Game-based cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar and Bhargav Bhatt

September 13, 2023

Abstract

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

Our proofs follow the game-hopping style advocated by Shoup [19] and Bellare and Rogaway [4], from which most of the examples have been taken. We generalise some of their results such that they can be reused in other proofs. Thanks to CryptHOL’s integration with Isabelle’s parametricity infrastructure, many simple hops are easily justified using the theory of representation independence.

Contents

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

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

definition ddh-0 :: 'grp adversary ⇒ bool spmf
where ddh-0 A = do 
x ← sample-uniform (order G);
y ← sample-uniform (order G);
A (g [^] x) (g [^] y) (g [^] (x * y))
}
definition ddh-1 :: 'grp adversary ⇒ bool spmf
where ddh-1 A = do 
x ← sample-uniform (order G);
y ← sample-uniform (order G);
z ← sample-uniform (order G);
A (g [^] x) (g [^] y) (g [^] z)
}
definition advantage :: 'grp adversary ⇒ real
where advantage A = |spmf (ddh-0 A) True − spmf (ddh-1 A) True|
definition lossless :: 'grp adversary ⇒ bool
where lossless A ←→ (∀ α β γ. lossless-spmf (A α β γ))

lemma lossless-ddh-0: 
[ [ lossless A; 0 < order G ] ]
⇒ lossless-spmf (ddh-0 A)
by(auto simp add: lossless-def ddh-0-def split-def Let-def)

lemma lossless-ddh-1: 
[ [ lossless A; 0 < order G ] ]
⇒ lossless-spmf (ddh-1 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 ← sample-uniform (order $\mathcal{G}$);
    y ← sample-uniform (order $\mathcal{G}$);
    zs ← $\mathcal{A}$ (g $\cdot^x$ x) (g $\cdot^y$ y);
    return-spmf (g $\cdot^x$ (x * y) ∈ zs)
  }

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

  definition lossless :: 'grp adversary \Rightarrow bool
  where lossless $\mathcal{A}$ ←→ (∀ α β. lossless-spmf ($\mathcal{A}$ α β))

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

end
end

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

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

1.3 The IND-CCA2 game for public-key encryption

We model an IND-CCA2 security game in the multi-user setting as described in [3].
locale ind-cca2 = pk-enc +

constrains key-gen :: security ⇒ ('ekey × 'dkey) spmf
and encrypt :: security ⇒ 'ekey ⇒ 'plain ⇒ 'cipher spmf
and decrypt :: security ⇒ 'dkey ⇒ 'cipher ⇒ 'plain option
and valid-plain :: security ⇒ 'plain ⇒ bool

begin

function encrypt-oracle
    :: security ⇒ ('ekey, 'dkey, 'cipher) state-oracle ⇒ ('ekey × 'dkey × 'cipher) list

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) -
1.3.1 Single-user setting

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

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

definition adversary1' = (ekey, plain, cipher) adversary1' =
(boos, (plain', cipher') call1, (ekey', plain', cipher') ret1) gpv
definition adversary1 = security ⇒ (ekey', plain', cipher') adversary1'

where

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

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

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

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

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

abbreviation ibounded-by1 :: (ekey, plain, cipher) adversary1 ⇒ (security ⇒ nat) ⇒
bool
where ibounded-by1 ≡ rel-envir ibounded-by1'

definition lossless1 :: (ekey, plain, cipher) adversary1' ⇒ bool
where lossless1 η = lossless-gpv η full η
abbreviation \texttt{lossless}_1 :: (\texttt{ekey}, \texttt{'plain'}, \texttt{'cipher'}) \texttt{adversary}_1 \Rightarrow \texttt{bool}
where \texttt{lossless}_1 \equiv \texttt{pred-envir} \texttt{lossless}_1^\dagger

lemma \texttt{lossless-encrypt-oracle} \texttt{[simp]}: \texttt{lossless-spmf} (\texttt{encrypt-oracle} \eta \sigma \texttt{cipher})
by (cases (\eta, \sigma, \texttt{cipher}) rule: encrypt-oracle\texttt{.cases}) \texttt{simp-all}

lemma \texttt{lossless-ekey-oracle} \texttt{[simp]}:
\texttt{lossless-spmf} (\texttt{ekey-oracle} \eta \sigma \texttt{cipher}) \leftrightarrow (\sigma = \texttt{None} \Rightarrow \texttt{lossless-spmf} (\texttt{key-gen} \eta))
by (cases (\eta, \sigma, \texttt{x}) rule: ekey-oracle\texttt{.cases})\texttt{(auto)}

lemma \texttt{lossless-encrypt-oracle} \texttt{[simp]}:
\[ (\sigma = \texttt{None} \Rightarrow \texttt{lossless-spmf} (\texttt{key-gen} \eta)) \]
by (cases (\eta, \sigma, \texttt{cipher}) \mapsto \texttt{auto})

1.3.2 Multi-user setting

definition \texttt{oracle}_n :: \texttt{bool} \Rightarrow \texttt{security}
\Rightarrow (i \Rightarrow (\texttt{ekey}, \texttt{'dkey'}, \texttt{'cipher}) \texttt{state-oracle}, i \times (\texttt{'plain'}, \texttt{'cipher}) \texttt{call}, (\texttt{ekey}, \texttt{'plain'}, \texttt{'cipher}) \texttt{ret}) \texttt{oracle}'
where \texttt{oracle}_n \texttt{b} \eta = \texttt{family-oracle} (\lambda. \texttt{oracle}_1 \texttt{b} \eta)

lemma \texttt{oracle}_n\texttt{-apply} \texttt{[simp]}:
\texttt{oracle}_n \texttt{b} \eta \texttt{s} (i, x) = \texttt{map-spmf} (\texttt{apsnd} (\texttt{fun-upd} s i)) (\texttt{oracle}_1 \texttt{b} \eta (s i) x)
by (simp add: \texttt{oracle}_n\texttt{-def})

type-synonym (\lambda. i, \texttt{'ekey'}, \texttt{'plain'}, \texttt{'cipher}) \texttt{adversary}_n =
(\texttt{bool}, i \times (\texttt{'plain'}, \texttt{'cipher}) \texttt{call}, (\texttt{ekey}', \texttt{'plain'}, \texttt{'cipher}') \texttt{ret}) \texttt{gpv}

type-synonym (\lambda. i, \texttt{'ekey'}, \texttt{'plain'}, \texttt{'cipher}) \texttt{adversary}_n =
\texttt{security} (i, \texttt{'ekey'}, \texttt{'plain'}, \texttt{'cipher}') \texttt{adversary}_n'

definition \texttt{ind-cca2}_n :: (\lambda. i, \texttt{'ekey'}, \texttt{'plain'}, \texttt{'cipher}) \texttt{adversary}_n \Rightarrow \texttt{security} \Rightarrow \texttt{bool} \texttt{spmfn}
where
\texttt{ind-cca2}_n \texttt{a} \eta = \texttt{TRY} \begin{cases} b \leftarrow \texttt{coin-spmf}; \\
\texttt{(guess, \sigma)} \leftarrow \texttt{exec-gpv} (\texttt{oracle}_n \texttt{b} \eta) (\texttt{a} \eta) (\lambda. \texttt{None}); \\
\texttt{return-spmf} (\texttt{guess} = b) \\
\end{cases}
\texttt{ELSE} \texttt{coin-spmf}

definition \texttt{advantage}_n :: (\lambda. i, \texttt{'ekey'}, \texttt{'plain'}, \texttt{'cipher}) \texttt{adversary}_n \Rightarrow \texttt{advantage}
where \texttt{advantage}_n \texttt{a} \eta = \texttt{spmfn} (\texttt{ind-cca2}_n \texttt{a} \eta) \texttt{True} - 1/2

lemma \texttt{advantage}_n\texttt{-nonneg}: \texttt{advantage}_n \texttt{a} \eta \geq 0 \texttt{by} (\texttt{simp add: \texttt{advantage}_n\texttt{-def})
1.4 The IND-CCA2 security for symmetric encryption schemes

theory IND-CCA2-sym imports
  CryptHOL.Computational-Model
begin
locale ind-cca =  
  fixes key-gen :: 'key spmf  
  and encrypt :: 'key ⇒ 'message ⇒ 'cipher spmf  
  and decrypt :: 'key ⇒ 'cipher ⇒ 'message option  
  and msg-predicate :: 'message ⇒ bool  
begin  

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

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

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

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

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

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

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

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

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

1.5 The IND-CPA game for symmetric encryption schemes

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

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

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

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

definition ind-cpa :: ('plain, 'cipher, 'state) adversary ⇒ bool spmf
where
  ind-cpa A = do
    let (A_1, A_2) = A;
    key ← key-gen;
    b ← coin-spmf;
    (guess, -) ← exec-gpv (encrypt-oracle key) (do
      ((m0, m1), σ) ← A_1;
      if valid-plain m0 ∧ valid-plain m1 then do
        cipher ← lift-spmf (encrypt key (if b then m0 else m1));
        σ_2 cipher σ
      } else lift-spmf coin-spmf
    ) ();
    return-spmf (guess = b)
definition advantage :: ('plain, 'cipher, 'state) adversary ⇒ real
  where advantage $\mathcal{A} = |\text{spmf (ind-cpa }\mathcal{A}\text{) }\text{True} - 1/2|

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

definition ibounded-by :: ('plain, 'cipher, 'state) adversary ⇒ enat ⇒ bool
  where ibounded-by = ($\lambda$ ($\mathcal{A}$1, $\mathcal{A}$2) q. ($\exists$ q1 q2. interaction-any-bounded-by $\mathcal{A}$1 q1 $\land$ ($\forall$ cipher $\sigma$. interaction-any-bounded-by ($\mathcal{A}$2 cipher $\sigma$) q2) $\land$ 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 $\forall$ cipher $\sigma$. interaction-any-bounded-by ($\mathcal{A}$2 cipher $\sigma$) q2
  using assms by(auto simp add: ibounded-by-def)

definition lossless :: ('plain, 'cipher, 'state) adversary ⇒ bool
  where lossless = ($\lambda$ ($\mathcal{A}$1, $\mathcal{A}$2). lossless-gpv $\mathcal{I}$-full $\mathcal{A}$1 $\land$ ($\forall$ cipher $\sigma$. lossless-gpv $\mathcal{I}$-full ($\mathcal{A}$2 cipher $\sigma$)))

end

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

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

locale ind-cpa-pk =
  fixes key-gen :: ('pubkey × 'privkey, 'call, 'ret) gpv — probabilistic
  and aencrypt :: 'pubkey ⇒ 'plain ⇒ ('cipher, 'call, 'ret) gpv — probabilistic w/ access to an oracle
  and adecrypt :: 'privkey ⇒ 'cipher ⇒ ('plain, 'call, 'ret) gpv — not used
  and valid-plains :: 'plain ⇒ 'plain ⇒ bool — checks whether a pair of plaintexts is valid, i.e., they have the right format
begin
We cannot incorporate the predicate valid-plain in the type 'plain of plaintexts, because the single 'plain must contain plaintexts for all values of the security parameter, as HOL does not have dependent types. Consequently, the game has to ensure that the received plaintexts are valid.

\[
\text{type-synonym } (\text{'pubkey', 'plain', 'cipher', 'call', 'ret', 'state}) \text{ adversary } = \\
(\text{'pubkey} \Rightarrow ((\text{'plain} \times \text{'plain}) \times \text{'state, 'call', 'ret'}) \text{ gpv}) \\
\times (\text{'cipher} \Rightarrow \text{'state } \Rightarrow \text{ (bool, 'call', 'ret') gpv})
\]

\[
\text{fun ind-cpa } :: (\text{'pubkey, 'plain, 'cipher, 'call, 'ret, 'state}) \text{ adversary } \Rightarrow \text{ (bool, 'call, 'ret) gpv }
\]

where

\[
\text{ind-cpa } (A1, A2) = \text{ TRY do }
\]
\[
(pk, sk) \leftarrow \text{ key-gen; } \\
b \leftarrow \text{ lift-spmf coin-spmf; } \\
((m0, m1), \sigma) \leftarrow (A1 pk); \\
\text{assert-gpv } (\text{valid-plains m0 m1}); \\
\text{cipher } \leftarrow \text{ aencryption pk (if } b \text{ then } m0 \text{ else m1); } \\
\text{guess } \leftarrow A2 \text{ cipher } \sigma; \\
\text{Done (guess } = b) \\
\} \text{ ELSE lift-spmf coin-spmf }
\]

\[
\text{definition advantage } :: (\sigma \Rightarrow \text{ 'call } \Rightarrow \text{ ('ret } \times \text{ 'σ) spmf}) \Rightarrow \text{ 'σ } \Rightarrow \text{ ('pubkey, 'plain, 'cipher, 'call, 'ret, 'state) adversary } \Rightarrow \text{ real }
\]

\[
\text{where advantage oracle } \sigma A \Rightarrow \text{ = spmf } \text{ (run-gpv oracle } (\text{ind-cpa } A) \text{ σ) } \text{ True } - 1/2 \]

\[
\text{lemma advantage-nonneg: advantage oracle } \sigma A \Rightarrow \text{ ≥ 0 by (simp add: advantage-def) }
\]

\[
\text{definition ibounded-by } :: (\text{'call } \Rightarrow \text{ bool}) \Rightarrow \text{ ('pubkey, 'plain, 'cipher, 'call, 'ret, 'state) adversary } \Rightarrow \text{ enat } \Rightarrow \text{ bool }
\]

\[
\text{where }
\]
\[
\text{ibounded-by consider } = (\lambda (A1, A2) q. } \\
(\exists q1 q2. (\forall pk. \text{ interaction-bounded-by consider } (A1 pk) q1) \land (\forall \text{ cipher } \sigma. \text{ interaction-}
\text{bounded-by consider } (A2 \text{ cipher } \sigma) q2) \land q1 + q2 \leq q)
\]

\[
\text{lemma ibounded-by'E [consumes 1, case-names ibounded-by', elim?]:}
\]
\[
\text{assumes ibounded-by consider } (A1, A2) q \\
\text{obtains q1 q2 }
\]

\[
\text{where q1 + q2 ≤ q }
\]
\[
\text{and } (\forall pk. \text{ interaction-bounded-by consider } (A1 pk) q1)
\]
\[
\text{and } (\forall \text{ cipher } \sigma. \text{ interaction-bounded-by consider } (A2 \text{ cipher } \sigma) q2)
\]
\[
\text{using assms by (auto simp add: ibounded-by-def) }
\]

\[
\text{lemma ibounded-by1 [intro?]:}
\]
\[
[ (\forall pk. \text{ interaction-bounded-by consider } (A1 pk) q1; \forall \text{ cipher } \sigma. \text{ interaction-}
\text{bounded-by consider } (A2 \text{ cipher } \sigma) q2; q1 + q2 \leq q ]
\]
\[
\implies \text{ ibounded-by consider } (A1, A2) q
\]
\[
\text{by (auto simp add: ibounded-by-def) }
\]

\[
\text{definition lossless } :: (\text{'pubkey, 'plain, 'cipher, 'call, 'ret, 'state) adversary } \Rightarrow \text{ bool }
\]
where lossless = (λ(\(A1, A2\)). (\(\forall pk. \text{lossless-gpv } \mathcal{I}\)-full (\(A1 pk\))) \land (\(\forall cipher \sigma. \text{lossless-gpv } \mathcal{I}\)-full (\(A2 cipher \sigma\)))))

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 (\(A1, A2\)) = TRY do {
(pk, sk) ← key-gen;
((m0, m1), σ) ← \(A1 pk\);
- :: unit ← assert-spmf (valid-plains m0 m1);
\(b\) ← coin-spmf;
cipher ← aencrypt pk (if \(b\) then m0 else m1);
\(b'\) ← \(A2 cipher \sigma\);
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
lossless $\iff$

$((\forall pk. \text{lossless-spmf } (\text{fit} \circ pk)) \land
(\forall cipher \sigma. \text{lossless-spmf } (\text{and} \circ cipher \sigma)))$

**lemma** lossless-ind-cpa:

\[ [\text{lossless } A ; \text{lossless-spmf } (\text{key-gen}) ] \implies \text{lossless-spmf } (\text{ind-cpa} A) \]

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

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, []))
  } | \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

decl: security ⇒ ((vkey, sigkey, message, signature) state-oracle, message, signature)

**where**

sign-oracle η None m = do { (-, σ) ← vkey-oracle η None (); sign-oracle η σ m }

A log. sign-oracle η (Some (vkey, skey, log)) m =
(if valid-message η m then do {
  sig ← sign η skey m;
  return-spmf (sig, Some (vkey, skey, (m, sig) # log))
} else return-pmf None)

by pat-completeness auto

**termination by** (relation Wellfounded.measure (λ(η, σ, m). case σ of None ⇒ I | - ⇒ 0)) auto

end

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

lossless-spmf (vkey-oracle η σ x) ⇔ (σ = None →→ lossless-spmf (key-gen η))

by (cases (η, σ, x) rule: vkey-oracle.cases) auto

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

[ σ = None →→ lossless-spmf (key-gen η);
  \{σ \text{ skey m, valid-message η m} \rightarrow lossless-spmf (sign η skey m)\}

implies lossless-spmf (sign-oracle η σ m) ⇔ valid-message η m

apply (cases (η, σ, m) rule: sign-oracle.cases)

apply (auto simp add: split-beta dest: lossless-spmfD-set-spmf-nonempty)

done

**lemma** lossless-sign-oracle-OrSome: fixes log shows

lossless-spmf (sign-oracle η (Some (vkey, skey, log)) m) ⇔ lossless-spmf (sign η skey m)

∧ valid-message η m

by (simp)

1.8.1 Single-user setting

**type-synonym** 'message' call₁ = unit + 'message'

**type-synonym** ('vkey', 'signature') ret₁ = 'vkey' + 'signature'

**definition** oracle₁ :: security

⇒ ((vkey, sigkey, message, signature) state-oracle, message call₁, (vkey, signature) ret₁) oracle'

**where** oracle₁ η = vkey-oracle η ⊕₀ sign-oracle η

**lemma** oracle₁-simps [simp]:

oracle₁ η s (Inl x) = map-spmf (apfst Inl) (vkey-oracle η s x)
oracle₁ η s (Inr y) = map-spmf (apfst Inr) (sign-oracle η s y)

by (simp-all add: oracle₁-def)

**type-synonym** ('vkey', 'message', 'signature') adversary₁' = ('message' × 'signature'), 'message' call₁, (vkey, signature) ret₁) gpv
type-synonym ('vkey', 'message', 'signature') adversary₁ =
  security ⇒ ('vkey', 'message', 'signature') adversary₁'

definition suf-cma₁ :: ('vkey', 'message', 'signature') adversary₁ ⇒ security ⇒ bool spmf
where
  \{\log. \text{suf-cma₁ } σ \not\in \eta\} = do {
    \langle m, sig, \rangle \leftarrow \text{exec-gpv } (\text{oracle₁ } \eta) (\not\in \eta) \text{ None};
    \text{return-spmf } \langle
      \begin{cases}
        \text{case } σ \text{ of } \text{None } \Rightarrow \text{False}
        \mid \text{Some } (\text{vkey, skey, log}) \Rightarrow \text{verify } \eta \text{ vkey m sig } \land (m, sig) \not\in \text{ set log}
      \end{cases}
    \rangle
  }

definition advantage₁ :: ('vkey', 'message', 'signature') adversary₁ ⇒ advantage
where
  advantage₁ σ \not\in \eta = \text{spmf } (\text{suf-cma₁ } σ \not\in \eta) \text{ True}

lemma advantage₁-nonneg: advantage₁ σ \not\in \eta \geq 0 \text{ by simp add: advantage₁-def pmf-nonneg}

abbreviation secure-for₁ :: ('vkey', 'message', 'signature') adversary₁ ⇒ bool
where
  secure-for₁ σ \not\in \eta ≜ \text{negligible } (\text{advantage₁ } σ)

definition ibounded-by₁' :: ('vkey', 'message', 'signature') adversary₁' ⇒ nat ⇒ bool
where
  ibounded-by₁' σ q = (interaction-any-bounded-by σ q)

abbreviation ibounded-by₁ :: ('vkey', 'message', 'signature') adversary₁ ⇒ (security ⇒ nat) ⇒ bool
where
  ibounded-by₁ ≜ rel-envir ibounded-by₁'

definition lossless₁' :: ('vkey', 'message', 'signature') adversary₁' ⇒ bool
where
  lossless₁' σ = (lossless-gpv σ full σ)

abbreviation lossless₁ :: ('vkey', 'message', 'signature') adversary₁ ⇒ bool
where
  lossless₁ ≜ pred-envir lossless₁'

1.8.2 Multi-user setting
definition oracleₙ :: security
  ⇒ ('i ⇒ ('vkey, 'sigkey', 'message', 'signature') state-oracle, 'i × 'message call₁, ('vkey, 'signature') ret₁) oracle'
where
  oracleₙ η = family-oracle (λ-. oracle₁ η)

lemma oracleₙ-apply [simp]:
  oracleₙ η \ s \ (i, x) = map-spmf (apsnd (fun-upd \ s \ i)) (oracle₁ \ η \ \ s \ i \ x)
by (simp add: oracleₙ-def)

type-synonym ('i, 'vkey', 'message', 'signature') adversaryₙ' =
  (\langle 'i × 'message × 'signature \rangle, 'i × 'message call₁, ('vkey, 'signature') ret₁) gpv
type-synonym ('i, 'vkey', 'message', 'signature') adversaryₙ =
  security ⇒ ('i, 'vkey', 'message', 'signature') adversaryₙ'
definition suf-cma :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ security ⇒ bool spmf
where
\forall \log. suf-cma \eta \Rightarrow \eta = do 
  ((i, m, sig), \sigma) ← exec-gpv (oracle_n \eta) (\lambda. None);
  return-spmf (
    case \sigma i of None ⇒ False
    | Some (vkey, skey, log) ⇒ verify \eta vkey m sig ∧ (m, sig) /∈ set log)
}

definition advantage :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ advantage
where
advantage_n \eta = spmf (suf-cma_n \eta) True

lemma advantage_n-nonneg: advantage_n \eta \geq 0 by (simp add: advantage_n-def pmf-nonneg)

abbreviation secure-for :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ bool
where
secure-for_n \eta = negligible (advantage_n \eta)

definition ibounded-by :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ nat ⇒ bool
where
ibounded-by_n q = (interaction-any-bounded-by q)

abbreviation ibounded-by :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ (security ⇒ nat) ⇒ bool
where
ibounded-by_n ≡ rel-envir ibounded-by_n'

definition lossless :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ bool
where
lossless_n = (lossless-gpv \eta \Rightarrow \eta)

abbreviation lossless :: (i, 'vkey, 'message, 'signature) adversary_n ⇒ bool
where
lossless_n = pred-envir lossless_n'

end

end

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

1.9 Pseudo-random function

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

type-synonym ('b,'a) dict = 'b ⇒ 'a

definition random-oracle :: ('b, 'a) dict ⇒ 'b ⇒ ('a × ('b, 'a) dict) spmf
where
random-oracle $\sigma$ $x$

\[(\text{case } \sigma \text{ of Some } y \Rightarrow \text{return-spmf } (y, \sigma) \\
| \text{None } \Rightarrow p \gg= (\lambda y. \text{return-spmf } (y, \sigma(x \mapsto y))) )\]

definition forgetful-random-oracle :: unit $\Rightarrow$ 'b $\Rightarrow$ ('a $\times$ unit) spmf

where

forgetful-random-oracle $\sigma$ $x$ $=$ $p \gg= (\lambda y. \text{return-spmf } (y, ()))$

lemma weight-random-oracle [simp]:

weight-spmf $p$ $=$ $1$ $\Rightarrow$ weight-spmf (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 $p$ $=$ $\Rightarrow$ lossless-spmf (random-oracle $\sigma$ $x$)

by(simp add: lossless-spmf-def)

subsection finite: callee-invariant-on random-oracle $\lambda \sigma$. finite (dom $\sigma$). $\mathcal{I}$-full

by(unfold-locales)(auto simp add: random-oracle-def split: option.split: option.split-asm)

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 (dom $\sigma$)

shows card (dom $\sigma'$) $\leq$ $q +$ card (dom $\sigma$)


end

1.10 Pseudo-random function

locale prf =

fixes key-gen :: 'key spmf

and prf :: 'key $\Rightarrow$ 'domain $\Rightarrow$ 'range

and rand :: 'range spmf

begin

sublocale random-function rand .

definition prf-oracle :: 'key $\Rightarrow$ unit $\Rightarrow$ 'domain $\Rightarrow$ ('range $\times$ unit) spmf

where prf-oracle key $\sigma$ $x$ $=$ return-spmf (prf key $x$, ())

