceptor\ (\mathcal{A}_2\ cipher\ \sigma))$  instead of  $\mathcal{A}$ . 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  $\lambda\sigma. \sigma$  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) (λσ. 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 ℳ = game0 unfolding game0-def ℳ 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<sub>0</sub>* 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, σ(x ↦ bs)) }
| Some bs ⇒ return-spmf (bs, σ);
return-spmf (h, insert x s, σ')
})
have *: exec-gpv ro.oracle (inline interceptor  $\mathcal{A}$  s) σ =
map-spmf (λ(a, b, c). ((a, b), c)) (exec-gpv oracle'  $\mathcal{A}$  (s, σ)) for  $\mathcal{A}$  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].<sup>8</sup>

```

have game0: run-gpv ro.oracle game0 ro.initial = do {
  x ← sample-uniform (order  $\mathcal{G}$ );
  y ← sample-uniform (order  $\mathcal{G}$ );
  b ← coin-spmf;
  ((msg1, msg2), σ), (s, s-h) ←
  exec-gpv oracle' ( $\mathcal{A}_1$  (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' ( $\mathcal{A}_2$  cipher σ) (s, s-h');
    return-spmf (b' = b)
  } ELSE do {
    b ← coin-spmf;
    return-spmf b
  }
}
including monad-normalisation
by(simp add: game0-def key-gen-def encrypt-def * exec-gpv-bind bind-map-spmf as-
sert-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<sub>0</sub>-def*, *key-gen-def*, and *encrypt-def* should be unfolded, so they are added first to the simplifier’s set of rewrite rules.

<sup>8</sup>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.

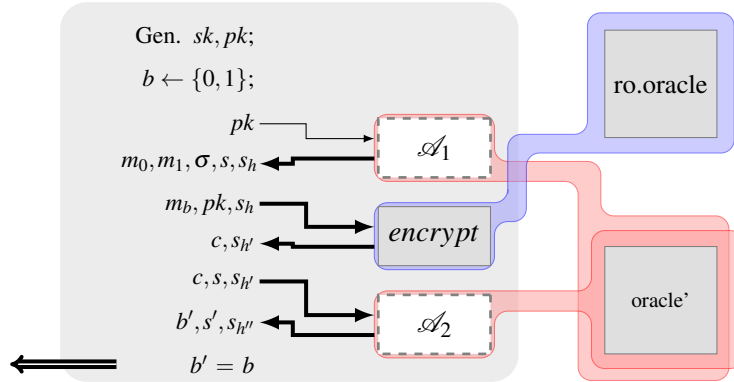


Figure 5: The IND-CPA game after flattening. The blue box around the encryption algorithm and the random oracle represents the expanded definition of them.

- The equations *exec-gpv-bind*, *try-bind-assert-gpv*, and *try-gpv-bind-lossless* ensure that the operator *exec-gpv*, which connects the *game<sub>0</sub>* 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  $(\mathbf{g} [\wedge] x) [\wedge] y$  to  $\mathbf{g} [\wedge] (x * y)$ .
- The other theorems *bind-map-spmf*, *assert-spmf-def*, *split-def*, *o-def*, and *if-distrib* 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  $\mathcal{A}_1$  and  $\mathcal{A}_2$ . 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<sub>0</sub>* 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  $\mathbf{g} [\wedge] (x * y)$  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<sub>1</sub>* 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<sub>1</sub>*.

```
define game1 :: nat ⇒ nat ⇒ (bool × bool) spmf
where game1 x y = do {
  b ← coin-spmf;
  (((m1, m2), σ), (s, s-h)) ← exec-gpv oracle' (A1 (g [∧] x)) ({}, ro.initial);
  TRY do {
    - :: unit ← assert-spmf (valid-plains m1 m2);
    (h, s-h') ← ro.oracle s-h (g [∧] (x * y));
    let c = (g [∧] y, h [⊕] (if b then m1 else m2));
    (b', (s', s-h')) ← exec-gpv oracle' (A2 c σ) (s, s-h');
    return-spmf (b' = b, g [∧] (x * y) ∈ s')
  } ELSE do {
    b ← coin-spmf;
    return-spmf (b, g [∧] (x * y) ∈ s)
  }
} for x y
```

It is easy to prove that *game<sub>0</sub>* combined with the random oracle is a projection of *game<sub>1</sub>* with the sampling added, as formalized in *game<sub>0</sub>-game<sub>1</sub>*.

