Game-based cryptography in HOL
Andreas Lochbihler and S. Reza Sefidgar and Bhargav Bhatt

June 11, 2019

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 LDDH 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 guesses</td>
<td>20</td>
</tr>
<tr>
<td>1.13</td>
<td>Unpredictable function</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Cryptographic constructions and their security</td>
<td>23</td>
</tr>
<tr>
<td>2.1</td>
<td>Elgamal encryption scheme</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>Hashed Elgamal in the Random Oracle Model</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>The random-permutation random-function switching lemma</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>Extending the input length of a PRF using a universal hash function</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>IND-CPA from PRF</td>
<td>32</td>
</tr>
<tr>
<td>2.6</td>
<td>IND-CCA from a PRF and an unpredictable function</td>
<td>34</td>
</tr>
<tr>
<td>A</td>
<td>Tutorial Introduction to CryptHOL</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>Introduction</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>Getting started</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Getting started</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>Modelling cryptography using CryptHOL</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Security notions without oracles: the CDH assumption</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>A Random Oracle</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Cryptographic concepts: public-key encryption</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Security notions with oracles: IND-CPA security</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Concrete cryptographic constructions: the hashed ElGamal encryption</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>Cryptographic proofs in CryptHOL</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>The reduction</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Concrete security statement</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Recording adversary queries</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Equational program transformations</td>
<td>60</td>
</tr>
<tr>
<td>5.5</td>
<td>Capturing a failure event</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>Game hop based on a failure event</td>
<td>63</td>
</tr>
<tr>
<td>5.7</td>
<td>Optimistic sampling: the one-time-pad</td>
<td>66</td>
</tr>
<tr>
<td>5.8</td>
<td>Combining several game hops</td>
<td>67</td>
</tr>
<tr>
<td>6</td>
<td>Asymptotic security</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Introducing a security parameter</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Asymptotic security statements</td>
<td>69</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)
  ⟨proof⟩

  lemma lossless-ddh-1:
  [ lossless A; 0 < order G ]
  ⇒ lossless-spmf (ddh-1 A)
  ⟨proof⟩
1.2 The LCDH game

locale lcdh = 
  fixes G :: 'grp cyclic-group (structure)
begin

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

definition lcdh :: 'grp adversary ⇒ bool spmf
where lcdh ⦃ ⦄ = do { 
  x ← sample-uniform (order ⦃ ⦄); 
  y ← sample-uniform (order ⦃ ⦄); 
  zs ← ⦃ g [^] x ⦄ ⦃ g [^] y ⦄; 
  return-spmf (g [^] (x ∗ y) ∈ zs)
}

definition advantage :: 'grp adversary ⇒ real
where advantage ⦃ ⦄ = spmf (lcdh ⦃ ⦄) True

definition lossless :: 'grp adversary ⇒ bool
where lossless ⦃ ⦄ = (∀ ⦁ ⦂. lossless-spmf (⦃ ⦁ ⦂ ⦄))

lemma lossless-lcdh: 
  [ lossless ⦃ ⦄; 0 < order ⦃ ⦄ ] 
  ⇒ lossless-spmf (lcdh ⦃ ⦄)
⟨proof⟩
end

end

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

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

1.3 The IND-CCA2 game for public-key encryption

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

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

begin

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

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, cstags)) cipher = return-spmf
  (if cipher ∈ set cstags then None else decrypt η dkey cipher, Some (ekey, dkey, cstags))

fun ekey-oracle :: security ⇒ ('ekey', 'dkey', 'cipher') state-oracle ⇒ unit ⇒ ('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)))

⟨proof⟩

context notes bind-spmf-cong|fundef-cong| begin

function encrypt-oracle
  :: bool ⇒ security ⇒ ('ekey', 'dkey', 'cipher') state-oracle ⇒ 'cipher
  ⇒ ('plain × ('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, cstags)) (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 # cstags))
  ) else return-pmf None