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

definition game-0 :: ('domain', 'range') adversary $\Rightarrow$ bool spmf

where

game-0 $\mathcal{A}$ $=$ do 

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

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

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

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

abbreviation lossless :: ('domain, 'range) adversary ⇒ bool
where
lossless \( \equiv \) lossless-gpv I-full

abbreviation (input) ibounded-by :: ('domain, 'range) adversary ⇒ enat ⇒ 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 → 'b) ⇒ 'a ⇒ ('b × ('a → 'b)) spmf
where
random-permutation \( \sigma \) \( x \) =
  (case \( \sigma \) \( x \) of Some \( y \) ⇒ return-spmf (y, \( \sigma \))
   | None ⇒ spmf-of-set (A - ran \( \sigma \)) \( \equiv \) (\( \lambda \)y. return-spmf (y, \( \sigma \)(x → y))))

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

lemma lossless-random-oracle [simp]:
\[ \text{finite } A; A - \text{ran } \sigma \neq \{ \} \implies \text{lossless-spmf (random-permutation } \sigma \) \( x \) \]

19
by (simp add: lossless-spmf-def)

sublocale finite: callee-invariant-on random-permutation λσ. finite (dom σ) \mathcal{F}-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 \( (y, \sigma') \in \text{set-spmf} \ (\text{exec-gpv random-permutation} \ \mathcal{A} \ \sigma) \)
  and \( \text{fin: finite} \ (\text{dom} \ \sigma) \)
  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-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 :: \( \text{'}c-o\times\text{'}c-a\times\text{'s} \) spmf
  and oracle :: \( \text{'}c-o\Rightarrow\text{'s}\Rightarrow\text{'call}\Rightarrow\text{'}{\text{ret}\times\text{'s}} \) spmf
  and eval :: \( \text{'}c-o\Rightarrow\text{'}c-a\Rightarrow\text{'s}\Rightarrow\text{'guess}\Rightarrow\text{bool} \) spmf
begin

type-synonym \( \text{'}c-a\times\text{'guess} \) adversary-single = \( \text{'}c-a\Rightarrow\text{'}{\text{guess}\times\text{'call}\times\text{'ret}} \)

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

definition advantage-single :: \( \text{'}c-a\times\text{'guess}\times\text{'call}\times\text{'ret} \) adversary-single \Rightarrow real
  where advantage-single \mathcal{A} = spmf (game-single \mathcal{A}) True


type-synonym \( \text{'}c-a\times\text{'guess} \) adversary-many = \( \text{'}c-a\Rightarrow\text{'}{\text{unit}\times\text{'call}}\times\text{'guess} \times\text{'ret}\times\text{'unit} \)

20
definition eval-oracle :: 'c-o ⇒ 'c-a ⇒ bool × 's ⇒ 'guess ⇒ (unit × (bool × 's)) spmf
where
eval-oracle c-o c-a = (λ(b, s') guess. map-spmf (λb'. ((b ∨ b', s'))) (eval c-o c-a s' guess))

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

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

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

primrec process-call :: 'guess reduction-state ⇒ 'call ⇒ ('ret option × 'guess reduction-state, 'call, 'ret) gpv
where
process-call (Inr j) x = do {
    ret ← Pause x Done;
    Done (Some ret, Inr j)
} | process-call (Inl guess) x = Done (None, Inl guess)

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

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

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

**Lemma** many-single-reduction:
\[\text{Assumes bound: } \forall c \cdot \alpha \cdot s . \ (c \cdot \alpha, c \cdot a, s) \in \text{set-spmf init} \implies \text{interaction-bounded-by } (\text{Not } o \isl) \ (\forall c \cdot \alpha) \ q\]
\[\text{And lossless-oracle: } \forall c \cdot \alpha \cdot s \ s' x . \ (c \cdot \alpha, c \cdot a, s) \in \text{set-spmf init} \implies \text{lossless-spmf (oracle } c \cdot \alpha, c \cdot a, s' x)\]
\[\text{And lossless-eval: } \forall c \cdot \alpha \cdot s \ s' \cdot \text{guess}. \ (c \cdot \alpha, c \cdot a, s) \in \text{set-spmf init} \implies \text{lossless-spmf (eval } c \cdot \alpha, c \cdot a, s' \cdot \text{guess)}\]
\[\text{Shows advantage-multi } \mathcal{A} \leq \text{advantage-single (reduction } q \mathcal{A} \ast q)\]

**Including** lifting-syntax
\[\text{Proof} –\]
\[\text{Define } \text{eval-oracle}'\]
\[\text{Where } \text{eval-oracle}' = (\lambda c \cdot \alpha \cdot c \cdot a . \ (\text{id, occ :: nat option}), s') \cdot \text{guess.}\]
\[\text{map-spmf } (\lambda b'. \text{case occ of } \text{Some } j_0 \Rightarrow (((() , (\text{Suc id, Some } j_0), s')) | \text{None} \Rightarrow (((() , (\text{Suc id, if } b' \text{ then Some id else None})), s')))\]
\[\text{Eval } c \cdot \alpha \cdot c \cdot a \cdot s' \cdot \text{guess})\]
\[\text{Let } \text{?multi'}-body = \lambda c \cdot \alpha \cdot c \cdot a \cdot \text{exec-gpv } (\uplus \text{oracle } c \cdot \alpha) \oplus_0 \text{eval-oracle'} c \cdot \alpha \cdot c \cdot a (\mathcal{A} \cdot c \cdot a) ((0, \text{None}), s)\]

Define **game-multi' where**
\[\text{game-multi'} = (\lambda c \cdot \alpha \cdot c \cdot a \cdot s . \text{do} \{- , ((\text{id, } j_0), s' :: 's) \mapsto \text{multi'-body } c \cdot \alpha \cdot c \cdot a \cdot s;\]
\[\text{return-spmf } (j_0 \neq \text{None})\} )\]

Define **initialize** :: \((c \cdot o \Rightarrow 'c \cdot a \Rightarrow 's \Rightarrow \text{nat } \Rightarrow \text{bool spmf}) \Rightarrow \text{bool spmf where}\
\[\text{initialize body} = \text{do} \{\]
\[\ (c \cdot a, c \cdot a, s) \mapsto \text{init};\]
\[\ j_s \Leftarrow \text{spmfof-set } \{..<q\};\]
\[\ \text{body } (c \cdot \alpha, c \cdot a, s) j_s \}\ for \text{body}\]

Define **body2 where**
\[\text{body2 } c \cdot a \; c \cdot a \; s \; j_s = \text{do} \{\]
\[\ (- , (\text{id, } j_0), s') \mapsto \text{?multi'-body } c \cdot a \; c \cdot a \; s;\]
\[\ \text{return-spmf } (j_0 = \text{Some } j_s) \}\ for \text{c } c \cdot a \; c \cdot a \; s \; j_s\]

Let **?game2 = initialize body2**

Define **stop-oracle where**
\[\text{stop-oracle} = (\lambda c \cdot \alpha .\]
\[\ (\lambda (\text{idgs, } s) x . \text{case idgs of } \text{Inr } - \Rightarrow \text{map-spmf } (\lambda (y, s) . \ (\text{Some } y, (\text{idgs, } s))) \ (\text{oracle } c \cdot \alpha \ s x) | \text{Inl } - \Rightarrow \text{return-spmf } (\text{None, } (\text{idgs, } s)))\]
\[\oplus_0^\circ\]
\[\ (\lambda (\text{idgs, } s) \cdot \text{guess} :: '\text{guess. return-spmf } (\text{case idgs of } \text{Inr } 0 \Rightarrow (\text{None, Inl } (\text{guess, s}), s) | \text{Inr } (\text{Suc } i) \Rightarrow (\text{Some } (), \text{Inr } i, s) | \text{Inl } - \Rightarrow (\text{None, } (\text{idgs, } s))))\]

Define **body3 where**
\[\text{body3 } c \cdot a \; c \cdot a \; s \; j_s = \text{do} \{\]
\[\ (- :: \text{unit option, idgs, } -) \mapsto \text{exec-gpv-stop } (\text{stop-oracle } c \cdot a) (\mathcal{A}' \cdot c \cdot a) (\text{Inr } j_s, s);\]
\[\ (b' :: \text{bool}) \mapsto \text{case idgs of } \text{Inr } - \Rightarrow \text{return-spmf False } | \text{Inl } (g, s') \Rightarrow \text{eval } c \cdot \alpha \cdot c \cdot a \; s' \; g;\]
\[\ \text{return-spmf } b' \}\ for \text{c } c \cdot a \; c \cdot a \; s \; j_s\]

Let **?game3 = initialize body3**

\[\{ \text{Define } S \; :: \text{bool } \Rightarrow \text{nat } \times \text{nat option } \Rightarrow \text{bool where} \ S \equiv \lambda b'(id, occ). b' \mapsto (\exists j_0. \ occ = \text{Some } j_0)\]
\[\ \text{Let } ?S = \text{rel-prod } S (=)\]
\(\text{define \ initial :: nat \times nat \ option \ where \ initial = (0, None)}\)
\(\text{define result :: nat \times nat \ option \Rightarrow bool \ where \ result \ p = (\text{snd} \ p \neq \ None) \ for} \ p\)

\[\text{have [transfer-rule]: } (\text{S} \Longrightarrow (=)) \ (\lambda b. b) \ \text{result by (simp add: rel-fun result-def \ S-def)}\]

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

\[\text{have eval-oracle \ [\text{transfer-rule}]:} \]
\[\text{((=) \Longrightarrow (=) \Longrightarrow \ ?S \ \Longrightarrow (=) \Longrightarrow \ rel-spmf \ (rel-prod \ (=) \ ?S))}\]

\[\text{eval-oracle eval-oracle'}\]

\[\text{unfolding eval-oracle-def [abs-def] eval-oracle'}-\text{def [abs-def]}\]

\[\text{by (auto simp add: rel-fun-def spmf-bind-spmf intro!: rel-spmf-bind-reflI split: option.split)}\]

\[\text{have game-multi'}: \text{game-multi} s' = \text{bind-spmf init} \ (\lambda \ (c-o, c-a, s) \ \text{game-multi} \ c-o \ c-a \ s)\]

\[\text{unfolding game-multi'-def game-multi'}-\text{def initial-def [symmetric]}\]

\[\text{by (rewrite in case-prod \ \text{in} \ \text{bind-spmf} - \ (case-prod \ \text{in} - \ \text{bind-spmf} - \ \text{split-def})}\]

\[\text{(fold result-def; transfer-prover) }\]

\[\text{moreover}\]

\[\text{have spmf \ (game-multi' \ c-o \ c-a \ s) \ True = spmf \ (bind-spmf \ (spmf-of-set \ 
\{..<q\}) \ (body2 \ c-o \ c-a \ s)) \ True \ast q}\]

\[\text{if} \ (c-o, c-a, s) \in \text{set-spmf init for} \ c-o \ c-a \ s\]

\[\text{proof –}\]

\[\text{have bnd: interaction-bounded-by (Not o isl) (s' c-a) \ q \ using bound that by blast}\]

\[\text{have bound-occ: } j_s < q \ \text{if that:} \ ((\), (id, Some \ j_s), s') \in \text{set-spmf} \ (\text{multi'}-body \ c-o \ c-a \ s)\]

\[\text{for} \ s' \ id \ j_s\]

\[\text{proof –}\]

\[\text{have id} \leq q\]

\[\text{by (rule o-True.interaction-bounded-by'}-\text{exec-gpv-count[OF bnd that, where count=fst o fst, simplified])}\]

\[\text{(auto simp add: eval-oracle'}-\text{def split: plus-oracle-split-asnm option.split-asnm)}\]

\[\text{moreover let} \ ?I = \lambda \ ((id, \ \text{occ}), s'). \ \text{case occ of None \Rightarrow True | Some} \ j_s \Rightarrow j_s \leq id\]

\[\text{have callee-invariant (\text{\textasciitilde}{oracle c-o} \ \text{\textcircled{=} eval-oracle'} c-o c-a) \ ?I}\]

\[\text{by (clarsimp simp add: split-def intro!: conj[OF callee-invariant-extend-state-oracle-const \ \text{\textasciitilde}])}\]

\[\text{unfold-locals; auto simp add: eval-oracle'}-\text{def split: option.split-asnm)}\]

\[\text{from callee-invariant-on.exec-gpv-invariant[OF this that] have} \ j_s < id \ \text{by simp}\]

\[\text{ultimately show \ ?thesis by simp}\]

\[\text{qed}\]

\[\text{let} \ ?M = \text{measure (measure-spmf (\text{multi'}-body c-o c-a s))}\]

\[\text{have spmf \ (game-multi' \ c-o \ c-a \ s) \ True = \ ?M \ (\{u, (id, j_0), s', j_0 \neq None\}\}}\]

\[\text{by(auto simp add: game-multi'}-\text{def map-spmf-conv-bind-spmf [symmetric] split-def spmf-conv-measure-spmf measure-map-spmf vimage-def)}\]

\[\text{also have} \ \{((\), (id, \text{Some} \ j_s), s') \mid j_s'' \ \text{id} \ j_s < q\} \cup \{((\), (id, \text{Some} \ j_s), s') \mid j_s'' \ \text{id} \ j_s \geq q\}\]

\[\text{(is - = \ ?A \cup -) by auto}\]

23
The text contains a proof or a fragment of code with various functions and variables. It appears to be a snippet from a formal proof or a program, involving lambda calculus, pattern matching, and case analysis. The context suggests it might be from a computer science or mathematics document, possibly related to formal verification or proof assistants.

```plaintext
let \(?R = \lambda((id1, j0), s1) (b', (id2, gs), s2). s1 = s2 \land id1 = id2 \land (j0 = Some j_s \Rightarrow b') = Some True) \land (id2 \leq j_s \Rightarrow b' = None)

from \text{init} have rel-spmf (rel-prod (=) ?R)
(\text{exec-gpv (extend-state-oracle (oracle c-o) \oplus_O eval-oracle') c-o c-a}) (\text{rel-spmf-reflI} ((0, None), s))
(\text{exec-gpv (extend-state-oracle (oracle c-o)) \oplus_O oracle2')} (\text{rel-spmf-reflI} ((0, None), s))
then have rel-spmf (\text{body2 c-o c-a s j_s})
(\text{do} \{\neg, b', \neg, \neg \leftarrow \text{exec-gpv} (\uplus (\text{oracle c-o}) \oplus_O oracle2') (\text{rel-spmf-reflI} ((0, None), s));
\text{return-spmf} (b' = Some True) \})
(\text{is rel-spmf - - ?body2'})
— We do not get equality here because the right hand side may return True even when the bad event has happened before the j_s-th iteration.
\text{unfolding body2-def by(rule rel-spmf-bindII) clarsimp}
also
let \(?I = \lambda(id1, gs, s) \text{ guess. return-spmf} (((), (Suc id, if id = j_s, then Some (guess, s) else gs), s)
\text{let \(?I = \lambda(id1, gs, s) \text{ case idgs of \{, None \Rightarrow False | \{ (i, Some \cdot) \Rightarrow j_s < i \}
\text{interpret \(I: \text{ callee-invariant-on \(\uplus (\text{oracle c-o}) \oplus_O \text{guess-oracle} \?I \?\text{-full}
\text{by(simp)(unfold-locals; auto split: option.split)
let \(?f = \lambda s. \text{ case snd (fst s) of None \Rightarrow return-spmf False | Some a \Rightarrow eval c-o c-a (snd a) (fst a)
\text{let \(?X = \lambda j_s (b1, (id1, gs1), s1) (b2, (id2, gs2), s2). b1 = b2 \land id1 = id2 \land gs1 = gs2 \land s1 = s2 \land (b2 = None \leftrightarrow gs2 = None) \land (id2 \leq j_s \rightarrow b2 = None)
\text{have \?body2' = do} \{\neg, r, s \leftarrow \text{exec-gpv} (\lambda(r, s) x. \text{ do} \{\neg, s' \leftarrow (\uplus (\text{oracle c-o}) \oplus_O \text{guess-oracle}) s x;
\text{if \(?I s' \land \neg \Rightarrow \text{ then return-spmf} (\lambda r. (y, Some r, s')) (\?f s') \text{ else return-spmf (\(y, r, s') \})\}
(\text{rel-spmf-reflII cong: \text{ conj-cong split: plus-oracle-split})
\text{also have \... = do} \{\neg, s' \leftarrow \text{exec-gpv} (\uplus (\text{oracle c-o}) \oplus_O \text{guess-oracle}) (\text{rel-spmf-reflII cong: \text{ conj-cong split: plus-oracle-split})
\text{by(rule 1.exec-gpv-bind-materialize[\text{symmetric}])(auto split: plus-oracle-split-asm option.split-asm)
```

The text is quite dense and technical, indicating a high level of detail in formal verification or proof writing.
also have \ldots = \text{do}
\begin{itemize}
\item \text{us}' \leftarrow \text{exec-gpv-stop} (\text{lift-stop-oracle} (\uparrow (\text{oracle c-o}) \oplus_{\text{o}} \text{guess-oracle})) (\alpha' \ x) ((0, \text{None}), \text{s})
\item \text{(b' :: bool) } \leftarrow \text{?f} (\text{snd us}')
\item \text{return-spmf b'}
\end{itemize}
supply \text{lift-stop-oracle-transfer}[\text{transfer-rule}] \text{gpv-stop-transfer}[\text{transfer-rule}] \text{exec-gpv-parametric}[\text{transfer-rule}]
by \text{transfer simp}
also let \text{?S = } (\lambda (\text{idI}, \text{gsI}), \text{sI}) (\text{(idI}, \text{gsI}), \text{sI}) \text{. gsI} = \text{gsI} \land (\text{gsI} = \text{None } \rightarrow \text{sI} = \text{sI} \land \text{idI} = \text{idI} \land (\text{gsI} = \text{None } \leftarrow \text{idI} \leq j_i))
\begin{itemize}
\item \text{have ord-spmf} (\rightarrow) \ldots (\text{exec-gpv-stop} ((\lambda (\text{id}, \text{gs}), \text{s} \cdot \text{case gs of None } \rightarrow \text{lift-stop-oracle} (\uparrow (\text{oracle c-o}) \oplus_{\text{o}} \text{guess-oracle})) (\text{(id}, \text{gs}), \text{s}) \cdot \text{Some - } \rightarrow \text{return-spmf} (\text{None}, ((\text{id}, \text{gs}), \text{s}))) \oplus_{\text{o}}^{5} (\lambda (\text{id}, \text{gs}), \text{s} \cdot \text{guess. return-spmf} (\text{if id} \geq j_i \text{ then None else Some } (\text{Suc id}), \text{id}) = j_i \text{ then Some (guess, s} \text{ else gs}), \text{s}))
\item \text{(\alpha' \ x) ((0, \text{None}), \text{s} \cdot \Rightarrow (\lambda \text{us}', \text{case snd (fist (snd us'))} ) \text{ of None } \rightarrow \text{return-spmf False} \text{ Some a } \rightarrow \text{eval c-o c-a (snd a) (fist a))}
\end{itemize}
unfolding \text{body3-def stop-oracle-def}
by (\text{rule ord-spmf-exec-gpv-stop}[\text{where} \text{stop} = \lambda ((\text{id}, \text{guess}), \cdot). \text{guess } \neq \text{None and S=2S, THEN ord-spmf-bindI}])
also let \text{?X = } (\lambda (\text{gsid, s}), \text{sI}) (\text{(gsid, sI)}), \text{sI} = \text{sI} \land \text{(case (gs, idgs) of (None, Inr id') } \Rightarrow \text{id' = j_i - id } \land \text{id } \leq j_i \mid (\text{Some gs, IntId gs' } \Rightarrow \text{gs = gs' } \land \text{id } > j_i \mid - \Rightarrow \text{False})
\begin{itemize}
\item \text{have \ldots = body3-def spmf-rel-eq[\text{symmetric}] stop-oracle-def}
\item \text{by (\text{rule exec-gpv-oracle-bisim}[\text{where} \text{X=?X, THEN rel-spmf-bindI}]})
\end{itemize}
(auto split: option.split-asm plus-oracle-stop-split nat.splits split!: sum.split simp add: spmf-rel-map intro!: rel-spmf-reflI)
\begin{itemize}
\item \text{finally show} \text{?thesis by (\text{rule spmf-rel-mono-strong}) (auto elim!: option.rel-cases ord-option.cases)}\end{itemize}
\begin{itemize}
\item \text{then have ord-spmf (\rightarrow) ?game2 ?game3}
\item \text{by (clarsimp simp add: initialize-def intro!: ord-spmf-bindrefI)}
\item \text{also let \text{?X = } (\lambda (\text{gsid, s}), \text{gsid}, \text{sI}) (\text{(gsid, sI)}), \text{sI = sI } \land \text{rel-sum (\lambda (g, sI}, \text{g'}, \text{g = g'} \land \text{sI = sI}) = \text{gsid gid}}
\item \text{have rel-spmf (\rightarrow) ?game3 (game-single (\text{reduction q \alpha'})}
\item \text{unfolding body3-def stop-oracle-def game-single-def \text{reduction-def split-def initial-def}}
\item \text{apply (clarsimp simp add: bind-map-spmf exec-gpv-bind exec-gpv-inline intro!: rel-spmf-bindrefI)}
\item \text{apply (rule rel-spmf-bindI[OF exec-gpv-oracle-bisim[\text{where} \text{X=?X}]])}
\end{itemize}
\begin{itemize}
\item \text{apply (auto split: plus-oracle-stop-split elim!: rel-sum.cases simp add: map-spmf-conv-bind-spmf[\text{symmetric}] split-def spmf-rel-map rel-spmf-reflI rel-spmf-return-spmf1 lossless-evl split: nat.split)}
\end{itemize}
\begin{itemize}
\item \text{done}
\item \text{finally have ord-spmf (\rightarrow) ?game2 (game-single (\text{reduction q \alpha'})\end{itemize}
\begin{itemize}
\item \text{by (rule spmf-rel-mono-strong) (auto elim!: option.rel-cases ord-option.cases)}
\text{from this)[THEN ord-spmf-measureD, of \{True\}]}
\text{have spmf ?game2 True } \leq \text{ spmf (game-single (\text{reduction q \alpha'})\text{ True unfolding spmf-conv-measure-spmf}}
\text{by (rule ord-le-eq-trans) (auto intro: arg-cong2[\text{where f=measure}])}
\end{itemize}
\begin{itemize}
\item \text{ultimately show ?thesis unfolding advantage-multi-def advantage-single-def}
by (simp add: mult-right-mono)
qed

end

end

1.13 Unpredictable function

theory Unpredictable-Function imports Guessing-Many-One begin

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

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

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

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

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

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

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

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

27
definition game1 :: (x, 'hash) adversary1 ⇒ bool spmf
where
game1 $A$ = do
key ← key-gen;
((m, h), L) ← exec-gpv (oracle-hash key) $A$ {}; 
return-spmf (h = hash key m ∧ m /∈ L)
}

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

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


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

end

end

theory Security-Spec imports
  Diffie-Hellman
  IND-CCA2
  IND-CCA2-sym
  IND-CPA
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 \( \mathcal{G} :: \text{`grp cyclic-group (structure)} \)
begin

  type-synonym \text{`grp pub-key} = \text{`grp'}
  type-synonym \text{`grp priv-key} = \text{nat}
  type-synonym \text{`grp plain} = \text{`grp'}
  type-synonym \text{`grp cipher} = \text{`grp'} \times \text{`grp'}

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

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

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

end
lemma aencrypt-all:
aencrypt α msg = map-spmf (λ y. (g[^] y, (α[^] y) ⊗ msg)) (sample-uniform (order G))
by(simp add: map-spmf-conv-bind-spmf aencrypt-def)

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

abbreviation valid-plains :: 'grp ⇒ 'grp ⇒ bool
where
valid-plains msg1 msg2 ≡ msg1 ∈ carrier G ∧ msg2 ∈ carrier G