```
let ?sample = λf :: nat ⇒ nat ⇒ - spmf. do {
  x ← sample-uniform (order ℒ);
  y ← sample-uniform (order ℒ);
  f x y }
have game0-game1:
  run-gpv ro.oracle game0 ro.initial = map-spmf fst (?sample game1)
by(simp add: game0 game1-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<sub>2</sub>* is identical except that

the call to the random oracle *oracle* is replaced with sampling a random bitstring.<sup>9</sup>

```

define game2 :: nat ⇒ nat ⇒ (bool × bool) spmf
where game2 x y = do {
  b ← coin-spmf;
  (((m1, m2), σ), (s, s-h)) ← exec-gpv oracle' (A1 (g [^] x)) ({} , ro.initial);
  TRY do {
    - :: unit ← assert-spmf (valid-plains m1 m2);
    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 game1 only if the adversary queries g [^] (x *
    y).
    let cipher = (g [^] y, h [⊕] (if b then m1 else m2));
    (b', (s', s-h')) ← exec-gpv oracle' (A2 cipher σ) (s, s-h);
    return-spmf (b' = b, g [^] (x * y) ∈ s')
  } ELSE do {
    b ← coin-spmf;
    return-spmf (b, g [^] (x * y) ∈ 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 (λ(win, bad) (win', bad')). bad = bad' ∧ (¬ bad' → win = win') (game2
x y) (game1 x y) for x y
proof —

```

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

```

interpret inv-oracle': callee-invariant-on oracle' (λ(s, s-h). s = dom s-h)  $\mathcal{I}$ -full
by unfold-locales(auto simp add: oracle'-def split: option.split-asm if-split)
interpret bad: callee-invariant-on oracle' (λ(s, -). z ∈ s)  $\mathcal{I}$ -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  $s = \text{dom } s-h$  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  $\mathbf{g} [^] (x * y)$ .

```

let ?X = λ(s, s-h) (s', s-h'). s = dom s-h ∧ s' = s ∧ s-h = s-h'(g [^] (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 = λ(s, s-h). g [^] (x * y) ∈ s
let ?R = (λ(a, s1') (b, s2')). ?bad s1' = ?bad s2' ∧ (¬ ?bad s2' → a = b ∧ ?X s1' s2')
have bisim: rel-spmf ?R (oracle' s1 plain) (oracle' s2 plain)

```

---

<sup>9</sup>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-refl 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<sub>1</sub>-def game<sub>2</sub>-def*  
— Peel off the first phase of the game using the structural decomposition rules *rel-spmf-bind-refl* and *rel-spmf-try-spmf*.  
**apply**(*clarsimp intro!: rel-spmf-bind-refl simp del: bind-spmf-const*)  
**apply**(*rule rel-spmf-try-spmf*)  
**subgoal** *TRY for b m<sub>1</sub> m<sub>2</sub>  $\sigma$  s s-h*  
**apply**(*rule rel-spmf-bind-refl*)  
— 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.,  $s = \text{dom } s-h$ .  
**apply**(*drule inv-oracle'.exec-gpv-invariant; clarsimp*)  
— Has the adversary queried the random oracle with  $\mathbf{g} \text{ } [\wedge] (x * y)$  during the first phase?  
**apply**(*cases  $\mathbf{g} \text{ } [\wedge] (x * y) \in s$* )  
**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  $\mathcal{S} = \mathcal{S}\text{-full}$ ] dest!: bad.exec-gpv-invariant*)  
**subgoal** *False* — Then let's see whether the adversary queries  $\mathbf{g} \text{ } [\wedge] (x * y)$  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-refl*)  
**apply**(*rule rel-spmf-bindI[where  $R = ?R$ ]*)  
— The lemma *exec-gpv-oracle-bisim-bad-full* lifts the bisimulation for *oracle'* to the adversary  $\mathcal{A}_2$  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-refl) 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<sub>2</sub>*) (*?sample game<sub>1</sub>*)  
**by**(*intro rel-spmf-bind-refl*)

**hence**  $|measure (measure\text{-}spmf (?sample\ game_2)) \{(win, -). win\} - measure (measure\text{-}spmf (?sample\ game_1)) \{(win, -). win\}|$   
 $\leq measure (measure\text{-}spmf (?sample\ game_2)) \{(-, bad). bad\}$   
**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**  $measure (measure\text{-}spmf (?sample\ game_2)) \{(win, -). win\} = spmf (map\text{-}spmf\ fst (?sample\ game_2))\ True$   
**and**  $measure (measure\text{-}spmf (?sample\ game_1)) \{(win, -). win\} = spmf (map\text{-}spmf\ fst (?sample\ game_1))\ True$   
**and**  $measure (measure\text{-}spmf (?sample\ game_2)) \{(-, bad). bad\} = spmf (map\text{-}spmf\ snd (?sample\ game_2))\ True$   
**unfolding** *spmf-conv-measure-spmf measure-map-spmf* **by** (*auto simp add: vimage-def split-def*)  
**ultimately have** *hop12*:  
 $|spmf (map\text{-}spmf\ fst (?sample\ game_2))\ True - spmf (map\text{-}spmf\ fst (?sample\ game_1))\ True|$   
 $\leq spmf (map\text{-}spmf\ snd (?sample\ game_2))\ True$   
**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*<sub>1</sub> to *game*<sub>2</sub> 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*<sub>2</sub> by the part that does the optimistic sampling and call this parametrized version *game*<sub>3</sub>.

**define**  $game_3 :: (bool \Rightarrow bitstring \Rightarrow bitstring \Rightarrow bitstring\ spmf) \Rightarrow nat \Rightarrow nat \Rightarrow (bool \times bool)\ spmf$   
**where**  $game_3\ f\ x\ y = do \{$   
 $b \leftarrow coin\text{-}spmf;$   
 $((m_1, m_2), \sigma), (s, s\text{-}h) \leftarrow exec\text{-}gpv\ oracle' (\mathcal{A}_1 (\mathbf{g} [\wedge] x)) (\{\}, ro.initial);$   
 $TRY\ do \{$   
 $- :: unit \leftarrow assert\text{-}spmf (valid\text{-}plains\ m_1\ m_2);$   
 $h' \leftarrow f\ b\ m_1\ m_2;$   
 $let\ cipher = (\mathbf{g} [\wedge] y, h');$   
 $(b', (s', s\text{-}h')) \leftarrow exec\text{-}gpv\ oracle' (\mathcal{A}_2\ cipher\ \sigma) (s, s\text{-}h);$   
 $\}$   
 $\}$