⟨proof⟩

termination (proof)
1.3.1 Single-user setting

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

definition oracle1 :: bool ⇒ security
⇒ (('ekey', 'dkey', 'cipher') state-oracle, ('plain', 'cipher') call1, ('ekey', 'plain', 'cipher') ret1) oracle'
where oracle1 b η = ekey-oracle η ⊕₀ (decrypt-oracle η ⊕₀ 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 o Inl)) (decrypt-oracle η s y)
oracle1 b η s (Inr (Inr z)) = map-spmf (apfst (Inr o Inl)) (encrypt-oracle b η s z)
〈proof〉

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

definition ind-cca21 :: ('ekey', 'plain', 'cipher') adversary1 ⇒ security ⇒ bool spmf
where
ind-cca21 \( \not S \) η = TRY do {
  b ← coin-spmf;
  (guess, s) ← exec-gpv (oracle1 b η) (\( \not S \) η) None;
  return-spmf (guess = b)
} ELSE coin-spmf

definition advantage1 :: ('ekey', 'plain', 'cipher') adversary1 ⇒ advantage
where advantage1 \( \not S \) η = |spmf (ind-cca21 \( \not S \) η) True - 1/2|

lemma advantage1-nonneg: advantage1 \( \not S \) η ≥ 0 〈proof〉

abbreviation secure-for1 :: ('ekey', 'plain', 'cipher') adversary1 ⇒ bool
where secure-for1 \( \not S \) ≡ negligible (advantage1 \( \not S \) )

definition ibounded-by1 :: ('ekey', 'plain', 'cipher') adversary1 ⇒ nat ⇒ bool
where ibounded-by1 \( \not S \) q = interaction-any-bounded-by \( \not S \) q

abbreviation ibounded-by1 :: ('ekey', 'plain', 'cipher') adversary1 ⇒ (security ⇒ nat) ⇒ bool
where ibounded-by1 \( \not S \) ≡ rel-envir ibounded-by1'

definition lossless1 :: ('ekey', 'plain', 'cipher') adversary1 ⇒ bool
where lossless1 \( \not S \) = lossless-gpv \( \not \) full \( \not S \)
abbreviation \textit{lossless}_1 :: (\textit{ekey}, \textit{plain}, \textit{cipher}) \text{adversary}_1 \Rightarrow \text{bool}
where \textit{lossless}_1 \equiv \text{pred-envir lossless}_1^	ext{'}

lemma \textit{lossless-decrypt-oracle} [\text{simp}]: \textit{lossless-spmf} (\text{decrypt-oracle} \eta \sigma \eta \text{cipher})
(proof)

lemma \textit{lossless-ekey-oracle} [\text{simp}]:
\textit{lossless-spmf} (\textit{ekey-oracle} \eta \sigma \eta \text{cipher}) \longleftrightarrow (\sigma = \text{None} \longrightarrow \text{lossless-spmf} (\text{key-gen} \eta))
(proof)

lemma \textit{lossless-encrypt-oracle} [\text{simp}]:
\begin{align*}
\text{\textit{lossless-spmf}} (\text{\textit{ekey-oracle}} \eta \sigma (\eta, m)) & \iff \text{\textit{valid-plain}} m \land \text{valid-plain} \eta \eta \text{cipher} \\
\text{\textit{lossless-spmf}} (\text{\textit{ekey-oracle}} (\eta \eta \text{cipher}, b) \eta (\eta, m)) & \iff \text{\textit{valid-plain}} m \land \text{valid-plain} \eta \eta \text{cipher} \\
\text{\textit{lossless-spmf}} (\text{\textit{ekey-oracle}} (\eta \eta \text{cipher}, \text{\textit{state-oracle}} (\eta \eta \text{cipher}, \textit{advantage})) \eta \eta \text{cipher}) & \iff \text{\textit{valid-plain}} m \land \text{valid-plain} \eta \eta \text{cipher} \\
\text{\textit{lossless-spmf}} (\text{\textit{ekey-oracle}} (\eta \eta \text{cipher}, \text{\textit{advantage-apply}} (\eta \eta \text{cipher}, \textit{advantage})) \eta \eta \text{cipher}) & \iff \text{\textit{valid-plain}} m \land \text{valid-plain} \eta \eta \text{cipher} \\
\end{align*}
(proof)