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

fun elgamal-adversary :: ('grp pub-key, 'grp plain, 'grp cipher, 'state) ind-cpa.adversary ⇒ 'grp ddh.adversary
where
elgamal-adversary (A1, A2) α β γ = TRY do
  b ← coin-spmf;
  ((msg1, msg2), σ) ← A1 α;
  — have to check that the attacker actually sends two elements from the group; otherwise
  flip a coin
  - :: unit ← assert-spmf (valid-plains msg1 msg2);
  guess ← A2 (β, γ ⊗ (if b then msg1 else msg2)) σ;
  return-spmf (guess ←→ b)
  } ELSE coin-spmf

end

locale elgamal = elgamal-base + cyclic-group G
begin

theorem advantage-elgamal: ind-cpa.advantage A = ddh.advantage (elgamal-adversary A)
including monad-normalisation
proof –
  obtain A1 and A2 where A = (A1, A2) by(cases A)
  note [simp] = this order-gr-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 A) = TRY do {
    x ← sample-uniform (order G);
    y ← sample-uniform (order G);
    (msg1, msg2, σ) ← A1 (g[^] x);
    - :: unit ← assert-spmf (valid-plains msg1 msg2);
    b ← coin-spmf;
    z ← map-spmf (λ z. g[^] z ⊗ (if b then msg1 else msg2)) (sample-uniform (order G));
    guess ← A2 (g[^] y, z) σ;
    return-spmf (guess ←→ b)
  }

end
} ELSE coin-spmf
by(simp add: ddh.ddh-1-def)
also have ... = TRY do {
    x ← sample-uniform (order γ);
    y ← sample-uniform (order γ);
    (msg1, msg2, σ) ← 1 (g [^] x);
    - :: unit ← assert-spmf (valid-plains msg1 msg2);
    z ← map-spmf (λz. g [^] z) (sample-uniform (order γ));
    guess ← A2 (g [^] y, z) σ;
    map-spmf (λz. 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 A) = ind-cpa.ind-cpa A
ultimately show thesis by(simp add: ddh.advantage-def ind-cpa.advantage-def)
qed

end

locale elgamal-asymp =
  fixes G :: security ⇒ 'grp cyclic-group
  assumes elgamal: ∀γ. elgamal (G γ)
begin

sublocale elgamal γ η for η by(simp add: elgamal)

theorem elgamal-secure:
egligible (λη. ind-cpa.advantage η (A η)) if negligible (λη. ddh.advantage η (elgamal-adversary η (A η)))
by(simp add: advantage-elgamal that)

end

context elgamal-base begin

lemma lossless-key-gen [simp]: lossless-spmf (key-gen) ←→ 0 < order γ
by(simp add: key-gen-def Let-def)

lemma lossless-aencrypt [simp]:
  lossless-spmf (aencrypt key plain) ←→ 0 < order γ
by(simp add: aencrypt-def Let-def)

lemma lossless-elgamal-adversary:
  ind-cpa.lossless A; 0 < order γ 
  ⇒⇒ ddh.lossless (elgamal-adversary A)
by(cases A)(simp add: ddh.lossless-def ind-cpa.lossless-def Let-def split-def)

31
2.2 Hashed Elgamal in the Random Oracle Model

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

type-synonym bitstring = bool list

locale hash-oracle =
fixes len :: nat
begin

type-synonym 'a state = 'a ⇒ bitstring

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

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

inductive invariant :: 'a state ⇒ bool
where
invariant: ![ finite (dom σ); length ' ran σ ⊆ {len} ] ⇒ invariant σ

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

lemma invariant-update [simp]: ![ invariant σ; length bs = len ] ⇒ invariant (σ(x ↦→ bs))
by (auto simp add: invariant.simps ran-def)

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

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

lemma lossless-oracle [simp]: lossless-spmf (oracle σ x)

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

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

end

locale elgamal-base =
  fixes G :: `'grp cyclic-group` (structure)
and len-plain :: nat
begin

sublocale hash: hash-oracle len-plain .
abbreviation hash :: `'grp \Rightarrow (\text{bitstring}, `grp`, \text{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 \text{sample-uniform} (\text{order} \ G);
  \text{return-spmf} \ (g [^x] \ x, \ x)
}
definition aencrypt :: `'grp` pub-key \Rightarrow plain \Rightarrow (\text{cipher}, `'grp`, \text{bitstring}) gpv
where
  aencrypt \(x\) \(msg\) = do 
  \(y\) \leftarrow \text{lift-spmf} \ (\text{sample-uniform} (\text{order} \ G));
\[
\begin{align*}
  & h \leftarrow \text{hash}(\alpha[^*]y); \\
  & \text{Done}(g[^*]y, h[^*]msg)
\end{align*}
\]

**definition** adecrypt :: 'grp priv-key ⇒ 'grp cipher ⇒ (plain', 'grp, bitstring) gpv

**where**

adecrypt x = (\lambda (\beta, \zeta). do {
  h \leftarrow \text{hash}(\beta[^*]x); \\
  \text{Done}(\zeta[^*]h)
})

**definition** valid-plains :: plain ⇒ plain ⇒ bool

**where**

valid-plains msg1 msg2 ←→ length msg1 = len-plain \land length msg2 = len-plain

**lemma** lossless-aencrypt [simp]: lossless-gpv \(I\) (aencrypt \(\alpha\) msg) ←→ 0 < order \(G\)

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

**lemma** interaction-bounded-by-aencrypt [interaction-bound, simp]:

interaction-bounded-by (\(\lambda -. \text{True}\)) (aencrypt \(\alpha\) msg) 1

**unfolding** aencrypt-def **by** interaction-bound(simp add: one-enat-def SUP-le-iff)

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

**sublocale** lcdh: lcdh \(G\) .

**fun** elgamal-adversary

\(:: (\text{"grp pub-key, plain, "grp cipher, "grp, bitstring, "state}) \text{ind-cpa.adversary} \Rightarrow \text{"grp lcdh.adversary}\)

**where**

elgamal-adversary (\(\alpha1, \alpha2\) \(\alpha\beta\)) = do {
  \(((\text{msg1, msg2, }\sigma), s) \leftarrow \text{exec-gpv hash.oracle} (\alpha1, \text{hash.initial});
  \text{— have to check that the attacker actually sends an element from the group; otherwise stop early}
  \text{TRY do \{}
    - :: \text{unit} \leftarrow \text{assert-spmf} \text{(valid-plains msg1 msg2)};
    h' \leftarrow \text{spmf-of-set} \text{(nlists UNIV len-plain)};
    \langle \text{guess, s'} \rangle \leftarrow \text{exec-gpv hash.oracle} (\alpha2, h') \sigma s;
    \text{return-spmf} \text{ (dom s')}
  \}\ ELSE \text{return-spmf} \text{ (dom s)}
}

**end**

**locale** elgamal = elgamal-base +

**assumes** cyclic-group: cyclic-group \(G\)

**begin**

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

**lemma** advantage-elgamal:
We change the adversary's oracle to record the queries made by the adversary

define hash-oracle' where hash-oracle' = (λσ x. do {
  h ← hash x;
  Done (h, insert x σ)
})

have [simp]: lossless-gpv J-full (hash-oracle' σ x) for σ x by (simp add: hash-oracle'-def)

by (rule lossless-inline where J = J-full) simp-all

define game0 where game0 = TRY do {
  (pk, -) ← lift-spmf key-gen;
  b ← lift-spmf coin-spmf;
  ( ((msg1, msg2), σ), s ) ← inline hash-oracle' (A 1 pk) { };
  assert-gpv (valid-plains msg1 msg2);
  cipher ← aencrypt pk (if b then msg1 else msg2);
  (guess, s') ← inline hash-oracle' (A 2 cipher σ) s;
  Done (guess = b)
) ELSE lift-spmf coin-spmf

{ define cr where cr = (λ· :: unit. λ· :: 'a set. True) 

  by (simp add: cr-def)

  have [transfer-rule]: cr () {} }

  have [transfer-rule]: ( (= ) =====> cr =====> cr) (λ· σ. σ) insert by (simp add: rel-fun-def cr-def)

  have [transfer-rule]: ( cr =====> ( = ) =====> rel-gpv (rel-prod (=) cr) ( = ) ) id-oracle hash-oracle'

  unfolding hash-oracle'-def id-oracle-def[abs-def] bind-gpv-Pause bind-rpv-Done by transfer-prover

  have ind-cpa.ind-cpa A = game0 unfolding game0-def A ind-cpa-pk.ind-cpa.simps

  by (transfer fixing: A len-plain A 1 A 2) (simp add: bind-map-gpv o-def ind-cpa-pk.ind-cpa.simps split-def ) }

note game0 = this

have game0-alt-def: game0 = do {
  x ← lift-spmf (sample-uniform (order A));
  b ← lift-spmf coin-spmf;
[((msg1, msg2), σ), (s, x)] \leftarrow \text{inline hash-oracle}' (A \{ g [^*] x \}) \{ \};

TRY do {
  - :: unit \leftarrow \text{assert-gpv} (valid-plains msg1 msg2);
  cipher \leftarrow \text{aencrypt} (g [^*] x) (if b then msg1 else msg2);
  (guess, s') \leftarrow \text{inline hash-oracle}' (A^2 cipher σ) s;
  Done (guess = b)
} ELSE lift-spmf coin-spmf

by (simp add: split-def game0-def key-gen-def lift-spmf-bind-spmf bind-gpv-assoc try-gpv-bind-lossless [symmetric])

define hash-oracle'' where hash-oracle'' = (λ (s, σ) (x :: 'a), do {
  (h, σ') \leftarrow \text{case σ} \text{ of'}
  None \Rightarrow \text{bind-spmf (spmfm-of-set (nlists UNIV len-plain)) (λ bs. return-spmf (bs, σ(x \mapsto bs)))}
  | Some (bs :: bitstring) \Rightarrow return-spmf (bs, σ);
  return-spmf (h, insert x s, σ')
})

have *: \text{exec-gpv hash.oracle} (inline hash-oracle'' A s) σ =
map-spmf (λ (a, b, c), ((a, b), c)) (exec-gpv hash-oracle'' A (s, σ)) for A σ s
by (simp add: hash-oracle''-def hash-oracle-def 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'' A 1s) s for s x
by (rule lossless-exec-gpv [where \$F = \$F-full]) simp-all

have [simp; lossless-spmf (exec-gpv hash-oracle'' A 2 cipher s) s for cipher s
by (rule lossless-exec-gpv [where \$F = \$F-full]) simp-all

let ?sample = λf. bind-spmf (sample-uniform (order \$F)) (λx. bind-spmf (sample-uniform (order \$F)) (f x))

define game1 where game1 = (λ(x :: nat) (y :: nat), do {
  b \leftarrow \text{coin-spmf};
  (((msg1, msg2), σ), (s, s-h)) \leftarrow \text{exec-gpv hash-oracle'' A 1 (g [^*] x)}) \{ \}, hash.initial;
  TRY do {
    - :: unit \leftarrow \text{assert-spmf} (valid-plains msg1 msg2);
    (h, s-h') \leftarrow \text{hasmap oracle s-h (g [^*] (x * y))};
    let cipher = (g [^*] y, h [^o]) (if b then msg1 else msg2);
    (guess, s', s-h'') \leftarrow \text{exec-gpv hash-oracle'' A 2 cipher σ} (s, s-h');
    return-spmf (guess = b, g [^*] (x * y) \in s')
  } ELSE do {
    b \leftarrow \text{coin-spmf};
    return-spmf (b, g [^*] (x * y) \in s)
  }
})

have game01: \text{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 lossless-inline [where \$F = \$F-full] bind-rpv-def nat-pow-pow del: bind-spmf-const)

including monad-normalisation by (simp add: bind-rpv-def nat-pow-pow)
define game2 where game2 = (λ(x :: nat) (y :: nat). do 
  b ← coin-spmf;
  (((msg1, msg2), σ), (s, s-h)) ← exec-gpv oracle-spmf'' (σf I (g [^] x)) ({}), hash-initial;
  TRY do 
    - :: unit ← assert-spmf (valid-plains msg1 msg2);
  h ← spmf-of-set (nilsets UNIV len-plain);
  — We do not do the lookup in s-h here, so the rest differs only if the adversary guessed y
  let cipher = (g [^] y, h [⊙] (if b then msg1 else msg2));
  (guess, (s', s-h')) ← exec-gpv oracle-spmf'' (σf cipher σ) (s, s-h);
  return-spmf (guess = b, g [^] (x * y) ∈ s')
} ELSE do 
  b ← coin-spmf;
  return-spmf (b, g [^] (x * y) ∈ s)
)
)
)
interpret inv'': callee-invariant-on hash-oracle'' λ(s, s-h). s = dom s-h .¬-full
by unfold-locales(auto simp add: hash-oracle''-def split: option.split_asm if-split)
have in-encrypt-oracle: callee-invariant hash-oracle'' (λ(s, -). x ∈ s) for x
by unfold-locales(auto simp add: hash-oracle''-def)

{ fix x y :: nat
let ?bad = λ(s, s-h), g [^] (x * y) ∈ s
let ?X = λ(s, s-h) (s', s-h'). s = dom s-h ∧ s' = s ∧ s-h = s-h'(g [^] (x * y) := None)
have bisim: 
  rel-spmf (λ(a, s1') (b, s2')). ?bad s1' = ?bad s2' ∧ (∼ ?bad s2' → a = b ∧ ?X s1' s2'))
  (hash-oracle'' s1 plain) (hash-oracle'' s2 plain)
if ?X s1 s2 for s1 s2 plain using that
by(auto split: prod.splits intro!: rel-spmf-bind-refl1 simp add: hash-oracle''-def rel-spmf-return-spmf2
fun-upd-twist split: option.split dest!: fun-upd-eqD)
have inv: callee-invariant hash-oracle'' ?bad
  by(unfold-locales)(auto simp add: hash-oracle''-def split: option.split_asm)
have rel-spmf (λ(win, bad) (win', bad'). bad = bad' ∧ (∼ bad' → win = win'))
(game2 x y) (game1 x y)
unfolding game1-def game2-def
apply(clarsimp simp add: split-def o-def hash-oracle-def rel-spmf-bind-refl1 if-distribs
intro!: rel-spmf-bind-refl1 simp del: bind-spmf-const)
apply(rule rel-spmf-try-spmf)

subgoal for b msg1 msg2 σ s-s-h
apply(rule rel-spmf-bind-refl)
apply(drule inv''.exec-gpv-invariant;clarsimp simp)
apply(cases s-h (g [^] (x * y)))
subgoal — case None
  apply(clarsimp intro!: rel-spmf-bind-refl1)
  apply(rule rel-spmf-bind)
  apply(rule exec-gpv-oracle-bisim-bisim-full[OF - - bisim inv, where R=λ(x, s1) (y, s2). ?bad s1 = ?bad s2 ∧ (∼ ?bad s2 → x = y)]; clarsimp simp add: fun-upd-idem;}
apply clarsimp

done

subgoal by(auto intro!: rel-spmf-bindI1 rel-spmf-bindI2 losiless-exec-gpv[where \(\mathcal{F} = \mathcal{F}\)-full] dest!: callee-invariant-on.exec-gpv-invariant[OF in-encrypt-oracle])
done

subgoal by(rule rel-spmf-reflI) simp
done

hence rel-spmf \((\lambda (\text{win}, \text{bad}) \cdot (\text{win}, \text{bad}')) \cdot (\text{bad} \iff \text{bad}') \cdot (\text{win} \iff \text{win}')\)
by(intro rel-spmf-bind-reflI)
hence measure (measure-spmf \((?\text{sample game2})\) \((?\text{sample game1})\)) \([x, -] \cdot x =\) measure (measure-spmf \((?\text{sample game1})\)) \([y, -] \cdot y \le\) measure (measure-spmf \((?\text{sample game2})\)) \([-, \text{bad}] \cdot \text{bad}

unfolding split-def by(rule fundamental-lemma)
moreover have measure (measure-spmf \((?\text{sample game2})\)) \([x, -] \cdot x =\) spmf (map-spmf \((?\text{sample game2})\)) True
and measure (measure-spmf \((?\text{sample game1})\)) \([y, -] \cdot y =\) spmf (map-spmf \((?\text{sample game2})\)) True
and measure (measure-spmf \((?\text{sample game2})\)) \([-, \text{bad}] \cdot \text{bad} =\) spmf (map-spmf \((?\text{sample game2})\)) True
unfolding spmf-conv-measure-spmf measure-map-spmf by(rule arg-cong2[where \(f =\) measure]; fastforce)+
ultimately have hop23: \(\text{spmf} (\text{map-spmf} \text{fst} (?\text{sample game2})) \text{True} \iff \text{spmf} (\text{map-spmf} \text{fst} (?\text{sample game1})) \text{True}\) by simp

define game3
where game3 = \((\lambda f : \cdot \Rightarrow \Rightarrow \Rightarrow \Rightarrow \Rightarrow \text{bitstring spmf} \Rightarrow \text{spmf}. \lambda (x :: \text{nat}) (y :: \text{nat}). \text{do} \{b \leftarrow \text{coin-spmf}; ((\text{msg}1, \text{msg}2), \sigma), (x, s-h)) \leftarrow \text{exec-gpv hash-oracle}'' (\sigma/I (\text{g} [\cdot] x)) \{\}, \text{hash.initial}\); TRY do \{ - :: \text{unit} \leftarrow \text{assert-spmf} (\text{valid-plains msg1 msg2}); h' \leftarrow f b \text{msg1} \text{msg2} (\text{spmf-of-set} (\text{nlists UNIV len-plain})); let cipher = (\text{g} [\cdot] y, h'); (\text{guess}, (s', s-h')) \leftarrow \text{exec-gpv hash-oracle}'' (\sigma/2 \text{cipher} \sigma) (s, s-h); return-spmf (\text{guess} = b, \text{g} [\cdot] (x * y) \in s') \} \} \} ELSE do \{b \leftarrow \text{coin-spmf}; return-spmf (b, \text{g} [\cdot] (x * y) \in s) \}
\}

let \(?f = \lambda b \text{msg1} \text{msg2}. \text{map-spmf} (\lambda h. \text{if} b \text{then} \text{msg1} \text{else} \text{msg2}) [\oplus] h)\)

have game2 \(x y = \text{game3} \{f x y \text{ for } x y\)
unfolding game2-def game3-def by(simp add: Let-def bind-map-spmf xor-list-commute o-def nat-pow-pow)
also have game3 \(?f x y = \text{game3} (\lambda - - x. x) x y \text{ for } x y\)
unfolding game3-def
finally have game23; game2 x y = game3 (λ - - - x. x) x y for x y.

define hash-oracle"" where hash-oracle"" = (λ(σ :: 'a ⇒ -). hash.oracle σ)
{ define bisim where bisim = (λσ (s :: 'a set, σ' :: 'a → bitstring). s = dom σ ∧ σ = σ')
  have [transfer-rule]: bisim Map-empty ({}), Map-empty by(simp add: bisim-def)
  have [transfer-rule]: (bisim ===> (=) ===> rel-spmf (rel-prod (=) bisim)) hash-oracle""
    hash-oracle""
    by(auto simp add: hash-oracle""-def split-def hash-oracle""-def spmf-rel-map hash-oracle-def
        rel-fun-def bisim-def split; option.split intro!: rel-spmf-bind-refl)
  have * [transfer-rule]: (bisim ===> (=)) dom fst by(simp add: bisim-def rel-fun-def)

  have * [transfer-rule]: (bisim ===> (=)) (λx. x) snd by(simp add: rel-fun-def)
    bisim-def)
  have game3 (λ - - - x. x) x y = do {
    b ← coin-spmf;
    (((msg1, msg2), σ), s) ← exec-gpv hash-oracle"" (λf (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"" (λ2 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(auto simp add: assumption)
moreover have map-spmf snd (... x y) = do {
  zs ← elgamal-adversary (λ (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-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)
by(auto simp add: o-def)
then have game2-snd: map-spmf snd (?sample game2) = lcdh.lcdh (elgamal-adversary)
by(auto)
using game23 by(auto 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"" (λf (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"" (λ2 (g ['^'] y, h') σ) (s, s-h);

39
map-spmf (\(=\) guess) coin-spmf

} ELSE coin-spmf

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 (\(\lambda\) - - \(x\). \(x\)) \(x\) \(y\)) = coin-spmf for \(x\) \(y\)

by (simp add: o-def if-distrib spmf.

map-comp map-eq-const-coin-spmf split-def)

have ind-cpa.advantage hash.oracle hash.initial \(\mathcal{A}\) = \(|\text{spmf } (\text{map-spmf } \text{fst } (?\text{sample game1}))\) \(\text{True} - 1 / 2\)

using game0 by (simp add: ind-cpa-pk.advantage-def game01 o-def)

also have \(\ldots = 1 / 2 = \text{spmf } (\text{map-spmf } \text{fst } (?\text{sample game1}))\) \(\text{True}\)

by (simp add: abs-minus-commute)

also have \(1 / 2 = \text{spmf } (\text{map-spmf } \text{fst } (?\text{sample game2}))\) \(\text{True}\)

by (simp add: game23 o-def game3-fst spmf-of-set)

also note hop23 also note game2-snd

finally show \(?\text{thesis}\) by (simp add: lcdh.advantage-def)

qed

end

category elgamal-base begin

lemma lossless-key-gen [simp]: lossless-spmf key-gen \(\leftrightarrow 0 < \text{order } \mathcal{G}\)

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

lemma lossless-elgamal-adversary:

\[\text{ind-cpa.lossless } \mathcal{A} ; \forall \eta . 0 < \text{order } \mathcal{G}\]

\(\Rightarrow\) lcddh.lossless (elgamal-adversary \(\mathcal{A}\))

by (cases \(\mathcal{A}\)) \(\text{auto simp add: lcddh.lossless-def ind-cpa.lossless-def split-def Let-def intro!}\:

lossless-exec-gpv[where \(\mathcal{I} = \mathcal{I}\text{-full}\) lossless-inline]

end

end

2.3 The random-permutation random-function switching lemma
proof \( (cases\ C = \{} \lor A \cap B = \{\}) \)

case False

define A' where A' \equiv A \cap B

from False have C : C \neq {} and A' : A' \neq {} by (auto simp add: A'\-def)

have B : B = A' \cup C using assms by (auto simp add: A'\-def)

with finB have finA; finite A' and finC; finite C by simp-all

from assms have A C : A' \cap C = {} by (auto simp add: A'\-def)

have bind-spmf (spmf-of-set B) \( \lambda x.\ if\ x \in A\ then \ spmf-of-set\ C\ else\ return-spmf\ x \) =

bind-spmf (spmf-of-set B) \( \lambda x.\ if\ x \in A'\ then \ spmf-of-set\ C\ else\ return-spmf\ x \)

by (rule bind-spmf-cong [OF refl]) (simp add: set-spmf-of-set finB A'\-def)

also have \( \ldots = \ spmf-of-set\ C \) (is ?lhs = ?rhs)

proof (rule spmf-eqI)

fix i

have \( (\sum x \in C.\ spmf (if\ x \in A'\ then \ spmf-of-set\ C\ else\ return-spmf\ x)\ i) = \ indicator\ C\ i \)

using finA finC

by (simp add: disjoint-notin1 [OF A' C] indicator-single-\ Some\ sum-mult-indicator\ [OF C lambda.1 - \ lambda.x.\ simplified] split: split-indicator cong: conj-cong sum.cong)

then show \( \ spmf\ ?lhs\ i = \ spmf\ ?rhs\ i \) using B finA finC A' C A'


qed

finally show \( \thesis \).

qed (use assms in 

\auto 4 3 cong: bind-spmf-cong simp add: subsetD bind-spmf-const spmf-of-set-empty disjoint-notin1 intro!: arg-cong [where f=\ spmf-of-set\] )

locale rp-rf =

rp: random-permutation A +

rf: random-function spmf-of-set A

for A :: 'a set

+ 

assumes finite-A: finite A

and nonempty-A: A \neq {} 

begin

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

definition game :: bool \Rightarrow 'a adversary \Rightarrow bool spmf where

game b A' = run-gpv (if b then rp.random-permutation else rf.random-oracle) A'

abbreviation prp-game :: 'a adversary \Rightarrow bool spmf where prp-game \equiv game True

abbreviation prf-game :: 'a adversary \Rightarrow bool spmf where prf-game \equiv game False

definition advantage :: 'a adversary \Rightarrow real where

advantage A' = \| spmf (prp-game A') True \ - \ spmf (prf-game A') \ True \|

lemma advantage-nonneg: 0 \leq advantage A' by (simp add: advantage-def)

lemma advantage-le-1: advantage A' \leq 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 J .lifting begin

lift-definition J :: ('a, 'a) J I is (λx. if x ∈ A then A else { }) .

lemma outs- J .I [simp]: outs- J .I = A by transfer auto

lemma responses- J .I [simp]: responses- J .I x = (if x ∈ A then A else { }) by transfer simp

lifting-update J .lifting

lifting-forget J .lifting

end

lemma rp-rf:
assumes bound: interaction-any-bounded-by $\mathcal{A}$ q

and lossless: lossless-gpv J $\mathcal{A}$

and WT: $\mathcal{J} + g \cong \mathcal{A}$ /

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

including lifting-syntax

proof –

let $?$run = λb. exec-gpv (if b then rp.random-permutation else rf.random-oracle) $\cong$

Map empty

define rp-bad :: bool × ('a → 'a) ⇒ 'a ⇒ ('a × (bool × ('a → 'a))) spmf

where rp-bad = (λ (bad, σ) x. case σ x of Some y ⇒ return-spmf (y, (bad, σ))

| None ⇒ bind-spmf (spmf-of-set A) (λ y. if y ∈ ran σ then map-spmf (λ y', (True, σ(x ⇒ y')))) (spmf-of-set (A − ran σ)) else return-spmf (y, (bad, (σ(x ⇒ y))))))

have rp-bad-simps: rp-bad (bad, σ) x = (case σ x of Some y ⇒ return-spmf (y, (bad, σ)))

| None ⇒ bind-spmf (spmf-of-set A) (λ y. if y ∈ ran σ then map-spmf (λ y', (True, σ(x ⇒ y')))) (spmf-of-set (A − ran σ)) else return-spmf (y, (bad, (σ(x ⇒ y))))))

for bad σ x by (simp add: rp-bad-def)