```

    return-spmf (b' = b, g [^] (x * y) ∈ s')
  } ELSE do {
    b ← coin-spmf;
    return-spmf (b, g [^] (x * y) ∈ s)
  }
} for f x y

```

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

```

let ?f = λ b m1 m2. map-spmf (λ h. (if b then m1 else m2) [⊕] h) (spmf-of-set (nlists UNIV len-plain))
have game2-game3: game2 x y = game3 ?f x y for x y
by(simp add: game2-def game3-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' = λ b m1 m2. spmf-of-set (nlists UNIV len-plain)
have game3: game3 ?f x y = game3 ?f' x y for x y
by(auto intro!: try-spmf-cong bind-spmf-cong[OF refl] if-cong[OF refl]
    simp add: game3-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*  $\mathcal{A}$ . 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 game3-bad: map-spmf snd (?sample (game3 ?f')) = lcdh.game (elgamal-reduction  $\mathcal{A}$ )
have game3-guess: map-spmf fst (game3 ?f' x y) = coin-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-cpa.advantage (ro.oracle, ro.initial)  $\mathcal{A}$  = |spmf (map-spmf fst (?sample game1))
True - 1 / 2|
using ind-cpa-game-eq-game0 by(simp add: game0-game1 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: game2-game3 game3 o-def game3-guess spmf-of-set)
also have |... - spmf (map-spmf fst (?sample game1)) True| ≤ spmf (map-spmf snd
  (?sample game2)) True
  by(rule hop12)
also have ... = lcdh.advantage (elgamal-reduction  $\mathcal{A}$ )
  by(simp add: game2-game3 game3 game3-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  $\eta$  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  $\eta$  encrypt  $\eta$  decrypt  $\eta$  valid-plains  $\eta$  for  $\eta$  .
end

```

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

```

locale hashed-elgamal' =
  lcdh: list-cdh'  $\mathcal{G}$  +

```

```

ro: random-oracle' len-plain
for  $\mathcal{G} :: security \Rightarrow 'grp\ cyclic-group$ 
and len-plain :: security  $\Rightarrow nat$ 
begin
sublocale hashed-ElGamal  $\mathcal{G} \eta$  len-plain  $\eta$  for  $\eta ..$ 

```

## 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 ( $\lambda \eta. lcdh.advantage \eta (ElGamal-reduction \eta (\mathcal{A} \eta))$ )
and eventually ( $\lambda \eta. ind-cpa.lossless (\mathcal{A} \eta)$ ) at-top
shows negligible ( $\lambda \eta. ind-cpa.advantage \eta (ro.oracle \eta, ro.initial) (\mathcal{A} \eta)$ )

```

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. Ballarín. 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](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](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](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.