1.3.2 Multi-user setting

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

lemma \textit{oracle}_n - apply [\text{simp}]:
\text{\textit{oracle}}_n b \eta s (i, x) = \text{\textit{map-spmf}} (\text{\textit{apsnd}} (\text{\textit{fun-upd}} s i)) (\text{\textit{oracle}}_1 b \eta (s i) x)
(proof)

type-synonym (\textit{i}, \textit{\textit{ekey}}, \textit{\textit{plain}}, \textit{\textit{cipher}) adversary}_n =
(\text{\textit{\textit{bool}}, \textit{i} \times (\textit{\textit{\textit{plain}}, \textit{\textit{cipher}) call}_1, (\textit{\textit{ekey}}, \textit{\textit{\textit{plain}}, \textit{\textit{cipher}) ret}_1) \text{ gpv}})

type-synonym (\textit{i}, \textit{\textit{ekey}}, \textit{\textit{plain}}, \textit{\textit{cipher}) adversary}_n =
\text{\textit{security}} \Rightarrow (\textit{i}, \textit{\textit{ekey}}, \textit{\textit{plain}}, \textit{\textit{cipher}) adversary}_n)

definition \textit{ind-cca}_2 \equiv (\textit{i}, \textit{\textit{ekey}}, \textit{\textit{plain}}, \textit{\textit{cipher}) adversary}_n \Rightarrow \text{\textit{security}} \Rightarrow \text{\textit{bool}} \text{ spmf}
where
\textit{\textit{ind-cca}}_2 \equiv \eta = \text{\textit{TRY}} \text{ do }
\begin{align*}
\text{\textit{b}} & \leftarrow \text{\textit{coin-spmf}}; \\
(\text{\textit{\textit{guess}}, \textit{\textit{\textit{sigma}}}}) & \leftarrow \text{\textit{exec-gpv}} (\text{\textit{\textit{oracle}}}_n b \eta) (\equiv \eta) (\lambda-. \text{\textit{\textit{None}}}); \\
\text{\textit{return-spmf}} (\text{\textit{\textit{\textit{guess}}} = \text{\textit{\textit{b}}}}) \\
\text{\textit{ELSE}} \text{\textit{coin-spmf}}
\end{align*}
\text{\textit{definition}}\textit{\textit{advantage}}_n :: (\textit{i}, \textit{\textit{\textit{ekey}}, \textit{\textit{plain}}, \textit{\textit{cipher}}) \text{\textit{adversary}}_n \Rightarrow \textit{\textit{advantage}}
where\textit{\textit{\textit{advantage}}}_n \equiv \eta = \text{\textit{\textit{spmf}} (\textit{\textit{\textit{\textit{ind-cca}}}_2 \equiv \eta}) \text{\textit{\textit{True}} - \textit{1/2}}
\text{\textit{abbreviation}} \textit{\textit{secure-for}}_n :: (\textit{i}, \textit{\textit{\textit{ekey}}, \textit{\textit{plain}}, \textit{\textit{cipher}) adversary}_n \Rightarrow \textit{\textit{bool}}
\text{\textit{where}} \textit{\textit{secure-for}}_n \equiv \eta \equiv \text{\textit{negligible}} (\textit{\textit{advantage}}_n \equiv \eta)
definition ibounded-by \( n \) :: \((i, 'ekey, 'plain, 'cipher) adversary_n \) ⇒ nat ⇒ bool
where ibounded-by \( n \) = interaction-any-bounded-by \( n \)

abbreviation ibounded-by \( n \) :: \((i, 'ekey, 'plain, 'cipher) adversary_n \) ⇒ bool
where ibounded-by \( n \) = rel-envir ibounded-by \( n \)

definition lossless \( n \) :: \((i, 'ekey, 'plain, 'cipher) adversary \) ⇒ bool
where lossless \( n \) = lossless-gpv I-full \( n \)

abbreviation lossless \( n \) :: \((i, 'ekey, 'plain, 'cipher) adversary \) ⇒ bool
where lossless \( n \) = pred-envir lossless \( n \)