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

define init :: bool × ('a → 'a) where init = (False, Map.empty)

have rp: rp.random-permutation = (λ σ x. case σ x of Some y ⇒ return-spmf (y, σ))

| None ⇒ bind-spmf (bind-spmf (spmf-of-set A) (λ y. if y ∈ ran σ then spmf-of-set (A − ran σ) else return-spmf y)) (λ y. return-spmf (y, σ(x ⇒ y))))

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

have [transfer-rule]: (?S ===> (=) ===> rel-spmf (rel-prod (=) ?S)) rp.random-permutation

rp-bad

unfolding rp rp-bad-def

by (auto simp add: rel-fun-def map-spmf-conv-bind-spmf split: option.split intro!: rel-spmf-bind-refl)

have [transfer-rule]: ?S Map.empty init by (simp add: init-def)

have spmf (prp-game $\mathcal{A}$') True = spmf (run-gpv rp-bad $\cong$ init) True

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

moreover |

define collision :: ('a → 'a) ⇒ bool where collision m ✖️→ ¬ inj-on m (dom m) for m

have [simp]: ¬ collision Map.empty by (simp add: collision-def)

have [simp]: collision m; m x = None] ✖️→ collision (m(x := y)) for m x y

by (auto simp add: collision-def fun-upd-ideom dom-minus fun-upd-image dest: inj-on-funupdD)
have collision-map-updI: \[ m \times = \text{None}; \ y \in \text{ran} \ m \ \Longrightarrow\ \text{collision} (m(x \mapsto y)) \] for \( m \times y \)
by(auto simp add: collision-def ran-def intro: rev-image-eqI)

have collision-map-upd-iff: \[ \sim\text{collision} \ m \ \Longrightarrow\ \text{collision} (m(x \mapsto y)) \ \iff\ y \in \text{ran} \ m \ \land \ m \times \neq \text{Some} \ y \] for \( m \times y \)
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 \(?b1\) = collision and \(?b2\) = \(\text{fst}\)
and \(\exists X = \lambda \sigma^1 (\text{bad}, \ \sigma^2). \ \sigma^1 = \sigma^2 \land \sim\text{collision} \ \sigma^1 \land \sim \text{bad}\)
and \(?I^1 = \lambda \sigma^1. \ \text{dom} \ \sigma^1 \subseteq A \land \text{ran} \ \sigma^1 \subseteq A\)
and \(?I^2 = \lambda (\text{bad}, \ \sigma^2). \ \text{dom} \ \sigma^2 \subseteq A \land \text{ran} \ \sigma^2 \subseteq A\)
let \(?X\text{-bad} = \lambda \sigma^1 2. \ ?I^1 \sigma^1 \land \ ?I^2 \sigma^2\)

have [simp]: \(\exists c \ \text{rf. random-oracle} \ s1 \ \vee\ \text{if} \ \text{ran} \ s1 \subseteq A \ \text{for} \ s1\) using that

have [simp]: callee-invariant-on rf.random-oracle \(?I\) \(\mathcal{I}\)
by(unfold-locales)(auto simp add: rf.random-oracle-def finite-A split: option.split-asym)

then interpret rf: callee-invariant-on rf.random-oracle \(?I\) \(\mathcal{I}\).

have [simp]: \(\exists c \ \text{rp-bad} \ s2 \ \vee\ \text{if} \ \text{ran} \ (snd \ s2) \subseteq A \ \text{for} \ s2\) using that
by(intro WT-calleeI)(auto simp add: rp-bad-def finite-A split: prod.split-asym option.split-asym if-split-asym intro: ranI)

have [simp]: callee-invariant-on rf.random-oracle \(\lambda \sigma^1. \ ?b1 \ \sigma^1 \ ?I^1 \sigma^1\) \(\mathcal{I}\)
by(unfold-locales)(clarsimp simp add: rf.random-oracle-def finite-A split: option.split-asym)+

have [simp]: callee-invariant-on rp-bad \(\lambda s2. \ ?I^2 \ s2\) \(\mathcal{I}\).

by(unfold-locales)(auto 4 3 simp add: rp-bad-simps finite-A split: option.splits if-split-asym
iff del: domIff)

have [simp]: \(\exists c \ \text{rp-bad} \ (\text{bad}, \ \sigma^2) \ \vee\ \text{if} \ \text{ran} \ \sigma^2 \subseteq A \ \text{for} \ \text{bad} \ \sigma^2\) using that
by(intro WT-calleeI)(auto simp add: rp-bad-def finite-A nonempty-A ran-def split: option.split-asym if-split-asym)

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 \(\lambda (b_1, \ \sigma^1) (b_2, \ \text{state}2). \ (?b1 \ \sigma^1 \leftarrow \ ?b2 \ \text{state}2) \ \land\ (if \ ?b2 \ \text{state}2 \ then \ ?X-bad \ ?I^1 \ \text{state}2 \ else \ b1 = b2 \ \land \ ?X \ ?I^1 \ \text{state}2))\)
(if False then rp-random-permutation else rf.random-oracle) \s1 \ x\) \(\text{rp-bad} \ s2 \ x\)
if \(?X \ s1 \ s2 \ x \in\ \text{outs-} \ \mathcal{I}\ \ ?I^1 \ s1 \ ?I^2 \ s2\) for \(s1 \ s2 \ x\) using that finite-A
by(auto split!: option.split simp add: rf.random-oracle-def rp-bad-def rel-spmf-return-spmf1 collision-map-updI dom-subset-ran-iff eq-None-iff-not-dom collision-map-upd-iff intro!: rel-spmf-bind-refl)

with - - have rel-spmf
\(\lambda (b_1, \ \sigma^1) (b_2, \ \text{state}2). \ (?b1 \ \sigma^1 \leftarrow \ ?b2 \ \text{state}2) \ \land\ (if \ ?b2 \ \text{state}2 \ then \ ?X-bad \ \sigma^1 \ \text{state}2 \ else \ b1 = b2 \ \land \ ?X \ \sigma^1 \ \text{state}2))\)
(?run False) (exec-gpv rp-bad \$ init)
by(rule exec-gpv-oracle-bism-bad-invariant[where \(\mathcal{I} = \mathcal{I}\) and \(?I.0 = ?I\) and
theory PRF-UHF imports
CryptHOL.GPV-Bisim
Pseudo-Random-Function

begin

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

begin

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

definition game-hash-set :: 'domain set ⇒ bool spmf  
where

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

This example is taken from [19, §4.2].
game-hash-set W = do 
  seed ← seed-gen;
  return-spmf (¬ inj-on (hash seed) W)
}

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

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

lemma hash-ineq-card:
assumes finite W
shows spmf (game-hash-set W) True ≤ ε-uh * card W * card W
proof
  - let ?M = measure (measure-spmf seed-gen)
  have bound: ?M {x. hash x w = hash x w' ∧ w ≠ w'} ≤ ε-uh for w w'
  proof
    - have ?M {x. hash x w = hash x w' ∧ w ≠ w'} = spmf (game-hash w w') True
    also have ... ≤ ε-uh unfolding ε-uh-def
      by(auto intro!: cSUP-upper2 bdd-aboveI[where M=1] cSUP-least simp add: pmf-le-1)
    finally show ?thesis .
  qed

have spmf (game-hash-set W) True = ?M {x. ∃xa∈W. ∃y∈W. hash x xa = hash x y ∧ xa ≠ y}
also have {x. ∃xa∈W. ∃y∈W. hash x xa = hash x y ∧ xa ≠ y} = (∪ {w, w'} ∈ W × W. {x. hash x w = hash x w' ∧ w ≠ w'})
  by(auto)
also have ... ≤ (∑ {w, w'} ∈ W × W. ?M {x. hash x w = hash x w' ∧ w ≠ w'})
  by(auto intro!: measure-spmf finite-measure-subadditive-finite simp add: split-def assms)
also have ... ≤ (∑ {w, w'} ∈ W × W. ε-uh) by(rule sum-mono)(clarsimp simp add: bound)
also have ... = ε-uh * card(W) * card(W) by(simp add: card-cartesian-product)
finally show ?thesis .
  qed

end

locale prf-hash =
fixes f :: 'key ⇒ 'α ⇒ 'γ
and h :: 'seed ⇒ 'β ⇒ 'α
and key-gen :: 'key spmf
and seed-gen :: 'seed spmf
and \( \text{range-f} :: \gamma \text{ set} \)

assumes \( \text{lossless-seed-gen; lossless-spmf seed-gen} \)
and \( \text{range-f-finite; finite range-f} \)
and \( \text{range-f-nonempty: range-f} \neq \{\} \)

begin

definition \( \text{rand} :: \gamma \text{ spmf} \)
where \( \text{rand} = \text{spmf-of-set range-f} \)

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

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

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

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

interpretation \( \text{prf': prf key-seed-gen f' rand} \).

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

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

theorem \( \text{prf-prf'}-\text{advantage} : \)
assumes \( \text{prf'.lossless} \sigma \)
and \( \text{bounded: prf'.ibounded-by} \sigma q \)
shows \( \text{prf'.advantage} \sigma \leq \text{prf'.advantage} (\text{prf'}-\text{reduction} \sigma) + \text{hash.t-uh} * q * q \)
including lifting-syntax
proof =
let \( \sigma' = \text{prf'}-\text{reduction} \sigma \)
{ define \( \text{cr where} \) \( \text{cr} = (\lambda\cdot :: \text{unit} \times \text{unit}. \lambda\cdot :: \text{unit}. \text{True}) \)
have [transfer-rule]: \( \text{cr} (()), (()) () \) by (simp add: cr-def)
have prf.game-0 ? σ′ = prf′.game-0 σ
unfolding prf′.game-0-def prf.game-0-def prf′-reduction-def unfolding key-seed-gen-def
by(simp add: exec-gpv-bind split-def exec-gpv-inline reduction-oracle-def bind-map-spmf
prf.prf-oracle-def prf′.prf-oracle-def[abs-def])
(transfer-prover )

note hop1 = this[symmetric]
def semi-forgetful-RO where semi-forgetful-RO = (λ.seed :: 'seed. λ(σ :: 'α ⇒ 'β × 'γ, b :: bool). λ.x.
case σ (h seed x) of Some (a, y) ⇒ return-spmf (y, (σ, a ≠ x ∨ b))
| None ⇒ bind-spmf rand (λ.y. return-spmf (y, (σ(h seed x) → (x, y)), b))))
def game-semi-forgetful where game-semi-forgetful = do {
seed :: 'seed ← seed-gen;
(b, rep) ← exec-gpv (semi-forgetful-RO seed) σ (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: option.split)

{ define cr
  where cr = (λ. :: unit, σ) (σ′ :: 'α ⇒ ('β × 'γ) option, - :: bool). σ = map-option
snd α σ′)
def initial where initial = (Map.empty :: 'α ⇒ ('β × 'γ) option, False)

have [transfer-rule]: cr (((), Map.empty) initial) by(simp add: cr-def initial-def fun-eq-iff)

have [transfer-rule]: (((=) ===> cr ===> (=) ===> rel-spmf (rel-prod (=) cr))
(λ'y p y a. 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 ? σ′ = map-spmf fst game-semi-forgetful
unfolding prf.game-1-def prf′-reduction-def game-semi-forgetful-def
by(simp add: exec-gpv-bind exec-gpv-inline split-def bind-map-spmf map-spmf-bind-spmf
o-def map-spmf-conv-bind-spmf reduction-oracle-def initial-def [symmetric])
(transfer-prover )

note hop2 = this

def game-semi-forgetful-bad where game-semi-forgetful-bad = do {
seed :: 'seed ← seed-gen;
x ← exec-gpv (semi-forgetful-RO seed) σ (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
\[
\begin{align*}
\text{by}(\text{simp add: map-spmf-bind-spmf o-def})
\end{align*}
\]

**have** bad-random-oracle-\(\lambda\)[simp]; callee-invariant prf.random-oracle (\(\lambda \sigma. \sim \text{inj-on}(h seed)(\text{dom } \sigma)\)) for seed

by unfold-locales(auto simp add: prf.random-oracle-def split: option.split_asm)

**define** invar

where invar = (\(\lambda.\text{seed}(\sigma_1, b) (\sigma_2 \sim \beta \Rightarrow \gamma \text{ option}). \sim b \land \text{dom } \sigma_1 = h \text{ seed} \sim \text{dom } \sigma_2 \land \) \(\forall x \in \text{dom } \sigma_2. \sigma_1(h \text{ seed } x) = \text{map-option(Pair } x) (\sigma_2 x))\)

**have** rel-spmf-oracle-adv:
rel-spmf (\(\lambda(x, s1)(y, s2). \text{snd } s1 \neq \text{inj-on } (h \text{ seed})(\text{dom } s2) \land (\text{inj-on } (h \text{ seed})(\text{dom } s2) \rightarrow x = y \land \text{invar seed } s1 s2)\))

\(\text{exec-gpv (semi-forgetful-RO seed) } \sigma' (\text{Map.empty, False})\)

\(\text{exec-gpv prf.random-oracle } \sigma' \text{ Map.empty}\)

**if** seed: seed \(\in\) set-spmf seed-gen for seed

**proof**

**have** invar-initial [simp]; invar seed (\(\text{Map.empty, False}\)) Map.empty by(simp add: invar-def)

**have** invarD-inj: inj-on (h seed) (dom s2) if invar seed hs1 s2 for hs1 s2

**using** that by(auto intro!: inj-onI simp add: invar-def)(metis domI domIff option.map-sel prod.inject)

**let** ?R = \(\lambda(a, s1)(b, s2 \sim \beta \Rightarrow \gamma \text{ option}). \text{snd } s1 = (\sim \text{inj-on } (h \text{ seed})(\text{dom } s2)) \land (\sim \text{inj-on } (h \text{ seed})(\text{dom } s2) \rightarrow a = b \land \text{invar seed } s1 s2)\)

**have** step: rel-spmf ?R (semi-forgetful-RO seed \(\sigma_1b x) (\text{prf.random-oracle } s2 x)\)

if X: invar seed \(\sigma_1b s2 \text{ for } s2 \sigma_1b x\)

**proof**

**obtain** \(\sigma_1 b \text{ where } [\text{simp}]: \sigma_1b = (\sigma_1, b) \text{ by(cases } \sigma_1b)\)

from X have not-b: \(\sim b\)

and eq: \(\forall x \in \text{dom } s2. \sigma_1(h \text{ seed } x) = \text{map-option(Pair } x) (s2 x)\)

by(simp-all add: invar-def)

from X have inj: inj-on (h seed) (dom s2) by(rule invarD-inj)

**have** not-in-image: h seed x \(\notin\) h seed \(\sim\) (dom s2 \(\sim\) \{x\} ) if \(\sigma_1(h \text{ seed } x) = \text{None}\)

**proof** (rule notI)

**assume** h seed x \(\in\) h seed \(\sim\) (dom s2 \(\sim\) \{x\})

then obtain y where y \(\in\) dom s2 and hx-hy: h seed x \(=\) h seed y by (auto)

then have \(\sigma_1(h \text{ seed } y) = \text{None}\) using that by (auto)

then have h seed y \(\notin\) h seed \(\sim\) (dom s2 using dom by (auto)

then have y \(\notin\) dom s2 by (auto)

then show False using \(\forall y \in \text{dom } s2. \text{by(auto)}\)

qed

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

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

let ?bad1 = λx. snd (snd x) and ?bad2 = snd
have hop3: rel-spmf (λx xa. (?bad1 x ↦→ ?bad2 xa) ∧ (¬ ?bad2 xa → fst x ↦→ fst xa)) game-semi-forgetful game-A
  unfolding game-semi-forgetful-def game-A-def
by(clarsimp simp add: restrict-bind-spmf split-def map-spmf-bind-spmf restrict-return-spmf
  o-def intro!: rel-spmf-bind-reflI simp del: bind-return-spmf)
(rule rel-spmf-bindI[OF rel-spmf-oracle-adv]; auto)

have bad1-bad2: "spmf (map-spmf (snd o snd) game-semi-forgetful) True = spmf (map-spmf snd game-A) True"
  vimage-def)

have bound-bad1-event: "spmf (map-spmf fst game-semi-forgetful) True - spmf (map-spmf
  fst game-A) True \leq spmf (map-spmf (snd o 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 snd game-A) True |\leq spmf (map-spmf snd game-A) True"
  using bad1-bad2 by (simp)

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

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

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

have hop4: "prf`.game-1 Σ = 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 Σ \leq |spmf (prf`.game-0 \notΑ\not) True - spmf (prf`.game-1 Σ) True|
  using hop1 by (simp add: prf`.advantage-def)
also have \ldots \leq prf`.advantage Σ + |spmf (prf`.game-1 \notΑ\not) True - spmf (prf`.game-1
  Σ) True|
  by(simp add: prf`.advantage-def)
also have $|\text{spmf} (\text{prf} .\text{game-1} \ ? ?) \ True − \text{spmf} (\text{prf} .\text{game-1} \ ? ?) \ True| ≤$

$|\text{spmf} (\text{map-spmf fst game-semi-forgetful}) \ True − \text{spmf} (\text{prf} .\text{game-1} \ ? ?) \ True|$

\text{using hop2 by simp}

also have $\ldots ≤ \text{hash} \cdot \text{uh} * q * q$

\text{using game-A-game-B game-B-bound bound-bad2-event hop by simp)}

\text{finally show } \text{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: $\forall xs ys. \ [ \text{length } xs = \text{len}; \text{length } ys = \text{len} \ ] \implies \text{length } (f \ xs \ ys) = \text{len}$

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 $\leftrightarrow \text{length } plain = \text{len}$

definition encrypt :: key ⇒ plain ⇒ cipher spmf

where encrypt key plain = do

  r $\leftarrow$ 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)

end

end
lemma encrypt-decrypt-correct:
\[ \text{length key} = \text{len}; \text{length plain} = \text{len} \]
\[ \Rightarrow \text{encrypt key plain} \gg (\lambda \text{cipher}. \text{return-spmf (decrypt key cipher)}) = \text{return-spmf} \]
(Some plain)
by (simp add: encrypt-def zip-map2 o-def split-def 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 
  \( r \leftarrow \text{lift-spmf (spmf-of-set (nlists UNIV len))}; \)
  \( \text{Pause } r (\lambda \text{pad}. \text{Done } ((r, \text{xor-list plain pad}), ())) \)

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 \( \alpha \) = do 
  let \( \alpha' = \alpha \);
  ((p1, p2), n) \leftarrow \text{inline prf-encrypt-oracle } \alpha 1 ;
  \text{if valid-plain p1 \land valid-plain p2 then do }
  \( b \leftarrow \text{lift-spmf coin-spmf}; \)
  \( \text{let pb} = (\text{if } b \text{ then p1 else p2}); \)
  \( \text{let } c = (r, \text{xor-list pb pad}); \)
  \( \text{let } c' = b \)
  \text{Done (b' = b)}
  \} \text{else lift-spmf coin-spmf}

theorem prf-encrypt-advantage:
assumes ind-cpa.bounded-by \( \alpha \) q
and lossless-gpv \( \mathcal{I} \)-full (fst \( \alpha \))
and \( \text{cipher } \sigma \). lossless-gpv \( \mathcal{I} \)-full (snd \( \alpha \) cipher \( \sigma \))
shows ind-cpa.advantage \( \alpha \leq \text{prf.advantage (prf-adversary } \alpha \) + q / 2 ^ \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-refI rel-spmf-refII
obtain $s'$ where $s' = (s' 1, s' 2)$ by (cases $s'$)
from ind-cpa.ibounded-by →
obtain $q 1\ q 2 :: \text{nat}$
where
$q 1 :: \text{interaction-any-bounded-by } s' 1 q 1$
and $q 2 :: \text{interaction-any-bounded-by } (s' 2 \ \text{cipher } \sigma) q 2$
and $q 1 + q 2 \leq q$
unfolding $s'$ by (rule ind-cpa.ibounded-byE) (auto simp: iadd-le-enat-iff)
from $s'$ asms have lossless1: lossless-gpv $\neg - \text{full } s' 1$
and lossless2: $\forall \text{cipher } \sigma. \text{lossless-gpv } \neg - \text{full } (s' 2 \ \text{cipher } \sigma) \text{ by simp-all}$
have weight1: $\forall \text{oracle } s. (\forall x. \text{lossless-spmf } (\text{oracle } s x))$
\[ \implies \text{weight-spmf } (\text{exec-gpv oracle } s' 1 s) = 1 \]
by (rule lossless-weight-spmfD) (rule lossless-exec-gpv [OF lossless1], simp-all)
have weight2: $\forall \text{oracle } s \ \text{cipher } \sigma. (\forall x. \text{lossless-spmf } (\text{oracle } s x))$
\[ \implies \text{weight-spmf } (\text{exec-gpv oracle } (s' 2 \ \text{cipher } \sigma) s) = 1 \]
by (rule lossless-weight-spmfD) (rule lossless-exec-gpv [OF lossless2], simp-all)

let $\text{oracle } 1 = \lambda \text{key } (s', y). \text{map-spmf } (\lambda ((x, s'), x, (), ()). (\text{exec-gpv } (\text{prf.prf-oracle } key) (\text{prf-encrypt-oracle } (\tau y) (\tau)))$

have bisim1: $\forall \text{key } \text{rel-spmf } (\lambda (x, -) (y, -), x = y)$
\[ (\text{exec-gpv } (\text{ind-cpa.encrypt-oracle-key}) s' 1 ()) \]
\[ (\text{exec-gpv } (\text{oracle } 1 \text{key}) s' 1 ((), ())) \]
using TrueI
by (rule exec-gpv-oracle-bisim) (auto simp: encrypt-def prf-encrypt-oracle-def ind-cpa.encrypt-oracle-def prf.prf-oracle-def o-def)

have bisim2: $\forall \text{key } \text{cipher } \sigma. \text{rel-spmf } (\lambda (x, -) (y, -), x = y)$
\[ (\text{exec-gpv } (\text{ind-cpa.encrypt-oracle-key}) (s' 2 \ \text{cipher } \sigma) ())) \]
\[ (\text{exec-gpv } (\text{oracle } 1 \text{key}) (s' 2 \ \text{cipher } \sigma) ((), ())) \]
using TrueI
by (rule exec-gpv-oracle-bisim) (auto simp: encrypt-def prf-encrypt-oracle-def ind-cpa.encrypt-oracle-def prf.prf-oracle-def o-def)

have ind-cpa-0: $\text{rel-spmf } (=) (\text{ind-cpa.ind-cpa } s' )$ (prf.game-0 (prf.adversary $s'$))

unfolding IND-CPA.ind-cpa.ind-cpa-def $s'$ key-gen-def Let-def prf-adversary-def Pseudo-Random-Function.prf.game
apply (simp)
apply (rewrite in bind-spmf - $\Pi$ bind-commute-spmf)
apply (rule rel-spmf-bind-refI)
apply (rule rel-spmf-bindI [OF bisim1])
apply (clarsimp simp add: if-distrib bind-coin-spmf-eq-const)
done

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

\textbf{interpret} \ rf\text{-}finite: \ \text{callee-invariant-on} \ rf\text{-}encrypt \ \lambda s. \ \text{finite} \ (\text{dom} \ s) \ \not\emptyset \text{-full} \\
\text{by unfold-locales(auto simp add: rf\text{-}encrypt-def dest: rf\text{-}finite.caller\text{-}invariant) \\
\text{have} \ lossless-rf\text{-}encrypt \ \text{simp}; \ \set{s}{\text{plain}}. \ \text{lossless-spmf} \ (\text{rf\text{-}encrypt} \ s \ \text{plain}) \\
\text{by(auto simp add: rf\text{-}encrypt-def) \\
\text{define} \ \text{game2} \ \text{where} \ \text{game2} \ = \ \text{do} \ \\
\ ((\{(p0, \ p1), \ (\sigma), \ s1\}) \leftarrow \ \text{exec-gpv} \ \text{rf\text{-}encrypt} \ \not\emptyset \ \text{Map}\.\text{empty}; \\
\text{if} \ \text{valid-plain} \ p0 \ \wedge \ \text{valid-plain} \ p1 \ \text{then} \ do \ \\
\ b \leftarrow \ \text{coin\text{-}spmf}; \\
\ \text{let} \ \text{pb} = \ (\text{if} \ b \ \text{then} \ p0 \ \text{else} \ p1); \\
\ (\text{cipher}, \ s2) \leftarrow \ \text{rf\text{-}encrypt} \ s1 \ \text{pb}; \\
\ (b', \ s3) \leftarrow \ \text{exec-gpv} \ \text{rf\text{-}encrypt} \ (\not\emptyset \ 2 \ \text{cipher} \ \sigma) \ s2; \\
\ \text{return-spmf} \ ((b' = b)) \\
\text{else} \ \text{coin\text{-}spmf} \\
\text{)} \\
\text{let} \ \text{oracle2} = \ \lambda (s', \ y). \ \text{map\text{-}spmf} \ (\lambda (x, \ s'). \ (x, \ (), \ s)) \ (\text{exec-gpv} \ \text{prf.random-oracle} \ (\text{prf\text{-}encrypt-oracle} (\lambda (y) \ s)) \\
\text{let} \ \text{I} = \ \lambda (s, \ s'). \ (s', \ y, \ x = y \ \wedge \ s = s') \\
\text{have} \ \text{bisim1: rel-spmf} \ ?I \ (\text{exec-gpv} \ \text{oracle2} \ \not\emptyset \ 1 \ (() , \ \text{Map}\.\text{empty})) \ (\text{exec-gpv} \ \text{rf\text{-}encrypt} \ \not\emptyset \ 1 \ \text{Map}\.\text{empty}) \\
\text{by}(\text{rule exec-gpv-oracle-bisim[where} \ X=\lambda (-, \ s) \ s' . \ s = s')\text{)} \\
\quad \text{(auto simp add: rf\text{-}encrypt-def prf\text{-}encrypt-oracle-def intro!: rel-spmf\text{-}bind\text{-}reflI)} \\
\text{have} \ \text{bisim2: rel-spmf} \ ?I \ (\text{exec-gpv} \ \text{oracle2} \ (\not\emptyset \ 2 \ \text{cipher} \ \sigma) \ (() , \ s)) \ (\text{exec-gpv} \ \text{rf\text{-}encrypt} \ (\not\emptyset \ 2 \ \text{cipher} \ \sigma) \ s) \\
\text{by}(\text{rule exec-gpv-oracle-bisim[where} \ X=\lambda (-, \ s) \ s' . \ s = s')\text{)} \\
\quad \text{(auto simp add: prf\text{-}encrypt-oracle-def rf\text{-}encrypt-def intro!: rel-spmf\text{-}bind\text{-}reflI)} \\
\text{have} \ \text{game1-2 [unfolded spmf\text{-}rel\text{-}eq]:} \ \text{rel-spmf} \ (\not\emptyset) \ (\text{prf.game1-1 (prf\text{-}adversary} \ \not\emptyset)) \ \text{game2} \\
\text{unfolding} \ \text{prf.game1-def game2-def prf\text{-}adversary-def} \\
\text{by(rewrite in if - then \ \not\emptyset \ else - rf\text{-}encrypt-def)} \\
\text{(auto simp add: Let-def \ \not\emptyset \ if\text{-}distribs intro!: rel-spmf\text{-}bind\text{-}I[OF \ bisim2] rel-spmf\text{-}bind\text{-}reflI \ rel-spmf\text{-}bind\text{-}I[OF \ bisimI])} \\
\text{define} \ \text{game2-2-a} \ \text{where} \ \text{game2-2-a} \ = \ \text{do} \ \\
\quad r \leftarrow \ \text{spmf\text{-}of\text{-}set} \ (\text{nlists} \ \text{UNIV} \ \text{len}); \\
\quad ((\{(p0, \ p1), \ (\sigma), \ s1\}) \leftarrow \ \text{exec-gpv} \ \text{rf\text{-}encrypt} \ \not\emptyset \ \text{Map}\.\text{empty}; \\
\text{let} \ \text{bad} = r \in \ \text{dom} \ s1; \\
\text{if} \ \text{valid-plain} \ p0 \ \wedge \ \text{valid-plain} \ p1 \ \text{then} \ do \ \\
\ b \leftarrow \ \text{coin\text{-}spmf}; \\
\ \text{let} \ \text{pb} = \ (\text{if} \ b \ \text{then} \ p0 \ \text{else} \ p1); \\
\ (\text{pad}, \ s2) \leftarrow \ \text{prf.random-oracle} \ s1 \ r; \\
\ \text{let} \ \text{cipher} = \ (r, \ \text{xor-list} \ \text{pb} \ \text{pad}); \\
\ (b', \ s3) \leftarrow \ \text{exec-gpv} \ \text{rf\text{-}encrypt} \ (\not\emptyset \ 2 \ \text{cipher} \ \sigma) \ s2; \\
\ \text{return-spmf} \ ((b' = b, \ \text{bad}) \\
\text{)} \ \text{else} \ \text{coin\text{-}spmf} \gg= \ (\lambda b. \ \text{return-spmf} \ (b, \ \text{bad})) \\
\)}
define game2-b where game2-b = do {
    r ← spmf-of-set (nlists UNIV len);
    ((p0, p1), σ), s1) ← exec-gpv rf-encrypt sτI Map.empty;
    let bad = r ∈ dom s1;
    if valid-plain p0 ∧ valid-plain p1 then do {
        b ← coin-spmf;
        let pad = (if b then p0 else p1);
        (b', s3) ← exec-gpv rf-encrypt (sτ/2 cipher σ) (s1(r → pad));
        return-spmf (b' = b, bad)
    } else coin-spmf >>= (λb. return-spmf (b, bad))
}

have game2 = do {
    r ← spmf-of-set (nlists UNIV len);
    ((p0, p1), σ), s1) ← exec-gpv rf-encrypt sτI Map.empty;
    if valid-plain p0 ∧ valid-plain p1 then do {
        b ← coin-spmf;
        let pad = (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 (sτ/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-distribR if-distrib split-def cong: if-cong)
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-distribR 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-map-id[unfolded id-def] spmf-map-restrict[output] simp)
**define** game2-a-bad **where** game2-a-bad = do {
  r ← spmf-of-set (nlists UNIV len);
  (((p0, p1), σ), s1) ← exec-gpv rf-encrypt σ I Map.empty;
  return-spmf (r ∈ dom s1)
}

**have** game2a-bad; map-spmf snd game2-a = game2-a-bad

**unfolding** game2-a-def game2-a-bad-def
  **by** (auto intro!: bind-spmf-cong[OF refl] simp add: o-def weight2 Let-def split-def split: if-split)

**have** card: ∀B :: bool list set. card (B ∩ nlists UNIV len) ≤ card (nlists UNIV len :: bool list set)
  **by** (rule card-mono) simp-all

**then have** spmf game2-a-bad True = \( ∫^+ x. card (dom (snd x) ∩ nlists UNIV len) / 2^{^n} \)
  **unfolding** game2-a-bad-def

**also** (fix x s)
  **assume** *: (x, s) ∈ set-spmf (exec-gpv rf-encrypt σ I Map.empty)
  **hence** finite (dom s) **by** (rule rf-finite-exec-gpv-invariant) simp-all
  **hence** 1: card (dom s ∩ nlists UNIV len) ≤ card (dom s) **by** (intro card-mono) simp-all

**moreover from** q1 *
  **have** card (dom s) ≤ q1 + card (dom (Map.empty :: (plain, plain) prf.dict))
    **by** (rule rf-finite.interaction-bounded-by1-exec-gpv-count)
    (auto simp add: rf-encrypt-def eSuc-enat prf.random-oracle-def card-insert-if split: option.split-asm if-split)

  **ultimately have** card (dom s ∩ nlists UNIV len) ≤ q1 **by** (simp)

  **then have** \( \ldots \leq \frac{q_1}{2^{^n}} \)
  **by** (intro nn-integral-mono-AE) (clarsimp simp add: field-simps)

  **also have** \( \ldots \leq q_1 / 2^{^n} \)
  **by** (simp add: measure-spmf.emeasure-eq-measure field-simps mult-left-le weight1)

**finally have** game2a-bad-bound: spmf game2-a-bad True \( \leq q_1 / 2^{^n} \)

**by** simp

**define** rf-encrypt-bad
  **where** rf-encrypt-bad = (λs. secsecret (s :: (plain, plain) prf.dict, bad) plain. bind-spmf
  (spmf-of-set (nlists UNIV len)) (λr.
    bind-spmf (prf.random-oracle s r) (λa. (pad, s')).
    return-spmf ((r, xor-list plain pad), (s', bad ∨ r = secret))))

**have** rf-encrypt-bad-sticky [simp]; ∀s. callee-invariant (rf-encrypt-bad s) snd
  **by** (unfold-locales (auto simp add: rf-encrypt-bad-def)

**have** lossless-rf-encrypt [simp]; ∀challenge s plain. lossless-spmf (rf-encrypt-bad challenge s plain)
  **by** (clarsimp simp add: rf-encrypt-bad-def-prf.random-oracle-def split: option.split)

**define** game2-c **where** game2-c = do {
  r ← spmf-of-set (nlists UNIV len);
  (((p0, p1), σ), s1) ← exec-gpv rf-encrypt σ I 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 x) (σ2 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 (σ2 cipher σ) (s(x ↦ r)))
(exec-gpv (rf-encrypt-bad x) (σ2 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 game2c-bad [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$s$ intro!: rel-spmf-bind-refl [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 σ₁ 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 x) (σ2 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 λs1 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) (σ2 cipher σ) (s(x ↦ r), False))
  (exec-gpv (rf-encrypt-bad x) (σ2 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-reflI rel-spmf-bindI OF_bisim-bad)

moreover
have rel-spmf \( \equiv \) (map-spmf fst (game2-c \{ snd \rightarrow \{ \text{'False'} \} \})) (map-spmf fst (game2-d \{ snd \rightarrow \{ \text{'False'} \} \}))

unfolding game2-c-def game2-d-def
by (clarsimp simp add: restrict-bind-spmf o-def Let-def if-distrib restrict-return-spmf intro!: rel-spmf-bind-reflI rel-spmf-bindI OF_bisim-bad)

hence spmf game2-c (True, False) = spmf game2-d (True, False)

unfolding spmf-rel.eq by (subst (1 2) spmf-map-restrict[symmetric] simp)

ultimately have game2-c-2d: \( \text{spmf} \text{(map-spmf fst game2-c)} \text{True} - \text{spmf} \text{(map-spmf fst game2-d)} \text{True} \leq \text{spmf} \text{(map-spmf snd game2-c)} \text{True} \)

apply (subst (1 2) spmf-map-conv-spmf)
apply (intro identical-until-bad)
apply (simp-all add: spmf.map-id[unfolded id-def] spmf-map-conv-spmf)
done

{ fix cipher \( \sigma \) and challenge :: plain and s
have card (nlists UNIV len \( \cap \) (\( \lambda x. x = \text{challenge} \) \( \Rightarrow \) \{True\})) \leq card \{challenge\}
by (rule card-mono) auto
then have spmf (map-spmf (snd \( \circ \) snd)) (exec-gpv (rf-encrypt-bad challenge) (\( \sigma \)/2 cipher \( \sigma \)) (s, False))) \( \leq \) \( (1 / 2 \wedge \text{len}) * q2 \)

hence \( (\lambda x. \text{ennreal} \text{(indicator } \{ \text{True} \} \times x) \partial \text{measure-spmf} \text{(map-spmf (snd \( \circ \) snd)) (exec-gpv (rf-encrypt-bad challenge) (\( \sigma \)/2 cipher \( \sigma \)) (s, False)))} \leq \( (1 / 2 \wedge \text{len}) * q2 \)
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) \( \text{True} \) \( \leq \) \( \text{ennreal} \text{(indicator } \{ \text{True} \} \times x) \partial \text{measure-spmf} \text{(map-spmf (snd \( \circ \) snd)) (exec-gpv (rf-encrypt-bad challenge) (\( \sigma \)/2 cipher \( \sigma \)) (s, False)))} \leq \( (1 / 2 \wedge \text{len}) * q2 \)
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) \( \text{True} \) \( \leq \) \( \text{ennreal} \text{(indicator } \{ \text{True} \} \times x) \partial \text{measure-spmf} \text{(map-spmf (snd \( \circ \) snd)) (exec-gpv (rf-encrypt-bad challenge) (\( \sigma \)/2 cipher \( \sigma \)) (s, False)))} \leq \( (1 / 2 \wedge \text{len}) * q2 \)
by (simp only: ennreal-indicator nn-integral-indicator sets-measure-spmf sets-count-space Pow-UNIV UNIV-I emeasure-spmf-single) simp 

unfolding game2-d-def
by (simp add: ennreal-spmf-bind o-def split-def Let-def if-distrib if-distrib where f = \( \lambda x. \text{ennreal} \text{(spmf x -)}\) indicator-single SOME nn-integral-mono if-mono-cong del: nn-integral-const cong: if-cong)
also have \( \ldots \leq \text{ennreal} \text{(indicator } \{ \text{True} \} \times x) \partial \text{measure-spmf} \text{(exec-gpv (rf-encrypt-bad challenge) (\( \sigma \)/2 cipher \( \sigma \)) (s, False)))} \leq \( (1 / 2 \wedge \text{len}) * q2 \)
by (simp only: ennreal-indicator nn-integral-indicator sets-measure-spmf sets-count-space Pow-UNIV UNIV-I emeasure-spmf-single) simp 

unfolding split-def
by (intro nn-integral-mono if-mono-cong) (auto simp add: measure-spmf.emeasure-eq-measure)
also have \( \ldots \leq q2 / 2 \wedge \text{len} \) by (simp add: split-def weight1 measure-spmf.emeasure-eq-measure)
finally have game2-d-bad: \( \text{spmf} \text{(map-spmf snd game2-d)} \text{True} \leq q2 / 2 \wedge \text{len} \) by simp 

define game3 where game3 = do {
\((p_0, p_1, \sigma, s_1) \leftarrow \text{exec-gpv rf-encrypt} \ \sigma / 1 \ \text{Map.empty};\)
if \text{valid-plain} p_0 \land \text{valid-plain} p_1 \ \text{then do} \{
\begin{align*}
b & \leftarrow \text{coin-spmf}; \\
\text{let} \ pb = (\text{if} \ b \ \text{then} \ p_0 \ \text{else} \ p_1); \\
r & \leftarrow \text{spmf-of-set (nlists UNIV len)}; \\
\text{pad} & \leftarrow \text{spmf-of-set (nlists UNIV len)}; \\
\text{let} \ \text{cipher} = (r, \text{xor-list} \ pb \ \text{pad}); \\
\text{(}b', s_2\text{)} & \leftarrow \text{exec-gpv rf-encrypt} \ (\sigma / 2 \ \text{cipher} \ \sigma) \ s_1; \\
\text{return-spmf} \ (b' = b)
\end{align*}
\} \ else \ \text{coin-spmf}
\}
\text{have} \ bisim2d-3: \ \lambda \sigma s r. \ \text{rel-spmf} \ (\lambda \ (x, -) (y, -), \ x = y) \\
\text{(exec-gpv} \ (\text{rf-encrypt-bad} \ r) \ (\sigma / 2 \ \text{cipher} \ \sigma) \ (s, \text{False})) \\
\text{(exec-gpv} \ \text{rf-encrypt} \ (\sigma / 2 \ \text{cipher} \ \sigma) \ s)
\by\ (\text{rule exec-gpv-oracle-bisim} \ \text{where} \ X=\lambda \ (s_1, \ s_2, s_1 = s_2)) \ \text{(auto simp add: rf-encrypt-bad-def}
\text{ rf-encrypt-def intro!} \ \text{rel-spmf-bind-refl})
\text{have} \ game2d-3: \ \text{rel-spmf} \ (=) \ (\text{map-spmf} \ \text{fst} \ \text{game2-d}) \ \text{game3}
\text{unfolding} \ \text{game2-d-def} \ \text{game3-def} \ \text{Let-def} \ \text{including} \ \text{monad-normalisation}
\by\ (\text{clarsimp simp add: o-def split-def if-distrib cong; if-cong intro!} \ \text{rel-spmf-bind-refl}
\text{rel-spmf-bind})
\text{have} \ \text{spmf} \ \text{game2} \ True - 1 / 2 \leq 
\text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-a}) \ \text{True} - \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-b}) \ \text{True} + \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-d}) \ True - 1 / 2
\text{unfolding} \ \text{game2-a-2b} \ \text{by}(\text{rule abs-diff-triangle-ineq2})
\text{also have} \ \ldots \leq q_1 / 2 \ ^ \text{len} + \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-b}) \ \text{True} - 1 / 2
\text{using} \ \text{game2a-2b} \ \text{game2a-bad-bound} \ \text{unfolding} \ \text{game2a-bad} \ \text{by}(\text{intro add-right-mono})
\text{simp}
\text{also have} \ \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-b}) \ \text{True} - 1 / 2 \leq 
\text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-c}) \ \text{True} - \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-d}) \ \text{True} + \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-d}) \ True - 1 / 2
\text{unfolding} \ \text{game2b-c} \ \text{by}(\text{rule abs-diff-triangle-ineq2})
\text{also} \ (\text{add-left-mono-trans}) \ \text{have} \ \ldots \leq q_2 / 2 \ ^ \text{len} + \text{spmf} \ (\text{map-spmf} \ \text{fst} \ \text{game2-d}) \ \text{True} - 1 / 2
\text{using} \ \text{game2c-2d} \ \text{game2-d-bad} \ \text{unfolding} \ \text{game2c-2d-bad} \ \text{by}(\text{intro add-right-mono})
\text{simp}
\text{finally} \ (\text{add-left-mono-trans})
\text{have} \ \text{game2:} \ \text{spmf} \ \text{game2} \ \text{True} - 1 / 2 \leq q_1 / 2 \ ^ \text{len} + q_2 / 2 \ ^ \text{len} + \text{spmf} \ \text{game3} \ \text{True} - 1 / 2
\text{using} \ \text{game2d-3} \ \text{by}(\text{simp add: field-simps spmf-rel-eq})
\text{have game3 = do} \{
\begin{align*}
((p_0, p_1, \sigma), s_1) & \leftarrow \text{exec-gpv rf-encrypt} \ \sigma / 1 \ \text{Map.empty}; \ \\
\text{if} \ \text{valid-plain} p_0 \land \text{valid-plain} p_1 \ \text{then do} \{
\begin{align*}
b & \leftarrow \text{coin-spmf}; \\
\text{let} \ pb = (\text{if} \ b \ \text{then} \ p_0 \ \text{else} \ p_1); \\
r & \leftarrow \text{spmf-of-set (nlists UNIV len)}; \\
\text{pad} & \leftarrow \text{spmf-of-set (nlists UNIV len)}; \\
\text{let} \ \text{cipher} = (r, \text{xor-list} \ pb \ \text{pad});
\end{align*}
\}
\[(b', s2) \leftarrow \text{exec-gpv rf-encrypt} (\mathcal{A}/2 \text{ cipher } \sigma) s1; \]
\[\text{return-spmf} \ (b' = b)\]
\}


also have \(\ldots = \) \(\) do \{ \[(p0, p1), (\sigma), s1) \leftarrow \text{exec-gpv rf-encrypt} \mathcal{A}/1 \text{ Map.empty}; \]
\[\text{if valid-plain} \ p0 \wedge \text{valid-plain} \ p1 \text{ then do } \{\]
\[b \leftarrow \text{coin-spmf}; \]
\[\text{let } pb = (\text{if } b \text{ then } p0 \text{ else } p1);\]
\[r \leftarrow \text{spmf-of-set} (\text{nlists UNIV len}); \]
\[\text{pad} \leftarrow \text{spmf-of-set} (\text{nlists UNIV len});\]
\[\text{let cipher} = (r, \text{pad});\]
\[(b', -) \leftarrow \text{exec-gpv rf-encrypt} (\mathcal{A}/2 \text{ cipher } \sigma) s1; \]
\[\text{return-spmf} \ (b' = b)\]
\} \text{ 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 \(\ldots = \) \(\) do \{ \[(p0, p1), (\sigma), s1) \leftarrow \text{exec-gpv rf-encrypt} \mathcal{A}/1 \text{ Map.empty}; \]
\[\text{if valid-plain} \ p0 \wedge \text{valid-plain} \ p1 \text{ then do } \{\]
\[r \leftarrow \text{spmf-of-set} (\text{nlists UNIV len}); \]
\[\text{pad} \leftarrow \text{spmf-of-set} (\text{nlists UNIV len});\]
\[\text{let cipher} = (r, \text{pad});\]
\[(b', -) \leftarrow \text{exec-gpv rf-encrypt} (\mathcal{A}/2 \text{ cipher } \sigma) s1; \]
\[
\text{map-spmf} \ ((=) b') \text{ coin-spmf}\]
\} \text{ else coin-spmf}\]

including monad-normalisation by (simp add: map-spmf-conv-bind-spmf split-def Let-def)

also have \(\ldots = \) \(\) coin-spmf

by (simp add: map-eq-const-coin-spmf Let-def split-def weight2 weight1)

finally have game3: \(\) game3 = coin-spmf.

have ind-cpa.advantage \(\mathcal{A} \leq \text{prf.advantage} (\text{prf-adversary} \ \mathcal{A}) + |\text{spmf} (\text{prf.game-1} (\text{prf-adversary} \ \mathcal{A})) \ \text{True} \ - \ 1 / 2|\)

unfolding ind-cpa.advantage-def prf.advantage-def ind-cpa-0[unfolded spmf-rel-eq]

by (rule abs-diff-triangle-ineq2)

also have \(|\text{spmf} \ (\text{prf.game-1} (\text{prf-adversary} \ \mathcal{A})) \ \text{True} - 1 / 2 | \leq q1 / 2 \ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ len \)

using game1-2 game2 game3 by (simp add: spmf-of-set)

also have \(\ldots = (q1 + q2) / 2 \ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ len\) by (simp add: field-simps)

also have \(\ldots \leq q / 2 \ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ ^\ len\) using \(q1 + q2 \leq q\) by (simp add: divide-right-mono)

finally show \?thesis by (simp add: field-simps)

qed

lemma interaction-bounded-prf-adversary:

fixes \(q :: \text{nat}\)
assumes \( \text{ind-cpa.\textit{ibounded-by } } \mathcal{A} \ \eta \)

shows \( \text{prf.\textit{ibounded-by } } (\text{prf-adversary } \mathcal{A}) (1 + q) \)

proof

\[
\text{fix } \eta \\
\text{from \ asssms \ have \ ind-cpa.\textit{ibounded-by } } \mathcal{A} \ \eta \ \text{by blast} \\
\text{then obtain } q_1 \ q_2 \text{ where } q : q_1 + q_2 \leq q \\
\text{and } [\text{interaction-bound}]: \text{interaction-any-\textit{bounded-by } } (\text{fst } \mathcal{A}) q_1 \\
\quad \forall x \ \sigma. \text{interaction-any-\textit{bounded-by } } (\text{snd } \mathcal{A} \ x \ \sigma) q_2 \\
\text{unfolding \ prf.\textit{ibounded-by} \ by (auto \ simp \ add \ split-beta \ iadd-le-enat-iff)} \\
\text{show \ prf.\textit{ibounded-by } } (\text{prf-adversary } \mathcal{A}) (1 + q) \text{ using } q \\
\text{apply } \text{(simp only: prf-adversary-def Let-def split-def)} \\
\text{apply } \text{interaction-bound} \\
\text{apply } \text{(auto \ simp \ add \ iadd-SUP-le-iff \ SUP-le-iff \ add.assoc \ [symmetric] \ one-enat-def \ cong \ del: image-cong-simp cong add: SUP-cong-simp)} \\
\text{done} \\
\text{qed}
\]

lemma \( \text{lossless-prf-adversary: ind-cpa.lossless } \mathcal{A} =⇒ \text{prf.lossless } (\text{prf-adversary } \mathcal{A}) \)

by (fastforce simp add: prf-adversary-def Let-def split-def ind-cpa.lossless-def intro: lossless-inline)

end

locale otp-\( \eta \) =

| \text{fixes } f :: \text{security } ⇒ \text{key } ⇒ \text{bool list } ⇒ \text{bool list} |
| \text{and } \text{len :: security } ⇒ \text{nat} |

assumes \( \text{length-f: } \forall \eta \ xs \ ys. \ [\text{length } xs = \text{len } \eta; \text{length } ys = \text{len } \eta] \implies \text{length } (f \ \eta xs \ ys) = \text{len } \eta \)

and \( \text{negligible-len [negligible-intros]: negligible } (\lambda \eta. 1 / 2 ^ (\text{len } \eta)) \)

begin

interpretation otp \( f \ \eta \) \text{len } \eta \text{ for } \eta \text{ by (unfold-locales) (rule length-f)}

interpretation ind-cpa: \text{ind-cpa.key-gen } \eta \text{ encrypt } \eta \text{ decrypt } \eta \text{ valid-plain } \eta \text{ for } \eta .