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

lemma cipher-queriesI:
\[
\text{ose n} = \text{Some (ek, dk, ciphers)}; x \in \text{set ciphers} \quad \Rightarrow \quad x \in \text{cipher-queries ose}
\]
(proof)

lemma cipher-queriesE:
assumes \( x \in \text{cipher-queries ose} \)
obtains (cipher-queries) \( n ek dk ciphers \) where \( ose n = \text{Some (ek, dk, ciphers)} \) \( x \in \text{set ciphers} \)
(proof)

lemma cipher-queries-updE:
assumes \( x \in \text{cipher-queries (ose(\( n \mapsto (ek, dk, ciphers)\)))} \)
obtains (old) \( x \in \text{cipher-queries ose x} \notin \text{set ciphers} \) | (new) \( x \in \text{set ciphers} \)
(proof)

lemma cipher-queries-empty [simp]: cipher-queries Map.empty = {}
(proof)

end

end

1.4 The IND-CCA2 security for symmetric encryption schemes

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

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

begin

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

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

lemma lossless-oracle-encrypt [simp]:
  assumes lossless-spmf (encrypt k m1) and lossless-spmf (encrypt k m0)
  shows lossless-spmf (oracle-encrypt k b L (m1, m0))

(proof)

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)
(proof)

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

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

lemma advantage-nonneg: 0 ≤ advantage ω (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end

definition game-advantage :: ('message, 'cipher) adversary ⇒ real
where
  game-advantage ω = advantage (game ω)

lemma game-advantage-monotone: game-advantage ω ≤ game-advantage ω' (proof)

end
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 σ = do 
  let (σ1, σ2) = σ;
  key ← key-gen;
  b ← coin-spmf;
  (guess, -) ← exec-gpv (encrypt-oracle key) (do 
    ((m0, m1), σ) ← σ1;
    if valid-plain m0 ∧ valid-plain m1 then do 
      cipher ← lift-spmf (encrypt key (if b then m0 else m1));
      σ2 cipher σ
    } else lift-spmf coin-spmf
  );
  return-spmf (guess = b)
definition advantage :: ('plain', 'cipher', 'state) adversary ⇒ real 
where advantage σ = |spmfs (ind-cpa σ) True − 1/2|
lemma advantage-nonneg: advantage σ ≥ 0 (proof)
definition ibounded-by :: ('plain, 'cipher, 'state) adversary ⇒ enat ⇒ bool
where
ibounded-by = (λ(′1, ′2) q.
(∃q1 q2. interaction-any-bounded-by ′1 q1 ∧ (∀cipher σ. interaction-any-bounded-by (′2 cipher σ) q2 ∧ q1 + q2 ≤ q))
)

lemma ibounded-byE [consumes 1, case-names ibounded-by, elim?]:
assumes ibounded-by (′1, ′2) q
obtains q1 q2
where q1 + q2 ≤ q
and interaction-any-bounded-by ′1 q1
and (∃cipher σ. interaction-any-bounded-by (′2 cipher σ) q2)
⟨proof⟩

lemma ibounded-byI [intro?]:
[ interaction-any-bounded-by ′1 q1; (∃cipher σ. interaction-any-bounded-by (′2 cipher σ) q2; q1 + q2 ≤ q) ]
⇒ ibounded-by (′1, ′2) q
⟨proof⟩

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

end
end

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

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

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

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

**type-synonym**

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

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

**where**

\text{ind-cpa} (\not\emptyset, \not\emptyset) = \text{TRY do}
\{(pk, sk) \leftarrow \text{key-gen};
\quad b \leftarrow \text{lift-spmf coin-spmf};
\quad ((m0, m1), \sigma) \leftarrow (\not\emptyset \text{ pk});
\quad \text{assert-gpv (valid-plains } m0 \text{ m1});
\quad \text{cipher} \leftarrow \text{aencrypt } pk \text{ if } b \text{ then } m0 \text{ else } m1;
\quad \text{guess} \leftarrow \not\emptyset 2 \text{ cipher } \sigma;
\quad \text{Done (guess } = b\}
\} \text{ ELSE lift-spmf coin-spmf}

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

**where** \text{advantage oracle } \sigma \not\emptyset = \text{spmf (run-gpv oracle (ind-cpa } \not\emptyset) \sigma) \text{ True } - 1/2