interpretation prf: \text{prf.key-gen } \eta \ f \ \eta \text{ spmf-of-set } (\text{nlists } \text{UNIV } (\text{len } \eta)) \text{ for } \eta .

lemma \( \text{prf-encrypt-secure-for:} \)

assumes \( \text{[negligible-intros]: negligible } (\lambda \eta. \text{prf.advantage } \eta \ (\text{prf-adversary } \mathcal{A} \ \eta)) \)

and \( q: \forall \eta. \text{ind-cpa.\textit{ibounded-by } } (\mathcal{A} \ \eta) (q \ \eta) \text{ and [negligible-intros]: polynomial } q \)

and \( \text{lossless: } \forall \eta. \text{ind-cpa.\textit{lossless } } (\mathcal{A} \ \eta) \)

shows \( \text{negligible } (\lambda \eta. \text{ind-cpa.\textit{advantage } } \mathcal{A} \ \eta) \)

proof\text{(rule negligible-mono)}

show \( \text{negligible } (\lambda \eta. \text{prf.advantage } \eta \ (\text{prf-adversary } \mathcal{A} \ \eta)) + q \ \eta / 2 ^ \text{len } \eta \)

by \text{(intro negligible-intros)}

\{ \text{fix } \eta \}

from \text{ind-cpa.\textit{ibounded-by } } - - \text{ have } \text{ind-cpa.\textit{ibounded-by } } (\mathcal{A} \ \eta) (q \ \eta) \text{ by blast}

moreover from \text{lossless } \text{have } \text{ind-cpa.\textit{lossless } } (\mathcal{A} \ \eta) \text{ by blast}

hence \text{lossless-gpv } \text{I-full } (\text{fst } (\mathcal{A} \ \eta)) \text{ } (\text{cipher } \sigma. \text{lossless-gpv } \text{I-full } (\text{snd } (\mathcal{A} \ \eta) \text{ cipher } \sigma))

61
by (auto simp add: ind-cpa.lossless-def) ultimately have ind-cpa.advance \( (A \eta) \leq \text{prf.advance} \eta (\text{prf-adversary} \eta (A \eta)) + q \eta / 2 ^ {\text{len} \eta}

by (rule prf-encrypt-advantage)

detects \( \lambda \eta. \text{ind-cpa.advance} \eta (A \eta) \leq 1 \ast \text{prf.advance} \eta (\text{prf-adversary} \eta (A \eta)) + q \eta / 2 ^ {\text{len} \eta} \)

by (simp add: always-eventually ind-cpa.advance-nonneg prf.advance-nonneg)

detects (\lambda \eta. \text{ind-cpa.advance} \eta (A \eta)) \in O(\lambda \eta. \text{prf.advance} \eta (\text{prf-adversary} \eta (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:

[ key-prf \in set-spmf prf-key-gen; m \in \text{prf-domain} ] \implies \text{length} (\text{prf-fun} \text{key-prf} m) = \text{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-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 [\oplus] 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 [\oplus] 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 def]

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
\[
\text{\texttt{intercept-upf}} ~ k ~ b = (\lambda (L, D) (m1, m0). \\
\begin{cases}
\text{case (length } m1 = \text{prf-clen} \land \text{length } m0 = \text{prf-clen) of} \\
\text{False } \Rightarrow \text{Done (None, L, D)} \\
\text{True } \Rightarrow \text{do } \\
\quad x \leftarrow \text{lift-spmf (spmf-of-set prf-domain)}; \\
\quad t \leftarrow \text{prf-fun k x \[\oplus\] (if b then m1 else m0);} \\
\quad \text{Done ((Some (x, c, projl t)), (insert (x, c, projl t) L, D))} \\
\end{cases}
\]
\]

\textbf{definition} \texttt{intercept-upf-dec}

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

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

\textbf{definition} \texttt{intercept-upf} ::

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

\textbf{where}
\[
\text{intercept-upf } k ~ b = \text{plus-intercept (intercept-upf-enc } k ~ b) \text{ intercept-upf-dec}
\]

\textbf{lemma} \texttt{intercept-upf-simps} [simp]:

\[
\text{intercept-upf } k ~ b (L, D) (\text{Inr (x, c, t)}) = \\
\begin{cases}
\text{if } (x, c, t) \in L \lor \text{length } x \neq \text{prf-dlen then Done (None, (L, D)) else do } \\
\text{Pause (Inr (x @ c)) Done;} \\
\text{Done (Inr None, (L, insert (x, c, t) D))}
\end{cases}
\]

\[
\text{intercept-upf } k ~ b (L, D) (\text{Inl (m1, m0)}) = \\
\begin{cases}
\text{case (length } m1 = \text{prf-clen} \land \text{length } m0 = \text{prf-clen) of} \\
\text{False } \Rightarrow \text{Done (Inl None, L, D)} \\
\text{True } \Rightarrow \text{do } \\
\quad x \leftarrow \text{lift-spmf (spmf-of-set prf-domain)}; \\
\quad t \leftarrow \text{prf-fun k x \[\oplus\] (if b then m1 else m0);} \\
\quad \text{Done (Inl (Some (x, c, projl t)), (insert (x, c, projl t) L, D))}
\end{cases}
\]

64
by (simp-all add: intercept-upf-def intercept-upf-dec-def intercept-upf-enc-def o-def map-gpv-bind-gpv gpv.map-id Let-def split!: bool.split)

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

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

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

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

lemma lossless-intercept-upf-enc [simp]: lossless-gpv (I -full ⊕ 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 (I -full ⊕ I -full) (intercept-upf-dec LD mm)
by (simp add: intercept-upf-dec-def split-beta)

lemma lossless-intercept-upf [simp]: lossless-gpv (I -full ⊕ 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 (I -full ⊕ I -full) (intercept-upf k b LD x) ⊆ responses-ι (I -full ⊕ I -full) x × UNIV
by (cases x) (auto simp add: intercept-upf-def)

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

65
lemma lossless-reduction-upf [simp]:
lossless-gpv (I-full $\oplus_3$ I-full) $\rightarrow$ lossless-gpv (I-full $\oplus_2$ 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 (I-full $\oplus_3$ I-full) $\mathcal{A}$
  shows $\exists spmf (\text{ind-cca.game} \mathcal{A}) \text{True} = \exists spmf (\text{ind-cca'.game} \mathcal{A}) \text{True}$ \leq UPF.advantage (reduction-upf $\mathcal{A}$)

proof |
  define oracle-decrypt0' where oracle-decrypt0' $\equiv$ (\lambda key (bad, L) (x', c', t'). return-spmf
  if (x', c', t') $\in$ L $\vee$ length x' \neq prf-dlen then (None, (bad, L))
  else (decrypt key (x', c', t'), (bad $\lor$ upf-fun (snd key) (x' $@$ c') = t', L)))
  have oracle-decrypt0'-simp: oracle-decrypt0' key (bad, L) (x', c', t') = return-spmf
  if (x', c', t') $\in$ L $\vee$ length x' $\neq$ prf-dlen then (None, (bad, L))
  else (decrypt key (x', c', t'), (bad $\lor$ upf-fun (snd key) (x' $@$ c') = t', L))
  for key L bad x' c' t' by (simp add: oracle-decrypt0'-def)
  have lossless-oracle-decrypt0' [simp]: lossless-spmf (oracle-decrypt0' k L bad c) for k L bad 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' $\equiv$ (\lambda (key :: 'prf-key $\times$ 'upf-key) (bad, L) (x', c', t').
  return-spmf (None :: bitstring option,
  (bad $\lor$ upf-fun (snd key) (x' $@$ c') = t' $\land$ (x', c', t') $\notin$ L $\land$ length x' = prf-dlen), L))
  have oracle-decrypt1'-simp: oracle-decrypt1' key (bad, L) (x', c', t') = return-spmf (None,
  (bad $\lor$ upf-fun (snd key) (x' $@$ c') = t' $\land$ (x', c', t') $\notin$ L $\land$ length x' = prf-dlen), L))
  for key L bad x' c' t' by (simp add: oracle-decrypt1'-def)
  have lossless-oracle-decrypt1' [simp]: lossless-spmf (oracle-decrypt1' k L bad c) for k L bad 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' $\equiv$ (\lambda (decrypt :: 'prf-key $\times$ 'upf-key $\Rightarrow$ (bitstring $\times$ bitstring $\times$ 'hash),
  bitstring option, bool $\times$ (bitstring $\times$ bitstring $\times$ 'hash) set) callee) $\mathcal{A}$. do {
  key $\leftarrow$ key-gen;
  b $\leftarrow$ coin-spmf;
\[(b', (\text{bad}', L')) \leftarrow \text{exec-gpv} (\dagger (\text{ind-cca.oracle-encrypt} \text{ key} k) \oplus_O \text{decrypt} \text{ key}) \not\emptyset (\text{False,} \\ {\}) ); \left\}
\text{return-spmf} (b = b', \text{bad}') \}\]
\text{let} ?game0' = game01' \text{oracle-decrypt0'}
\text{let} ?game1' = game01' \text{oracle-decrypt1'}

\text{have} game0'-eq: \text{ind-cca.game} \not\emptyset = \text{map-spmf} f st (?game0' \not\emptyset) (\text{is} ?game0)
\text{and} game1'-eq: \text{ind-cca'.game} \not\emptyset = \text{map-spmf} f st (?game1' \not\emptyset) (\text{is} ?game1)

\text{proof} --
\text{let} ?S = \text{rel-prod2} (=)
\text{define} initial where initial = (\text{False,} {\}} :: \text{hash cipher-text set})
\text{have} [\text{transfer-rule}]: ?S {\} initial by (\text{simp add: initial-def})

\text{have} [\text{transfer-rule}]:
\left\}(=) \Longrightarrow ?S \Longrightarrow (=) \Longrightarrow \text{rel-spmf} (\text{rel-prod} (=) ?S)
\text{ind-cca.oracle-decrypt oracle-decrypt0'}
\text{unfolding} \text{ind-cca.oracle-decrypt-def}[\text{abs-def}] \text{oracle-decrypt0'-def}[\text{abs-def}]
\text{by} (\text{simp add: rel-spmf-return-spmf1 rel-fun-def})

\text{have} [\text{transfer-rule}]:
\left\}(=) \Longrightarrow ?S \Longrightarrow (=) \Longrightarrow \text{rel-spmf} (\text{rel-prod} (=) ?S)
\text{ind-cca'.oracle-decrypt oracle-decrypt1'}
\text{unfolding} \text{ind-cca'.oracle-decrypt-def}[\text{abs-def}] \text{oracle-decrypt1'-def}[\text{abs-def}]
\text{by} (\text{simp add: rel-spmf-return-spmf1 rel-fun-def})

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

\text{have} \vdash: \text{rel-spmf} (\lambda (b'1, (\text{bad}1, L1)) (b'2, (\text{bad}2, L2)). \text{bad}1 = \text{bad}2 \land (\neg \text{bad}2 \rightarrow b'1 = b'2))
\left(\text{exec-gpv} (\dagger (\text{ind-cca.oracle-encrypt} k \text{ b}) \oplus_O \text{oracle-decrypt1'} k) \not\emptyset (\text{False,} {\}})
\left(\text{exec-gpv} (\dagger (\text{ind-cca.oracle-encrypt} k \text{ b}) \oplus_O \text{oracle-decrypt0'} k) \not\emptyset (\text{False,} {\}})
\text{for} k \text{ b}
\text{by} (\text{cases} k; \text{rule exec-gpv-oracle-bisim-bad}[\text{where} X=(=) \text{ and} ?\text{bad}1.0=\text{fst} \text{ and} ?\text{bad}2.0=\text{fst} \text{ and} \not\emptyset = \not\emptyset \oplus \not\emptyset \not\emptyset \text{-full})
\left(\text{auto intro: rel-spmf-refl1 callee-invariant-extend-state-oracle-const simp add: spmf-rel-map1 spmf-rel-map2 oracle-decrypt0'-simp's oracle-decrypt1'-simp's assms split: plus-oracle-split})
\right)
\text{We cannot get rid of the losslessness assumption on} \not\emptyset \text{ in this step, because if it were not, then the bad event might still occur, but the adversary does not terminate in the case of game01' oracle-decrypt1'. Thus, the reduction does not terminate either, but it cannot detect whether the bad event has happened. So the advantage in the UPF game could be lower than the probability of the bad event, if the adversary is not lossless.}
\text{have} [\text{measure} (\text{measure-spmf} (?\text{game0'} \not\emptyset)) \{ (b, \text{bad}). b \} - \text{measure} (\text{measure-spmf} (?\text{game0'} \not\emptyset)) \{ (b, \text{bad}). b \}]
\leq \text{measure} (\text{measure-spmf} (?\text{game1'} \not\emptyset)) \{ (b, \text{bad}). \text{bad} \}
\text{by} (\text{rule fundamental-lemma}[\text{where} ?\text{bad}2.0=\text{snd}]) (\text{auto intro!: rel-spmf-bind-refl1})
rel-spmf-bindI[OF \star] simp add: game01-def
also have \ldots = \spmf\ \{\spmf \\text{snd} \?game1\ \?}\ True
by (simp add: \spmf-com-measure-\spmf \text{measure-map-}\spmf \text{split-def vimage-def})
also have \spmf \\text{snd} \?game1\ \? = \UPF.game \text{(reduction-upf \?)}
proof
-
  note \text{split def} = if-split
have \spmf (\lambda x. \text{fst} (\text{snd} x)) \ (\text{exec-gpv} \ (\text{ind-cca}. \text{oracle-encrypt} \ (k-prf, k-upf) b)
\oplus_{O} \text{oracle-decryptI'} (k-prf, k-upf)) \? \ (False, \{\}) =
\spmf (\lambda x. \text{fst} (\text{snd} x)) \ (\text{exec-gpv} \ (\UPF. \text{oracle} k-upf) \ (\text{inline} \ (\text{intercept-upf} k-prf b) \? \ (\{\}, \{\}) \ (False, \{\}))
(is \spmf ?fl \?lhs = \spmf ?fr \?rhs is \spmf - \ (\text{exec-gpv} \ ?fr ?oracle-normal - ?init-normal) = -)
for k-prf k-upf b
proof(rule map-spmf-eq-map-spmfI)
define \text{oracle-intercept}
where \text{simp}: \text{oracle-intercept} = (\lambda (s', s) y. \spmf (\lambda ((x, s'), s) . (x, s', s))
(\text{exec-gpv} \ (\UPF. \text{oracle} k-upf) \ (\text{intercept-upf} k-prf b s' y) s))
let \?I = (\lambda ((L, D), (flg, Li)).
(\forall (x, c, t) \in L. \text{upf-fun} k-upf \ (x @ c) = t \land \text{length} x = \text{prf-dlen}) \wedge
(\forall e \in Li. \exists (x, c, t) \in L. e = x @ c) \land
(\exists (x, c, t) \in D. \text{upf-fun} k-upf \ (x @ c) = t \leftarrow flg))
interpret callee-invariant-on oracle-intercept ?I \text{-full}
apply(unfold-locales)
subgoal for s \ x \ y \ s'
  apply(cases x, cases s', cases x)
  apply(clarsimp simp add: set-spmf-of-set-finite[OF prf-domain-finite]
    \text{UPF}.\text{oracle-hash-def} \text{prf-domain-length} \text{exec-gpv-bind} \text{Let-def split: bool.splits})
  apply(force simp add: exec-gpv-bind \text{UPF}.\text{oracle-flag-def} \text{split: if-split-asm})
done
subgoal by simp
done

define S :: bool \times \text{'hash cipher-text set} \Rightarrow (\text{'hash cipher-text set} \times \text{'hash cipher-text set}) \times bool \times \text{bitstring set} \Rightarrow bool
where S = (\lambda (\text{bad}, \?L). ((L2, D), -). \text{bad} = (\exists (x, c, t) \in D. \text{upf-fun} k-upf \ (x @ c) = t) \land L1 = \?L) \ (\lambda -. True) \ ?I
define initial :: (\text{'hash cipher-text set} \times \text{'hash cipher-text set}) \times bool \times \text{bitstring set}
where initial = (\{\}, \{\}, (False, \{\}))
have [transfer-rule]: S ?init-normal initial by(simp add: S-def initial-def)
have [transfer-rule]: (S \Longrightarrow (=) \Longrightarrow \text{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-\text{pmf-assoc} bind-assoc-\text{pmf} bind-return-\text{pmf} \text{spmf-rel-map} \text{exec-gpv-bind} \text{Let-def}
\text{ind-cca}.\text{oracle-encrypt-def} \text{oracle-decryptI'-def} \text{decrypt-simps} \text{UPF.\text{oracle-hash-def} \text{UPF.\text{oracle-flag-def} bind-map-spmf o-def split: plus-oracle-split bool.split if-split intro!: rel-spmf-bind-reflI rel-pmf-bind-reflI})
have rel-spmf (rel-prod (=) S) ?lhs (exec-gpv oracle-intercept \? \ initial)
by (transfer-prover)
then show \( \text{rel-spmf} (\lambda x y. ?fl x = ?fr y) ?lhs ?rhs \)
  by (auto simp add: S-def exec-gpv-inline spmf-rel-map initial-def elim: rel-spmf-mono)
qed
then show \(?thesis\) including monad-normalisation
  by (auto simp add: reduction-upf-def UPF game-def game01'-def key-gen-def map-spmf-conv-bind-spmf
    split-def exec-gpv-bind intro!: bind-spmf-cong [OF refl])
qed
finally show \(?thesis\) using game0' eq game1' eq
  by (auto simp add: spmf-conv-measure-spmf measure-map-spmf vimage-def fst-def UPF
    advantage-def)
qed

definition oracle-encrypt2 :: 
  ('prf-key × 'upf-key) ⇒ bool ⇒ (bitstring, bitstring) PRF.dict ⇒ bitstring × bitstring
⇒ ('hash cipher-text option × (bitstring, bitstring) PRF.dict) spmf
where
oracle-encrypt2 = (λ(k-prf, k-upf) b D (msg1, msg0). (case (length msg1 = prf-clen ∧ length msg0 = prf-clen) of
  False ⇒ return-spmf (None, D)
| True ⇒ do 
  x ← spmf-of-set prf-domain;
  P ← spmf-of-set (nlists UNIV prf-clen);
  let p = (case D x of Some r ⇒ r | None ⇒ P);
  let c = p [⊕] (if b then msg1 else msg0);
  let t = upf-fun k-upf (x @ c);
  return-spmf (Some (x, c, t), D(x↦→ p))
))

definition oracle-decrypt2 :: ('prf-key × 'upf-key) ⇒ (hash cipher-text, bitstring option, state) callee
where oracle-decrypt2 = (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-spf)

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 A ≡ do 

69
key ← key-gen;
(b', D) ← exec-gpv
(oracle-encrypt2 key b ⊕_O oracle-decrypt2 key) ≡ 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 ≡ = do {
k ← lift-spmf upf-key-gen;
b ← lift-spmf coin-spmf;
(b', -) ← inline (intercept-prf k b) ();
Done (b' = b)
}

lemma round-2: | spmf (ind-cca'.game ≡) True − spmf (game2 ≡) True| = PRF.advantage
(reduction-prf ≡)
proof −
define oracle-encrypt1'' where
oracle-encrypt1'' = (∀(k-prf, k-upf) b (- :: unit) (msg1, msg0). case length msg1 = prf-clen ∧ length msg0 = prf-clen of
False ⇒ return-spmf (None, ())
| True ⇒ do {
x ← spmf-of-set prf-domain;
let p = prf-fun k-prf x;
let c = p [⊕] (if b then msg1 else msg0);
let t = upf-fun k-upf (x @ c);
return-spmf (Some (x, c, t), ()))}
define game1'' where
game1'' = do {
key ← key-gen;
b ← coin-spmf;
}
\[(b', D) \leftarrow \text{exec-gpv} \ (\text{oracle-encrypt}1'' \text{ key } b \oplus_O \text{ oracle-decrypt2 } \text{ key}) \not\in\ ()\text{; return-spmf } (b = b')\]

**have** ind-cca'.game \not\in = \text{game1}''

**proof**

- **define** \( S \text{ where } S = (\lambda (L :: \text{ 'hash cipher-text set} ) \ (D :: \text{ unit}) \cdot \text{ True}) \)
- **have** [transfer-rule]: \( S \{ \} \) by (simp add: S-def)
- **have** [transfer-rule]:
  \[
  (\ (=) ==>) (=) ==>) S ==>) (=) ==>) \text{ rel-spmf} \ (\text{ rel-prod} \ (=) S)\]
  **unfolding** ind-cca'.oracle-encrypt-def [abs-def] oracle-encrypt1''.def [abs-def]
- **by** (auto simp add: rel-fun-def Let-def S-def simp add: rel-spmf-bind-reflI rel-spmf-reflI split: bool.split)

**show** ?thesis **unfolding** ind-cca'.game-def game1''-def by transfer-prover

**qed**

**also have** \( \ldots = \text{PRF} \cdot \text{game-0} \ (\text{ reduction-prf} \ not\in) \)

**proof**

- \{ fix k-prf k-upf b \}
- **define** oracle-normal
  - **where** oracle-normal = oracle-encrypt1'' (k-prf, k-upf) b \oplus_O oracle-decrypt2 (k-prf, k-upf)
- **define** oracle-intercept
  - **where** oracle-intercept = \((\lambda (s', s :: \text{ unit}) \cdot \text{ map-spmf} \ (\lambda ((x, s'), s) \cdot (x, s', s)) \)
- **exec-gpv** (PRF.prf-oracle k-prf) (intercept-prf k-upf b s' y) ()
- **define** initial where initial = ()
- **define** \( S \text{ where } S = (\lambda (s2 :: \text{ unit}, - :: \text{ unit}) \cdot (s1 :: \text{ unit}) \cdot \text{ True}) \)
- **have** [transfer-rule]: \( S \{ (), () \} \) initial by (simp add: S-def init-def)
- **have** [transfer-rule]: \( (S ==>) (=) ==>) \text{ rel-spmf} \ (\text{ rel-prod} \ (=) S) \)
  - **oracle-intercept**
  - **oracle-normal**
  - **unfolding** oracle-normal-def oracle-intercept-def
- **have** map-spmf \((\lambda x. b = \text{ fst} \ x) \ (\text{exec-gpv oracle-normal} \ not\in \ initial) =\)
  - map-spmf \((\lambda x. b = \text{ fst} \ (\text{fst} \ x)) \ (\text{exec-gpv} \ (PRF.prf-oracle k-prf)) \ (\text{inline} \ (\text{intercept-prf k-upf} b) \ not\in \ ()) ()\)
  - **by** (transfer fixing: b \ not\in prf-fun k-prf prf-domain prf-clen upf-fun k-upf)
  - (auto simp add: map-spmf-eq-map-spmf-iff exec-gpv-inline spmf-rel-map oracle-intercept-def split-def intro!: rel-spmf-reflI)
- **then show** ?thesis **unfolding** game1''-def PRF.game-0-def key-gen-def reduction-prf-def by (auto simp add: exec-gpv-bind-lift-spmf exec-gpv-bind map-spmf-conv-bind-spmf split-def eq-commute intro!: bind-spmf-cong[OF refl])

**qed**
also have game2 ⇧ = PRF.game-1 (reduction-prf ⇧)
proof –
  note [split del] = if-split
  have k-upf b k-prf
  define oracle2
    where oracle2 = oracle-encrypt2 (k-prf, k-upf) b ⊕O oracle-decrypt2 (k-prf, k-upf)
  define oracle-intercept
    where oracle-intercept = (λ(x’, s). map-spmf (λ((x, s’), s). (x, s’, s)) (exec-gpv PRF.random-oracle (intercept-prf k-upf b s’ y) s))
  define S
    where S = (λ(s2 :: unit, s2’). (s1 :: (bitstring, bitstring) PRF.dict). s2’ = s1)

  have [transfer-rule]: S = (λ(x. b = fst (fst x)) (exec-gpv (PRF.random-oracle))
    (inline (intercept-prf k-upf b) ⇧ ())) Map.empty =
    map-spmf (λ(x. b = fst x) (exec-gpv oracle2 ⇧ Map.empty)
      by (transfer fixing: b prf-clen prf-domain upf-fun k-upf ⇧ 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: PRFadic)
  qed

ultimately show ?thesis by simp add: PRF.advan-def
qed

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

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

return-spmf (Some (x, c, t), (bad ∨ F, D(x ▷ p)))

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

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

deinition game3 :: (bitstring, 'hash cipher-text) ind-cca adversary ⇒ (bool × bool) spmf

where

game3 α ≡ do

key ← key-gen;

b ← coin-spmf;

(b', (bad, D)) ← exec-gpv (oracle-encrypt3 key b ⊕ O oracle-decrypt2 key) α (False, Map-empty);

return-spmf (b = b', bad)

lemma round-3:
assumes lossless-gpv (I-full ⊕ I full) A
shows |measure (measure-spmf (game3 A)) {(b, bad). b = spmf (game2 α) True} − spmf (game3 A) True| ≤ |measure (measure-spmf (game3 A)) {(b, bad), bad}|

proof −

define oracle-encrypt2' where

oracle-encrypt2' = (λ(k-prf :: 'prf-key, k-upf) b (bad, D) (msg1, msg0). case length msg1 = prf-clen ∧ length msg0 = prf-clen of False ⇒ return-spmf (None, (bad, D)) | True ⇒ do {

x ← spmf-of-set prf-domain;
P ← spmf-of-set (nlists UNIV prf-clen);

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

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

let t = upf-fun k-upf (x ▷ c);

return-spmf (Some (x, c, t), (bad ∨ F, D(x ▷ p)))
})

have [simp]: lossless-spmf (oracle-encrypt2' key b D msg10) for key b D msg10
by (cases msg10) (simp add: oracle-encrypt2'-def prf-domain-nonempty prf-domain-finite split-def Let-def split: bool.splits)

have [simp]: callee-invariant (oracle-encrypt2' key b) fst for key b
by (unfold-locales) (auto simp add: oracle-encrypt2'-def split-def Let-def split: bool.splits)

define game2' where

game2' = (λ.α. do {

\end{verbatim}
key ← key-gen;
b ← coin-spmf;
(b′, (bad, D)) ← exec-gpv (oracle-encrypt2 key b ⊕ oracle-decrypt2 key) /\ (False, Map-empty);
return-spmf (b = b′, bad))

have [\textit{game2'-eq}]: \textit{game2} /\ = \textit{map-spmf} \textit{fst} (\textit{game2'} /\ )
proof –
define \textit{S} where \textit{S} = (λ(D1 :: (\textit{bitstring}, \textit{bitstring}) \textit{PRF}.dict) (\textit{bad} :: \textit{bool}, D2). D1 = D2)

have [\textit{transfer-rule}, \textit{simp}]: \textit{S Map-empty} (\textit{b}, \textit{Map-empty}) for \textit{b} by (\textit{simp add: S-def})

have [\textit{transfer-rule}]: ((=) ===> (=)) ===> \textit{S} ===> (=) ===> \textit{rel-spmf} \textit{(rel-prod (=) \textit{S})}
oracle-encrypt2 oracle-encrypt2'
unfolding oracle-encrypt2-def[\textit{abs-def}] oracle-encrypt2'-def[\textit{abs-def}]
by (auto simp add: rel-fun-def Let-def split-def S-def
intro!: rel-spmf-bind-reflI split: bool.split option.split)

have [\textit{transfer-rule}]: ((=) ===> \textit{S} ===> (=) ===> \textit{rel-spmf} \textit{(rel-prod (=) \textit{S})})
oracle-decrypt2 oracle-decrypt2
by(auto simp add: rel-fun-def oracle-decrypt2-def)

show \textit{?thesis} unfolding game2-def game2'-def
by (simp add: \textit{map-spmf-bind-spmf} o-def split-def Map-empty-def[\textit{symmetric}] del: Map-empty-def)

transfer-prover

qed

moreover have *: rel-spmf (λ(b'1, \textit{bad1}, \textit{L1}) (b'2, \textit{bad2}, \textit{L2}). (\textit{bad1} \leftrightarrow \textit{bad2}) \land (\neg \textit{bad2} \rightarrow b'1 \leftrightarrow b'2))

(exec-gpv (oracle-encrypt3 key b ⊕ oracle-decrypt2 key) /\ (False, Map-empty))
(exec-gpv (oracle-encrypt2' key b ⊕ oracle-decrypt2 key) /\ (False, Map-empty))

for key b
apply rule exec-gpv-oracle-bisim-bad

where \textit{X}=(=) \textit{and} \textit{X-bad} = λ- -. \textit{True and}

?bad1.0=\textit{fst} and ?bad2.0=\textit{fst} and \textit{I}=\textit{I-full} ⊕ \textit{I} \textit{I} \textit{I-full}
apply(simp-all add: assms)
apply(auto simp add: assms spmf-rel-map Let-def oracle-encrypt2'-def oracle-encrypt3-def
split: oracle-spmf prod.split bool.split option.split intro!: rel-spmf-bind-reflI rel-spmf-reflI)
done

have \textit{measure} (measure-spmf (\textit{game3} /\ )) \{(b, \textit{bad}), b\} \leq
measure (measure-spmf (\textit{game3} /\ )) \{(b, \textit{bad}), \textit{bad}\}

unfolding game2'-def game3-def
by(rule fundamental-lemma[\textit{where} ?bad2.0=\textit{snd}])(intro rel-spmf-bind-reflI rel-spmf-bindI[OF *]; clarsimp)
ultimately show \textit{?thesis} by(simp add: spmf-com-measure-spmf measure-map-spmf vimage-def fst-def)

qed

lemma round-4:
assumes lossless-gpv (\mathcal{F}^{\text{full}} \oplus \mathcal{F}^{\text{full}}) \mathcal{A}
shows map-spmf \text{fst} (game3 \mathcal{A}) = \text{coin-spmf}

proof 

define oracle-encrypt4
where oracle-encrypt4 = (\lambda (k-prf :: \text{prf-key}, k-upf) \ (s :: \text{unit}) \ (msg1 :: \text{bitstring}, msg0 :: \text{bitstring})).
case length msg1 = prf-clen \land length msg0 = prf-clen of
False \Rightarrow return-spmf (None, s)
| True \Rightarrow do 
  x \leftarrow \text{spmfn-of-set prf-domain};
  P \leftarrow \text{spmfn-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 \text{key-gen};
  (b', -) \leftarrow \text{exec-gpv (oracle-encrypt4 key} \oplus \text{O oracle-decrypt2 key}) \mathcal{A};
  map-spmf ((=) b') \text{coin-spmf})

have map-spmf \text{fst} (game3 \mathcal{A}) = game4 \mathcal{A}

proof 

note [split del] = if-split

define S where S = (\lambda (- :: \text{unit}) (- :: bool \times \text{bitstring, bitstring}) \text{PRF.dict}). True

define initial3 where initial3 = (False, Map.empty :: \text{bitstring, bitstring} \text{PRF.dict})

have [transfer-rule]: S () initial3 by (simp add: S-def)

have [transfer-rule]: ((=) === > (=) === > S === > (=) === > rel-spmf (rel-prod (=) === > S))

(\lambda key b. oracle-encrypt4 key) oracle-encrypt3

proof(intro rel-fun1; hypsubst)

fix key unit msg10 b Dbad

have map-spmf \text{fst} (oracle-encrypt4 key () msg10) = map-spmf \text{fst} (oracle-encrypt3 key b Dbad msg10)

unfolding oracle-encrypt3-def oracle-encrypt4-def

apply (clarsimp simp add: map-spmf-com-bind-spmf Let-def split: bool.split prod.split; rule conj1; clarsimp)

apply (rewrite in \equiv = \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-del cong def: 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)

75
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)

qeda

also have \ldots = coin-spmf

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

finally show thesis.

qed

lemma game3-bad:
assumes interaction-bounded-by isl A q
shows measure (measure-spmf (game3 A)) { (b, bad). bad } \leq q / card prf-domain * q

proof
have measure (measure-spmf (game3 A)) { (b, bad). bad } = spmf (map-spmf snd (game3 A)) True
by (simp add: map-eq-const-coin-spmf game4-def bind-spmf-const split-def lossless-exec-gpv[OF assms] lossless-weight-spmf)

also
have spmf (map-spmf (fst o snd) (exec-gpv (oracle-encrypt3 k b \oplus O oracle-decrypt2 k) False, Map.empty)) True \leq q / card prf-domain * q

if k, k \in set-spmf key-gen for k b
proof
obtain k-prf k-upf where k = (k-prf, k-upf) by (cases k)

let ?I = \lambda (bad, D). finite (dom D) \land dom D \subseteq 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 (bad, D). card (dom D)

show \forall x s y s'. \forall (y, s') \in set-spmf (?oracle s x); (?I s; isl x) \implies ?count s' \leq Suc (?count s)

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

show \forall (y, s') \in set-spmf (?oracle s x); (?I s; \neg isl x) \implies ?count s' \leq ?count s for s x y s'

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

show spmf (map-spmf (fst o snd) (?oracle s' x)) True \leq q / card prf-domain

if \exists I. \forall s' and bad: \neg fst s' and count: ?count s' < q + ?count (False, Map.empty)

and x: isl x

for s' x

proof –
obtain bad D where s' [simp]: s' = (bad, D) by(cases s')
from x obtain m1 m0 where x [simp]: x = Inl (m1, m0) by(auto elim: islE)
have *: (case D x of None ⇒ False | Some x ⇒ True) ←→ x ∈ dom D for x
by(auto split: option.split)
show ?thesis
proof(cases length m1 = prf-clen ∧ length m0 = prf-clen)
case True
with bad
  have spmf (map-spmf (fst o snd) (?oracle s' x)) True = pmf (bernoulli-pmf (card (dom D ∩ prf-domain)) / card prf-domain)) True
  by(simp add: spmf_map_comp o_def oracle-encrypt3-def k * bool_case_distrib\[where
h=λp. spmf (map-spmf - p) \cdot 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_def: if-split split: bool_split)
  also have ... = card (dom D ∩ prf-domain) / card prf-domain
  by(rule pmf_bernoulli_True)(auto simp add: field_simps prf-domain-finite prf-domain-nonempty card_gt_0_iff card mono)
  also have dom D ∩ prf-domain = dom D using \ by auto
  also have card (dom D) ≤ q using \ 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

(auto split: plus_oracle_split_asm simp add: oracle_decrypt2_def_asm)
then have spmf (map_spmf snd (game3 a)) True ≤ q / card prf-domain * q
  by(auto 4 simp add: game3_def map_spmf_bind_spmf o_def split_def map_spmf_conv_bind_spmf intro: spmf_bind_le)
finally show ?thesis .
qed

theorem security:
assumes lossless: lossless-gpv \[ A -full \oplus \not\ A -full \] s
and bound: interactionbounded-by isl s q
shows ind-cca.advantage s ≤
  PRF.advantage (reduction-prf as) + UPF.advantage (reduction-upf as) +
  real q / real (card prf-domain) * real q (is ?LHS ≤ \)
proof –
  have ?LHS ≤ \{ spmf \{ ind-cca.game s \} True \} - spmf \{ ind-cca'.game s \} True \} + |spmf \{ ind-cca'.game s \} True - 1 / 2 \}
  (is - ≤ ?round1 + ?rest) using abs_triangle_ineq by(simp add: ind-cca.advantage_def)
  also have ?round1 ≤ UPF.advantage (reduction-upf as)
  using lossless by(rule round_1)
  also have ?rest ≤ \{ spmf \{ ind-cca'.game s \} True \} - spmf \{ game2 as \} True \} + |spmf \}
\[(\text{game2 } A) \text{ True } - 1 / 2]\\[(\text{is } \leq \text{ ?round2 } + \text{?rest}) \text{ using abs-triangle-ineq by simp}\\also \text{ have } \text{ ?round2 } = \text{ PRF}\text{-advantage (reduction-prf } A\text{)} \text{ by (rule round-2)}\\also \text{ have } \text{ ?rest } \leq \text{ measure (measure-spmf (game3 } A\text{)) } \{(b, \text{bad}). b\} - \text{ spmf (game2 } A\text{) True} +\\\text{ measure (measure-spmf (game3 } A\text{)) } \{(b, \text{bad}). b\} - 1 / 2\\(\text{is } \leq \text{ ?round3 } + \text{-} \text{?rest}) \text{ using abs-triangle-ineq by simp}\\also \text{ have } \text{ ?round3 } \leq \text{ measure (measure-spmf (game3 } A\text{)) } \{(b, \text{bad}). b\} - \text{ spmf (game2 } A\text{) True} +\\\text{ measure (measure-spmf (game3 } A\text{)) } \{(b, \text{bad}). b\} - \text{ spmf (game2 } A\text{) True} +\\\text{ measure (measure-spmf (game3 } A\text{)) } \{(b, \text{bad}). b\} - \text{ spmf (game2 } A\text{) True} +\\\text{ using round-3[OF lossless] ,}\\also \text{ have } \ldots \leq q / \text{card prf-domain } \ast q \text{ using bound by (rule game3-bad)}\\also \text{ have measure (measure-spmf (game3 } A\text{)) } \{(b, \text{bad}). b\} = \text{ spmf coin-spmf True}\\using round-4[\text{OF lossless, symmetric}]\\by (\text{simp add: spmf-conv-measure-spmf measure-map-spmf vimage-def fst-def})\\also \text{ have } |... - 1 / 2| = 0 \text{ by (simp add: spmf-of-set)}\\finally \text{ show } \text{thesis by (simp)}\\qed\\\\\text{theorem security1:}\\\text{assumes lossless: lossless-gpv (I } \oplus I \text{-full}) \ A\\\text{assumes q': interaction-bounded-by (Not } \circ \text{isl } q'\\\text{and q': interaction-bounded-by (Not } \circ \text{isl } q'\\\text{shows ind-cca}.advantage A \leq\\\text{PRF}.advantage (reduction-prf } A\text{) } +\\\text{UPF}.advantage1 (\text{guessing-many-one}.reduction q' (\lambda -. reduction-upf } A\text{) () } \ast q' +\\\text{real } q \ast \text{real } q / \text{real (card prf-domain)}\\\text{proof=}\\\text{have ind-cca}.advantage A \leq\\\text{PRF}.advantage (reduction-prf } A\text{) } + \text{UPF}.advantage (reduction-upf } A\text{) +\\\text{real } q / \text{real (card prf-domain) } \ast q\\\text{using lossless } q \text{ by (rule security)}\\\text{also note } q'[\text{interaction-bound}]\\\text{have interaction-bounded-by (Not } \circ \text{isl } (\text{reduction-upf } A\text{) } q'\\\text{unfolding reduction-upf-def by (interaction-bound)}(\text{simp-all add: SUP-le-iff})\\\text{then have UPF}.advantage (reduction-upf } A\text{) \leq \text{UPF}.advantage1 (\text{guessing-many-one}.reduction q' (\lambda -. reduction-upf } A\text{) () } \ast q' +\\\text{by (rule UPF}.advantage-advantage1)\\\text{finally show } \text{thesis by (simp)}\\\qed\\\\\text{locale simple-cipher'} =\\\text{fixes prf-key-gen :: security } \Rightarrow '\text{prf-key spmf}\\\text{and prf-fun :: security } \Rightarrow '\text{prf-key } \Rightarrow \text{bitstring } \Rightarrow \text{bitstring}\\\text{and prf-domain :: security } \Rightarrow \text{bitstring set}\\\text{and prf-range :: security } \Rightarrow \text{bitstring set}\\\text{and prf-dlen :: security } \Rightarrow \text{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:\(\forall \eta. \text{simple-cipher}(\text{prf-key-gen}\ \eta)\ (\text{prf-fun}\ \eta)\ (\text{prf-domain}\ \eta)\ (\text{prf-dlen}\ \eta)\ (\text{prf-clen}\ \eta)\ (\text{upf-key-gen}\ \eta)\)
begin

sublocale simple-cipher

prf-key-gen \(\eta\) prf-fun \(\eta\) prf-domain \(\eta\) prf-range \(\eta\) prf-dlen \(\eta\) prf-clen \(\eta\) upf-key-gen \(\eta\)
for \(\eta\)
by(rule simple-cipher)

theorem security-asymptotic:
fixed \(q\ q'\ ::\ \text{security} ⇒ \text{nat}\)
assumes lossless: \(\forall \eta. \text{lossless-gpv}(I_{\text{full}} \oplus I_{\text{full}})(\alpha\ \eta)\)
and bound: \(\forall \eta. \text{interaction-bounded-by}isl(\alpha\ \eta)(q\ \eta)\)
and bound': \(\forall \eta. \text{interaction-bounded-by}(\text{Not} \circ isl)(\alpha\ \eta)(q'\ \eta)\)
and [negligible-intros]:

polynomial \(q'\) polynomial \(q\)
negligible \((\lambda\ \eta. \text{PRF}\text{-advantage}\ \eta\ (\text{reduction-prf}\ \eta\ (\alpha\ \eta)))\)

negligible \((\lambda\ \eta. \text{UPF}\text{-advantage1}\ \eta\ (\text{guessing-many-one}\text{-reduction}\ (q'\ \eta)\ (\lambda\text{-reduction-upf}\ \eta\ (\alpha\ \eta))))\)

negligible \((\lambda\ \eta. 1 / \text{card}(\text{prf-domain}\ \eta))\)

shows negligible \((\lambda\ \eta. \text{ind-cca}\text{-advantage}\ \eta\ (\alpha\ \eta))\)

proof

have negligible \((\lambda\ \eta. \text{PRF}\text{-advantage}\ \eta\ (\text{reduction-prf}\ \eta\ (\alpha\ \eta)) + \text{UPF}\text{-advantage1}\ \eta\ (\text{guessing-many-one}\text{-reduction}\ (q'\ \eta)\ (\lambda\text{-reduction-upf}\ \eta\ (\alpha\ \eta))))\)

\((q'\ \eta) + \text{real}(q\ \eta) / \text{real}(\text{card}(\text{prf-domain}\ \eta)) * \text{real}(q\ \eta))\)

by (rule negligible-intros)+

thus \(\text{thesis by}(\text{rule negligible-le})(\text{simp add: security1}\ \text{OF lossless bound bound'}\text{-ind-cca}\text{-advantage-nonneg})\)
qed

end

end

definition of theory Cryptographic-Constructions imports

Elgamal
Hashed-Elgamal
RP-RF
PRF-UHF
PRF-IND-CPA
PRF-UPF-IND-CCA
begin

end

79
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]. 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.

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

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

theory CryptHOL-Tutorial imports CryptHOL.CryptHOL begin

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

4 Modelling cryptography using CryptHOL

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

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

4.1 Security notions without oracles: the CDH assumption

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

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

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

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 cyclic-group` and comes with a generator $g_{\mathcal{G}}$. Basic facts about finite groups are formalized in the CryptHOL theory `CryptHOL.Cyclic-Group.3`

```isabelle
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
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.  

**type-synonym**  

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

The LCDH game itself is expressed as a function from the adversary $A$ to the sub-probability 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 $x ← p$; samples an element $x$ from the distribution $p$. The sampling is independent, unless the distribution $p$ depends on previously sampled variables. At the end of the block, the *return-spmf_* returns whether the adversary has won the game.

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

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

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

**definition** game :: 'grp adversary ⇒ bool spmf where
game $A$ = do 
  $x ← sample-uniform (order $G$);
  $y ← sample-uniform (order $G$);
  $zs ← A (\langle \rangle^x x) (\langle \rangle^y y);
  return-spmf (\langle \rangle^z (x * y) ∈ 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 $A$ = spmf (game $A$) True

---

4Actually, 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.’
This completes the formalisation of the LCDH game and we close the locale \textit{list-cdh} with \texttt{end}. The above definitions are now accessible under the names \texttt{game} and \texttt{advantage}. Furthermore, when we later instantiate the locale \textit{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 \textit{random-oracle} for modularity. The random oracle will return a \texttt{bitstring}, i.e. a list of booleans, of length \texttt{len}.

\begin{verbatim}
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 \((s, a, b)\texttt{oracle}'\) abbreviates \(s \Rightarrow a \Rightarrow (b \times s)\texttt{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 \(a \rightarrow \texttt{bitstring}\). 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.

\begin{verbatim}
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 := 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 \(\lambda x.\ None\), as no queries have been asked. For readability, we introduce an abbreviation:
\end{verbatim}
abbreviation (input) initial :: 'a state where initial = 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 σ x)
by (simp add: oracle-def split: option.split)

lemma fresh:
oracle σ x =
(do { bs ← spmf-of-set (nlists UNIV len);
    return-spmf (bs, σ(x ↦→ bs)) })
if σ x = None
using that by (simp add: oracle-def)
end

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

4.3 Cryptographic concepts: public-key encryption

A cryptographic concept consists of a set of operations and their functional behaviour. We have already seen two simple examples: the cyclic group in §4.1 and the random oracle in §4.2. We have formalized both of them as locales; we have not modelled their functional behavior as this is not needed for the proof. In this section, we now present a more realistic example: public-key encryption with oracle access.
A public-key encryption scheme consists of three algorithms: key generation, encryption, and decryption. They are all probabilistic and, in the most general case, they may access an oracle jointly with the adversary, e.g., a random oracle modelling a hash function. As before, the operations are modelled as parameters of a locale, \textit{ind-cpa-pk}.

- The key generation algorithm \textit{key-gen} outputs a public-private key pair.
- The encryption operation \textit{encrypt} takes a public key and a plaintext of type \textit{‘plain} and outputs a ciphertext of type \textit{‘cipher}.
- The decryption operation \textit{decrypt} takes a private key and a ciphertext and outputs a plaintext.
- Additionally, the predicate \textit{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.

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

begin

The three actual operations are generative probabilistic values (GPV) of type \((\cdot, \textit{query}, \textit{response})\) 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 \textit{‘query} for queries and \textit{‘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.

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

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

**Stage 2 (guess):**

1. The challenger flips a coin $b$ (either 0 or 1) and gives $\text{encrypt } pk \ m_b$ 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 $P_{\text{win}}$ denote the winning probability. His advantage is $|P_{\text{win}} - 1/2|$

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.

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

The IND-CPA game formalization below follows the above informal definition. There are three points that need some explanation. First, this game differs from the simpler LCDH game in that it works with GPVs instead of SPMFs. Therefore, probability distributions like coin flips $\text{coin-spmf}$ must be lifted from SPMFs to GPVs using the coercion $\text{lift-spmf}$. Second, the assertion $\text{assert-gpv (valid-plains } m_0 m_1 \text{)}$ ensures that the pair of messages is valid. Third, the construct $\text{TRY \ ELSE \ _}$ 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.

```haskell
fun game :: ('pubkey', 'plain', 'cipher', 'query', 'response', 'state) adversary
  ⇒ (bool, 'query', 'response') gpv
where
  game (A₁, A₂) = TRY do {
    (pk, sk) ← key-gen;
    ((m₀, m₁), σ) ← A₁ pk;
    assert-gpv (valid-plains m₀ m₁);
    b ← lift-spmf $\text{coin-spmf}$;
```
cipher ← encrypt pk (if \( b \) then \( m_0 \) else \( m_1 \));
\( b' \leftarrow \mathcal{A}_2 \) cipher \( \sigma \);
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 \( \text{oracle} :: (s', q', r') \times s' \text{ initial state} \) and the initial state \( s' :: s' \text{ initial state} \). The operator \( \text{run-gpv} \) connects the game with the oracle, whereby the GPV becomes an SPMF.

\[
\text{fun advantage} :: (\sigma, q', r') \times \sigma \\
\Rightarrow (\text{pubkey, plain, cipher, query, response, state}) \text{ adversary} \Rightarrow \text{real}
\]

\[
\text{where advantage} (\text{oracle, } \sigma) \mathcal{A} = |\text{spm} (\text{run-gpv oracle (game } \mathcal{A}) \sigma) \text{ True } - 1/2|
\]

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 \textit{lcdh} and the random oracle for the hash function with namespace \textit{ro}. This ensures that the construction can be used for arbitrary cyclic groups. For the message space, it suffices to fix the length \textit{len-plain} of the plaintexts.

```isar
locale hashed-elgamal = 
  lcdh: list-cdh \mathcal{G} + 
  ro: random-oracle \textit{len-plain} 
  for \mathcal{G} :: \{grp cyclic-group (structure)
  and \textit{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 \( \{0, \ldots, q - 1\} \) 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 \( \{0, \ldots, q - 1\} \) 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.\footnote{More precisely, the private key ranges between 0 and \( q - 1 \) and the bitstrings are of length \textit{len-plain}. However, Isabelle/HOL’s type system cannot express such properties that depend on locale parameters.} For readability, we introduce meaningful abbreviations for these concepts.

**type-synonym** \textit{grp} pub-key = \textit{grp}
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 \emph{hash}. Here, \( \text{Pause } x \text{ Done} \) 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** \( \text{hash} :: \ell \rightarrow (\text{bitstring}, \ell, \text{bitstring}) \) **gpv**

**Definition** \( \text{key-gen} :: (\ell \text{ pub-key} \times \ell \text{ priv-key}) \) **spmf**

**Definition** \( \text{encrypt} :: \ell \text{ pub-key} \Rightarrow \text{plain} \Rightarrow (\ell \text{ cipher}, \ell, \text{bitstring}) \) **gpv**

**Definition** \( \text{decrypt} :: \ell \text{ priv-key} \Rightarrow \ell \text{ cipher} \Rightarrow (\text{plain}, \ell, \text{bitstring}) \) **gpv**

**Definition** \( \text{valid-plains} :: \text{plain} \Rightarrow \text{plain} \Rightarrow \text{bool} \)

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 \( \text{ind-cpa-pk} \) to our scheme.