**lemma** \text{advantage-nonneg}: \text{advantage oracle } \sigma \not\emptyset \geq 0 \text{ (proof)}

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

**where**

\text{ibounded-by consider} = (\lambda (\not\emptyset, \not\emptyset) \text{ q}).
(\exists q1 \text{ q2}. (\forall pk. \text{interaction-bounded-by consider} (\not\emptyset \text{ pk}) q1) \land (\forall \text{cipher } \sigma. \text{interaction-bounded-by consider} (\not\emptyset 2 \text{ cipher } \sigma) q2) \land q1 + q2 \leq q)

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

**assumes** \text{ibounded-by consider} (\not\emptyset, \not\emptyset) q

**obtains** q1 \text{ q2}

**where** q1 + q2 \leq q

**and** \text{pk. interaction-bounded-by consider} (\not\emptyset \text{ pk}) q1

**and** \text{cipher } \sigma. \text{interaction-bounded-by consider} (\not\emptyset 2 \text{ cipher } \sigma) q2

**proof**

**lemma** \text{ibounded-by'I [intro?]}:

\[ \text{pk. interaction-bounded-by consider} (\not\emptyset \text{ pk}) q1; \text{cipher } \sigma. \text{interaction-bounded-by consider} (\not\emptyset 2 \text{ cipher } \sigma) q2; q1 + q2 \leq q \]

\Rightarrow \text{ibounded-by consider} (\not\emptyset, \not\emptyset) q

**proof**

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

**where** \text{lossless} = (\lambda (\not\emptyset, \not\emptyset). (\forall pk. \text{lossless-gpv } \not\emptyset \text{-full} (\not\emptyset \text{ pk}) ) \land (\forall \text{cipher } \sigma. \text{lossless-gpv } \not\emptyset \text{-full} (\not\emptyset 2 \text{ cipher } \sigma) ))
theory IND-CPA-PK-Single

imports
  CryptHOL.Computational-Model

begin

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

declare ind-cpa.simps [simp del]

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

definition lossless :: ('pub-key, 'plain, 'cipher, 'state) adversary ⇒ bool
where
  lossless $\mathcal{A} ≜$
  $\forall pk. \text{lossless-spmf (fst }\mathcal{A} pk) \land$
  $\forall\text{cipher }\sigma. \text{lossless-spmf (snd }\mathcal{A} \text{ cipher }\sigma))$
lemma lossless-ind-cpa:

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

⟨proof⟩

end

end

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

1.8 Strongly existentially unforgeable signature scheme

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

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

begin

type-synonym (′vkey, ′sigkey, ′message, ′signature) state-oracle
= (′vkey × ′sigkey × (′message × ′signature) list) option

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

context notes bind-spmf-cong|fundef-cong| begin

function sign-oracle
:: security ⇒ ((′vkey, ′sigkey, ′message, ′signature) state-oracle, ′message, ′signature) oracle'

14
where

\begin{align*}
\text{sign-oracle} \eta \text{ None } m & = \text{ do } (\cdot, \sigma) \leftarrow \text{vkey-oracle } \eta \text{ None } (); \text{sign-oracle } \eta \sigma m \} \\
\text{\&log. sign-oracle } \eta \text{ (Some vkey, skey, log)) } m & = \\
\text{(if valid-message } \eta \text{ m then do} \\
\text{sig } \leftarrow \text{ sign } \eta \text{ skey m; } \\
\text{return-spmf (sig, Some (vkey, skey, (m, sig) # log))} \\
\text{) else return-pmf None) }
\end{align*}

\text{⟨proof⟩}

\text{termination (proof)}

\text{end}

\text{lemma lossless-vkey-oracle [simp]}:

\text{lossless-spmf (vkey-oracle } \eta \sigma x \rangle \leftarrow \sigma = \text{ None } \rightarrow \text{lossless-spmf (key-gen } \eta))

\text{⟨proof⟩}

\text{lemma lossless-sign-oracle [simp]}:

\begin{align*}
\sigma = \text{ None } & \Rightarrow \text{lossless-spmf (key-gen } \eta); \\
\text{\&skey m. valid-message } \eta \text{ m } & \Rightarrow \text{lossless-spmf (sign } \eta \text{ skey m)} \\
& \Rightarrow \text{lossless-spmf (sign-oracle } \eta \sigma m) \leftarrow \text{valid-message } \eta \text{ m}
\end{align*}