**Sublocale** \( \text{ind-cpa} :: \text{ind-cpa-pk} (\text{lift-spmf key-gen}) \) \( \text{encrypt decrypt valid-plains} \).

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

5 Cryptographic proofs in CryptHOL

This section explains how cryptographic proofs are expressed in CryptHOL. We will continue our running example by stating and proving the IND-CPA security of the hashed Elgamal encryption scheme under the computational Diffie-Hellman assumption in the random oracle model, using the definitions from the previous section. More precisely, we will formalize a reduction argument (§5.1) and bound the IND-CPA advantage using the CDH advantage. We will not formally state the result that CDH hardness in the cyclic group implies IND-CPA security, which quantifies over all feasible adversaries—to that end, we would have to formally define feasibility, for which CryptHOL currently does not offer any support. The actual proof of the bound consists of several game transformations. We will focus on those steps that illustrate common steps in cryptographic proofs (§5.3–§5.8).

5.1 The reduction

The security proof involves a reduction argument: We will derive a bound on the advantage of an arbitrary adversary in the IND-CPA game game for hashed Elgamal that depends on another adversary’s advantage in the LCDH game game of the
underlying group. The reduction transforms every IND-CPA adversary \( \mathcal{A} \) into a LCDH adversary \( \mathcal{A}_{\text{elgamal-reduction}} \), 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 \( \lambda x. \text{None} \).
3. The assertion checks that the adversary returns two valid plaintexts, i.e., \( m_1 \) and \( m_2 \) are strings of length \( \text{len-plain} \).
4. Instead of actually performing an encryption, the reduction generates a random bitstring \( h \) of length \( \text{len-plain} \) (\( \text{nlists UNIV len-plain} \) denotes the set of all bitstrings of length \( \text{len-plain} \) and \( \text{spmf-of-set} \) 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 \( \text{dom} s' \) 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 \( \text{dom} s \) 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 (A₁, A₂) α β = do { ((m₁, m₂), σ) ← exec-gpv ro.oracle (A₁, α) ro.initial;
TRY do {
- :: unit ← assert-spmf (valid-plains m₁ m₂);
  h ← spmf-of-set (nlists UNIV len-plain);
  (b', s') ← exec-gpv ro.oracle (A₂ (β, h) σ) s;
  return-spmf (dom s')
} ELSE return-spmf (dom s)
}

5.2 Concrete security statement

A concrete security statement in CryptHOL has the form: Subject to some side conditions for the adversary A, the advantage in one game is bounded by a function of the transformed adversary’s advantage in a different game.

theorem concrete-security:
assumes side conditions for A
shows advantage₁ A ≤ f (advantage₂ (reduction 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 A
shows ind-cpa.advantage (ro.oracle, ro.initial) A ≤ lcdh.advantage (elgamal-reduction A)

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

---

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:

\[
\text{definition (in ind-cpa-pk) lossless} ::\ ('\text{pubkey}', '\text{plain}', '\text{cipher}', '\text{query}', '\text{response}', '\text{state}) \text{ adversary} \Rightarrow \text{bool}
\]

where

\[
\text{lossless} = (\lambda (A_1, A_2). (\forall \text{pk}. \text{lossless-gpv I -full} (A_1 \text{ pk}))
\land (\forall \text{cipher} \sigma. \text{lossless-gpv I -full} (A_2 \text{ cipher} \sigma)))
\]

So now let’s start with the proof.

\[
\text{proof –}
\]

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 \(\mathcal{A}\).\(^7\)

\[
\text{obtain } \mathcal{A}_1 \mathcal{A}_2 \text{ where } \mathcal{A} \text{ [simp]: } \mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2) \text{ by (cases } \mathcal{A})
\]

\[
\text{from lossless have lossless1 [simp]: } (\forall \text{pk. lossless-gpv I -full} (\mathcal{A}_1 \text{ pk}))
\land (\forall \text{cipher } \sigma. \text{lossless-gpv I -full} (\mathcal{A}_2 \sigma \text{ cipher})
\text{ by (auto simp add: ind-cpa.lossless-def})
\]

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 \textit{interceptor} between the adversary and the oracle that records all adversary’s queries.

\[
\text{define interceptor :: 'grp set } \Rightarrow \text{ 'grp } \Rightarrow (\text{bitstring } \times \text{ 'grp set, -}, -) \text{ gpv}
\]

where

\[
\text{interceptor } \sigma x = (\text{do } \{\text{h }\leftarrow \text{hash } x; \text{Done(h, insert x } \sigma)\}) \text{ for } \sigma x
\]

We integrate this interceptor into the \textit{game} using the \textit{inline} function as illustrated in Fig. 4 and name the result \textit{game}₀.

\[
\text{define game}_0 \text{ where}
\]

\(^7\)Later in the proof, we will often prove losslessness of the definitions in the proof. We will not show them in this document, but they are in the Isabelle sources from which this document is generated.
Figure 4: The IND-CPA game after expanding the key generation algorithm’s definition and inlining the query-recording hash oracle. The red boxes represent the inline operator.

game\_0 = \text{TRY do } \{
\begin{align*}
& (pk, -) \leftarrow \text{lift-spmf key-gen}; \\
& ((m_1, m_2), \sigma), s \leftarrow \text{inline interceptor} (\mathcal{A}_1 pk) \{\}; \\
& \text{assert-gpv} (\text{valid-plains } m_1 m_2); \\
& b \leftarrow \text{lift-spmf coin-spmf}; \\
& c \leftarrow \text{encrypt} pk \ (\text{if } b \text{ then } m_1 \text{ else } m_2); \\
& (b', s') \leftarrow \text{inline interceptor} (\mathcal{A}_2 c \sigma) s; \\
& \text{Done} (b' = b)
\end{align*}
\} \text{ 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\_0 have identical structure. They differ only in that game\_0 uses the adversary \((\lambda pk. \text{inline interceptor} (\mathcal{A}_1 pk) \emptyset, \lambda \text{cipher } \sigma. \text{inline interceptor} (\mathcal{A}_2 \text{cipher } \sigma))\) instead of \(\mathcal{A}\). The formal justification for this replacement happens in two steps:

1. We replace the oracle transformer \(\text{interceptor}\) with \(\text{id-oracle}\), which merely passes queries and results to the oracle.

2. Inlining the identity oracle transformer \(\text{id-oracle}\) 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 \textit{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 \([\text{transfer-rule}]\). 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: \textit{interceptor} records all queries in a set whereas \textit{id-oracle} has no state, which is modelled with the singleton type \textit{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 \textit{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 (\textbf{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 \textit{insert} w.r.t. \(cr\), the \textit{transfer-prover} then lifts this replacement to the bisimulation rule. Here, \textit{transfer-prover} is similar to \textit{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 \textit{transfer} works.

\[
\begin{align*}
\text{define } & cr :: \text{unit }\Rightarrow \text{grp set }\Rightarrow \text{bool where } cr \sigma \sigma' = \text{True for } \sigma \sigma' \\
\text{have } & [\text{transfer-rule}]: cr () {} \text{ by (simp add: cr-def) } \text{— initial states} \\
\text{have } & [\text{transfer-rule}]: ((=) ===> cr ===> cr) (\lambda-\sigma.\sigma) \text{ insert } \text{— state update} \\
\text{by (simp add: rel-fun-def cr-def)} \\
\text{have } & [\text{transfer-rule}]: cr \text{ is a bisimulation for the oracle transformers} \\
\text{by transfer-prover} \\
\text{have ind-cpa.game } \not\equiv \text{ game0 unfolding game0-def } \text{ inl-cpa.game.simps}\end{align*}
\]

\[
\begin{align*}
\text{by transfer (simp add: bind-map-gpv o-def ind-cpa.game.simps split-def)}
\end{align*}
\]

5.4 Equational program transformations

Before we move on, we need to simplify \textit{game0} and inline a few of the definitions. All these simplifications are equational program transformations, so the Isabelle simplifier can justify them. We combine the \textit{interceptor} with the random oracle \textit{oracle} into a new oracle \textit{oracle'} with which the adversary interacts.

\[
\begin{align*}
\text{define } & oracle' :: \text{grp set }\times (\text{grp }\Rightarrow \text{bitstring}) \Rightarrow \text{grp }\Rightarrow - \\
\text{where } & oracle' = (\lambda(s,\sigma) x. \text{ do } \{ \\
& \langle h, \sigma' \rangle \leftarrow \text{ case } \sigma x \text{ of }
\end{align*}
\]
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 σ s) σ =
  map-spmf (λ(a, b, c). ((a, b), c)) (exec-gpv oracle' σ (s, σ)) for 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].

have game0: run-gpv ro.oracle game0 ro.initial = do {
  x ← sample-uniform (order F);
  y ← sample-uniform (order F);
  b ← coin-spmf;
  ((msg1, msg2), (s, s-h)) ←
    exec-gpv oracle' (σ1 (g [σ] x)) ({}), ro.initial;
  TRY do {
    - :: unit ← assert-spmf (valid-plains msg1 msg2);
    (h, s-h') ← ro.oracle s-h (g [σ] (x*y));
    let cipher = (g [σ] y, h [σ]) (if b then msg1 else msg2);
    (b', (s', s-h'')) ←
      exec-gpv oracle' (σ2 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 assert-spmf-def
    try-bind-assert-gpv try-gpv-bind-lossless split-def o-def if-distrib 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 game0-def, key-gen-def, and encrypt-def should be unfolded, so they are added first to the simplifier’s set of rewrite rules.

8The 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.
A\text{1}
\text{encrypt}
A\text{2}
ro.oracle
\text{Gen. } sk, pk;
\text{b} \leftarrow \{0, 1\};
\text{pk}
\text{m}_0, m_1, \sigma, s, s_I
\text{pk}, \text{m}_0, \text{pk}, s_I
\text{c}, s_I
\text{c}, s_I
\text{b}', s', s'_{I'}
\text{b}' = b$

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 \text{exec-gpv-bind}, \text{try-bind-assert-gpv}, and \text{try-gpv-bind-lossless} ensure that the operator \text{exec-gpv}, which connects the \text{game}_0 with the random oracle, is distributed over the sequencing. Together with \ast, this gives the adversary access to \text{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 \text{lcdh.nat-pow-pow} rewrites the iterated exponentiation ($g [^\text{x}] x$) \text{[^y]} to $g [^\text{x}] (x \ast y)$.

- The other theorems \text{bind-map-spmf}, \text{assert-spmf-def}, \text{split-def}, \text{o-def}, and \text{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 \text{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 (\text{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 \texttt{game}_0 and \texttt{game}_1 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 \( g[^\dagger] (x \ast y) \) that is used for encrypting the chosen message \( m_0 \). So we simply change the return type of the game to return whether the adversary guessed right \textit{and} whether the failure event has occurred.

The next definition \texttt{game}_2 does so. (Recall that \texttt{oracle}' \textit{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 \texttt{game}_1.

\begin{verbatim}
define \texttt{game}_1 :: nat \Rightarrow nat \Rightarrow (bool \times bool) \texttt{spmf}
where \texttt{game}_1 x y = do 
  b <- \texttt{coin-spmf};
  \((m_1, m_2), \sigma), (s, s-h') \leftarrow \texttt{exec-gpv oracle}' (\emptyset_1 (g[^\dagger] x)) (\{\}, ro.initial);
 TRY do 
    h <- assert-spmf (valid-plains m_1 m_2);
    (s', s-h'') <- ro.oracle s-h (g[^\dagger] (x \ast y));
    let c = (g[^\dagger] y, h \[\oplus\] (if b then m_1 else m_2));
  \((b', (s', s-h'')) \leftarrow \texttt{exec-gpv oracle}' (\emptyset_2 c \sigma) (s, s-h');
  return-spmf (b' = b, g[^\dagger] (x \ast y) \in s') 
  ELSE do 
    b <- \texttt{coin-spmf};
    return-spmf (b, g[^\dagger] (x \ast y) \in s') 
  \} for x y
\end{verbatim}

It is easy to prove that \texttt{game}_0 combined with the random oracle is a projection of \texttt{game}_1 with the sampling added, as formalized in \texttt{game}_0-\texttt{game}_1.

\begin{verbatim}
let ?sample = \lambda f :: nat \Rightarrow nat \Rightarrow \texttt{spmf}. do 
  x <- \texttt{sample-uniform} (order \$);
  y <- \texttt{sample-uniform} (order \$);
  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) 
\end{verbatim}

\section{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 \texttt{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 \texttt{game}_2 is identical except that
the call to the random oracle \textit{oracle} is replaced with sampling a random bitstring.\footnote{In Shoup’s terminology \cite{19}, such a step makes (a gnome sitting inside) the random oracle forgetting the query.}

\begin{verbatim}
define game2 :: nat ⇒ nat ⇒ (bool × bool) spmf
where game2 x y = do 
  b ← coin-spmf;
  ((m1, m2), σ), (s, s-h)) ← exec-gpv oracle' (σ/s') (g [x] 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 \(g\ [x\ast y]\), but instead sample a random bitstring \(h\) directly. So the rest differs from \(game_1\) only if the adversary queries \(g\ [x\ast y]\).
    let cipher = (g [x\ast y], h [⊕]) (if b then m1 else m2);
    (b', (s', s-h')) ← exec-gpv oracle' (σ/s' cipher σ) (s, s-h);
    return-spmf (b' = b, g [x\ast y] (x * y) ∈ s')
  } ELSE do 
    b ← coin-spmf;
    return-spmf (b, g [x\ast y] (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 (λx. (yield (\{(\}_) \}, ro.initial)) (game2 x y) (game1 x y) for x y

proof =
\end{verbatim}

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

\begin{verbatim}
interpret inv-oracle': callee-invariant-on oracle' (λ(s, s-h), s = dom s-h). \{ split \}
  by unfold-locales(auto simp add: oracle'-def split: option.split-asm if-split)
interpret bad: callee-invariant-on oracle' (λ(s, z), z ∈ s). \{ split \}
  by unfold-locales(auto simp add: oracle'-def)
\end{verbatim}

First, we identify a bisimulation relation \(?X\) between the different states of \textit{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 \(g\ [x\ast y]\).

\begin{verbatim}
let ?X = λ(s, s-h) (s', s-h'). s = dom s-h ∧ s' = s ∧ s-h = s-h' (g [x\ast y] (x * y) := None)
\end{verbatim}

Then, we can prove that \(?X\) really is a bisimulation for \textit{oracle}' except when the failure event occurs. The next statement expresses this.

\begin{verbatim}
let ?bad = λ(s, s-h), g [x\ast y] \in s
let ?R = (\{ \}, \{ \}) \ast \{ \}\ast \{ \}\ast \{ \}
\end{verbatim}
if ?X s1 s2 for s1 s2 plain using that
by(auto split; prod.split intro!: rel-spmf-bind-reflI simp add: oracle'!-def rel-spmf-return-spmf2
fun-upd-twist split; option.split dest!: fun-upd-eqD)

have inv: callee-invariant oracle'?bad
— Once the failure event has happened, it will not be forgotten any more.
by(unfold-locales)(auto simp add: oracle'!-def split)

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

show ?thesis

unfolding game_1-def game_2-def
— Peel off the first phase of the game using the structural decomposition rules rel-spmf-bind-reflI
and rel-spmf-try-spmf.
apply(clarsimp intro!: rel-spmf-bind-reflI simp del: bind-spmf-const)
apply(rule rel-spmf-try-spmf)

subgoal TRY for b m1 m2 σ s s-h
apply(rule rel-spmf-bind-reflI)
— Exploit that in the first phase of the game, the set s of queried strings and the map
of the random oracle s-h are updated in lock step, i.e., s = dom s-h.
apply(drule inv-oracle'!.exec-gpv-invariant; clarsimp)
— Has the adversary queried the random oracle with g [^] (x * y) during the first phase?
apply(cases g [^] (x * y) ∈ 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 $I = \mathcal{I}$-full]$dest!:$ bad.exec-gpv-invariant)

subgoal False — Then let’s see whether the adversary queries g [^] (x * y) in the second
phase. Thanks to ro.fresh, the call to the random oracle simplifies to sampling a random
bitstring.
apply(clarsimp iff del: domIf simp add: domIf ro.fresh intro!: rel-spmf-bind-reflI)
apply(rule rel-spmf-bindI[where $R = ?R$])
— The lemma exec-gpv-oracle-bisim-bad-full lifts the bisimulation for oracle' to the
adversary $s_f^{2}$ interacting with oracle'.
apply(rule exec-gpv-oracle-bisim-bad-full[OF - - bisim inv inv])
apply(auto simp add: fun-upd-idem)
done

subgoal ELSE by(rule rel-spmf-reflI) clarsimp
done

qed

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

hence rel-spmf ($\lambda (win, bad) (win', bad'). (bad \leftrightarrow bad') \land (\neg bad' \rightarrow win \leftrightarrow win'$))
(?sample game_2) (?sample game_1)
by(intro rel-spmf-bind-reflI)
hence \[\text{measure}(\text{measure}\text{-}\text{spmf}(?\text{sample game}_2))\{(\text{win}, -), \text{win}\} - \text{measure}(\text{measure}\text{-}\text{spmf}(?\text{sample game}_1))\{(\text{win}, -), \text{win}\}]\leq \text{measure}(\text{measure}\text{-}\text{spmf}(?\text{sample game}_2))\{(-, \text{bad}), \text{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 \[\text{measure}(\text{measure}\text{-}\text{spmf}(?\text{sample game}_2))\{(\text{win}, -), \text{win}\} = \text{spmf}\{\text{map}\text{-}\text{spmf}\text{ fst}(?\text{sample game}_2)\}\text{True}\]
and \[\text{measure}(\text{measure}\text{-}\text{spmf}(?\text{sample game}_1))\{(\text{win}, -), \text{win}\} = \text{spmf}\{\text{map}\text{-}\text{spmf}\text{ fst}(?\text{sample game}_1)\}\text{True}\]
and \[\text{measure}(\text{measure}\text{-}\text{spmf}(?\text{sample game}_2))\{(-, \text{bad}), \text{bad}\} = \text{spmf}\{\text{map}\text{-}\text{spmf}\text{ snd}(?\text{sample game}_2)\}\text{True}\]
unfolding spmf\text{-}\text{con}\text{-}\text{measure}\text{-}\text{spmf} measure\text{-}\text{map}\text{-}\text{spmf} by(auto simp add: vimage-def split-def)
ultimately have hop12:
[spmf(\text{map}\text{-}\text{spmf}\text{ fst}(?\text{sample game}_2))\text{True} - \text{spmf}(\text{map}\text{-}\text{spmf}\text{ fst}(?\text{sample game}_1))\text{True}]\leq \text{spmf}(\text{map}\text{-}\text{spmf}\text{ snd}(?\text{sample game}_2))\text{True}\]
by simp

5.7 Optimistic sampling: the one-time-pad

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

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

define game\(_3\)::(bool ⇒ bitstring ⇒ bitstring ⇒ bitstring spmf) ⇒ nat ⇒ nat ⇒ (bool × bool) spmf
where game\(_3\)\(\text{f}\)\(\text{x}\)\(\text{y}\) = do {
  b ← coin\text{-}spmf;
  ((m\(_1\), m\(_2\)), (s, s\(-\)h)) ← exec\text{-}gpv oracle\text{'}(s\(_\text{f}_1\)(g[^]{})\text{x}) (\{\}, ro\text{-}\text{initial});
  TRY do {
    :: unit ← assert\text{-}spmf (valid\text{-}plains m\(_1\) m\(_2\));
    h\(_\text{'}\) ← f b m\(_1\) m\(_2\);
    let cipher = (g[^]{}) y, h\(_\text{'}\);
    (b\(_\text{'}\), (s\(_\text{'}\), s\(-\)h\(_\text{'}\))) ← exec\text{-}gpv oracle\text{'}(s\(_\text{f}_2\) cipher σ) (s, s\(-\)h);
return-spmf \( (b' = b, g \in \{x \ast y\} \in s') \)
} ELSE do {
  b ← coin-spmf;
  return-spmf \( (b, g \in \{x \ast y\} \in s) \)
}

for \( f x y \)

Clearly, if we plug in the appropriate function \( ?f \), then we get \( \text{game}_2 \):

\[
\text{let } \lambda b m_1 m_2. \text{map-spmf} \ (\lambda h. (\text{if } b \text{ then } m_1 \text{ else } m_2) [\oplus] h) \ (\text{spmf-of-set} (\text{nlists } \text{UNIV } \text{len-plain}))
\]

\[
\text{have } \text{game}_2 \cdot \text{game}_3 \colon \text{game}_2 x y = \text{game}_3 ?f x y \text{ for } x y
\]

 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 \( \text{try-spmf-cong} \ \text{bind-spmf-cong} [\text{OF refl}] \ \text{if-cong} [\text{OF refl}] \) extract this information from the program of the game.

\[
\text{let } ?f' = \lambda b m_1 m_2. \text{spmf-of-set} (\text{nlists } \text{UNIV } \text{len-plain})
\]

\[
\text{have } \text{game}_3 \cdot ?f \colon \text{game}_3 ?f' x y \text{ for } x y
\]

The rest of the proof consists of simplifying \( \text{game}_3 ?f' \). 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 \( \text{game}_2 \) or \( \text{game}_3 \) any more because this information is already stored in the random oracle’s state. So we change the \( \text{oracle}' \) back into \( \text{oracle} \) using the Transfer package. With a bit of rewriting, the result is then the \( \text{game} \) for the adversary elgamal-reduction \( \alpha' \). Moreover, the guess \( b' \) of the adversary is independent of \( b \) in \( \text{game}_3 \ ?f \), so the first boolean returned by \( \text{game}_3 ?f' \) is just a coin flip.

\[
\text{have } \text{game}_3 \cdot \text{bad} \colon \text{map-spmf} \ \text{snd} \ (\text{?sample} (\text{game}_3 ?f')) = \text{lcdh.game} (\text{elgamal-reduction} \alpha')
\]

\[
\text{have } \text{game}_3 \cdot \text{guess} \colon \text{map-spmf} \ \text{fst} \ (\text{?sample} (\text{game}_3 ?f')) = \text{coin-spmf} \text{ for } x y
\]

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.

\[
\text{have } \text{ind-cpa} \cdot \text{advantage} (\text{ro.oracle}, \text{ro.initial}) \alpha' = \text{spmf} (\text{map-spmf} \ \text{fst} (\text{?sample} \ \text{game}_1))
\]

\[
\text{True} = \frac{1}{2}
\]

\text{using } \text{ind-cpa-game-eq-game}_0 \text{ by}(\text{simp add: } \text{game}_0 \cdot \text{game}_1 \text{ o-def})
also have \( |1/2 - \text{spmf} (\text{map-spmf} \text{fst (?sample game}_1)) \text{True} | \)
by (simp add: abs-minus-commute)
also have \( 1/2 = \text{spmf} (\text{map-spmf} \text{fst (?sample game}_2)) \text{True} \)
by (simp add: game2-game3 game3 o-def game3-guess spmf-of-set)
also have \( |\ldots - \text{spmf} (\text{map-spmf} \text{fst (?sample game}_1)) \text{True} | \leq \text{spmf} (\text{map-spmf} \text{snd (?sample game}_2)) \text{True} \)
by (rule hop12)
also have \( \ldots = \text{lcdh}. \text{advantage} (\text{elgamal-reduction} \mathcal{A}) \)
by (simp add: game2-game3 game3 game3-bad lcdh.advantage-def o-def del: map-bind-spmf)
finally show \(?\text{thesis} \).
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 \Rightarrow ('pubkey \times 'privkey, 'query, 'response) gpv
and encrypt :: security \Rightarrow 'pubkey \Rightarrow 'plain \Rightarrow ('cipher, 'query, 'response) gpv
and decrypt :: security \Rightarrow 'privkey \Rightarrow 'cipher \Rightarrow ('plain, 'query, 'response) gpv
and valid-plains :: security \Rightarrow 'plain \Rightarrow 'plain \Rightarrow 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' =
\text{lcdh}: \text{list-cdh}'(\mathcal{G} + \)
\( \text{ro: random-oracle } \) len-plain
\( \text{for } G :: \text{security } \Rightarrow \text{grp cyclic-group} \)
\( \text{and len-plain :: security } \Rightarrow \text{nat} \)
begin
sublocale hashed-elgamal \( G \eta \) 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` (\( \lambda \eta. \text{lcdh.advantage } \eta (\text{elgamal-reduction } \eta (\emptyset \eta)) \))
**and** `eventually` (\( \lambda \eta. \text{ind-cpa.lossless } (\emptyset \eta) \)) at-top
**Shows** `negligible` (\( \lambda \eta. \text{ind-cpa.advantage } \eta (\text{ro.oracle } \eta, \text{ro.initial}) (\emptyset \eta) \))

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

**References**