\text{⟨proof⟩}

\text{lemma lossless-sign-oracle-Some: fixes log shows}

\text{lossless-spmf (sign-oracle } \eta \text{ (Some (vkey, skey, log)) m) } \leftarrow \text{lossless-spmf (sign } \eta \text{ skey m) } \land \text{valid-message } \eta \text{ m}

\text{⟨proof⟩}

\subsection{1.8.1 Single-user setting}

\text{type-synonym } \text{'message' call} \text{1 = unit + 'message'}

\text{type-synonym } \text{('vkey', 'signature') ret} \text{1 = 'vkey' + 'signature'}

\text{definition oracle} \text{1 :: security}

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

\text{where oracle} \text{1 } \eta = \text{vkey-oracle } \eta \oplus_{O} \text{sign-oracle } \eta

\text{lemma oracle} \text{1-simps [simp]}:

\text{oracle} \text{1 } \eta \text{ s (Inl } x \text{)} = \text{map-spmf (apfst Inl) (vkey-oracle } \eta \text{ s x)}

\text{oracle} \text{1 } \eta \text{ s (Inr } y \text{)} = \text{map-spmf (apfst Inr) (sign-oracle } \eta \text{ s y)}

\text{⟨proof⟩}

\text{type-synonym } (('vkey', 'message', 'signature') adversary} \text{1}' =

\text{((message } \times \text{ 'signature'), 'message' call1, ('vkey', 'signature') ret1) gpv}

\text{type-synonym } (('vkey', 'message', 'signature') adversary} \text{1} =

\text{security } \Rightarrow (('vkey', 'message', 'signature') adversary} \text{1'}

\text{definition suf-cma} \text{1 :: ('vkey', 'message', 'signature') adversary} \text{1 } \Rightarrow \text{security } \Rightarrow \text{bool spmf}

\text{where}

\text{\&log. suf-cma} \text{1 } \eta = \text{ do } \{
\[(m, \text{ sig}) \mapsto \text{exec-gpv} (\text{oracle}_1 \eta) (\not\in \eta) \text{ None};\]
\[
\text{return-spmf} (\text{case } \sigma \text{ of } \text{None } \Rightarrow \text{False})
\]
\[
| \text{Some } (\text{vkey}, \text{skey}, \text{log}) \Rightarrow \text{verify } \eta \text{ vkey } m \text{ sig } (m, \text{ sig}) \not\in \text{set } \log) \}
\]

definition \text{advantage}_1 :: (\text{vkey}, \text{message}, \text{signature}) \text{adversary}_1 \Rightarrow \text{advantage}
\where \text{advantage}_1 \not\in \eta = \text{spmff} (\text{suf-cma}_1 \not\in \eta) \text{ True}
\]

lemma \text{advantage}_1-	ext{nonneg}; \text{advantage}_1 \not\in \eta \geq 0 \langle \text{proof} \rangle

abbreviation \text{secure-for}_1 :: (\text{vkey}, \text{message}, \text{signature}) \text{adversary}_1 \Rightarrow \text{bool}
\where \text{secure-for}_1 \not\in \eta \equiv \text{negligible} (\text{advantage}_1 \not\in \eta)
\]

definition \text{ibounded-by}_1 :: (\text{vkey}, \text{message}, \text{signature}) \text{adversary}_1 \Rightarrow \text{nat} \Rightarrow \text{bool}
\where \text{ibounded-by}_1 \not\in \eta q = (\text{interaction}-\text{any-bound-by} \not\in \eta q)
\]

abbreviation \text{ibounded-by}_1 :: (\text{vkey}, \text{message}, \text{signature}) \text{adversary}_1 \Rightarrow (\text{security} \Rightarrow \text{nat})
\Rightarrow \text{bool}
\where \text{ibounded-by}_1 \not\in \eta \equiv \text{rel-environ} \text{ibounded-by}_1'
\]

definition \text{lossless}_1 :: (\text{vkey}, \text{message}, \text{signature}) \text{adversary}_1 ' \Rightarrow \text{bool}
\where \text{lossless}_1 ' \not\in \eta = (\text{lossless-gpv} \not\in \text{full } \not\in \eta)
\]

abbreviation \text{lossless}_1 :: (\text{vkey}, \text{message}, \text{signature}) \text{adversary}_1 \Rightarrow \text{bool}
\where \text{lossless}_1 \not\in \eta \equiv \text{pred-environ} \text{lossless}_1'
\]

1.8.2 Multi-user setting

\begin{definition}
\text{oracle}_n :: \text{security}
\Rightarrow (i \Rightarrow (\text{vkey}, \text{sigkey}, \text{message}, \text{signature}) \text{state-oracle}, i \times \text{message call}_1, (\text{vkey}, \text{'signature'} \text{ ret}_1) \text{oracle}')
\where \text{oracle}_n \eta = \text{family-oracle} (\lambda -. \text{oracle}_1 \eta)
\end{definition}

lemma \text{oracle}_n-apply [simp]:
\text{oracle}_n \eta s (i, x) = \text{map-spmff} (\text{apsnd} (\text{fun-upd } s i)) (\text{oracle}_1 \eta (s i) x)
\langle \text{proof} \rangle
\]

\begin{type-synonym}
\text{('i, 'vkey, 'message', 'signature') \text{adversary}_n}' =
\text{('i \times \text{message}' \times \text{'signature'}), 'i \times \text{message}' \text{call}_1, ('\text{vkey}', 'signature' \text{ ret}_1) \text{gpv}}
\end{type-synonym}

\begin{type-synonym}
\text{('i, 'vkey, 'message', 'signature') \text{adversary}_n} =
\text{security} \Rightarrow (i, 'vkey', 'message', 'signature') \text{adversary}_n '
\end{type-synonym}

\begin{definition}
\text{suf-cma}_n :: (i, 'vkey', 'message', 'signature') \text{adversary}_n \Rightarrow \text{security} \Rightarrow \text{bool} \text{ spmf}
\where
\text{\lambda log. suf-cma}_n \not\in \eta = \text{do}
\text{\langle (i, m, \text{ sig}), \sigma \mapsto \text{exec-gpv} (\text{oracle}_n \eta) (\not\in \eta) (\lambda -. \text{None});
\text{return-spmff} (\text{case } \sigma i \text{ of } \text{None } \Rightarrow \text{False} \rangle}
\end{definition}
Some \((vkey, skey, log) \Rightarrow \text{verify } \eta \text{ vkey } m \text{ sig } \land (m, \text{sig}) \notin \text{set log}) \)

**Definition** advantage_n :: \(('i, 'vkey, 'message, 'signature) \text{ adversary}_n \Rightarrow \text{advantage} \)

**Where** advantage_n \(\not\in \eta = \text{spmf } (\text{sup-cma}_n \not\in \eta) \text{ True} \)

**Lemma** advantage_n-nonneg: advantage_n \(\not\in \eta \geq 0 \langle \text{proof} \rangle \)

**Abbreviation** secure-for_n :: \(('i, 'vkey, 'message, 'signature) \text{ adversary}_n \Rightarrow \text{bool} \)

**Where** secure-for_n \(\not\in \eta \equiv \text{negligible } (\text{advantage}_n \not\in \eta) \)

**Definition** ibounded-by_n' :: \(('i, 'vkey, 'message, 'signature) \text{ adversary}_n' \Rightarrow \text{nat} \Rightarrow \text{bool} \)

**Where** ibounded-by_n' \(\not\in q = (\text{interaction-any-bounded-by } \not\in q) \)

**Abbreviation** ibounded-by_n :: \(('i, 'vkey, 'message, 'signature) \text{ adversary}_n \Rightarrow (\text{security } \Rightarrow \text{nat}) \Rightarrow \text{bool} \)

**Where** ibounded-by_n \(\equiv \text{rel-envir } \text{ibounded-by}_n' \)

**Definition** lossless_n' :: \(('i, 'vkey, 'message, 'signature) \text{ adversary}_n' \Rightarrow \text{bool} \)

**Where** lossless_n' \(\not\in \equiv (\text{lossless-gpv } \text{I-full } \not\in) \)

**Abbreviation** lossless_n :: \(('i, 'vkey, 'message, 'signature) \text{ adversary}_n \Rightarrow \text{bool} \)

**Where** lossless_n \(\equiv \text{pred-envir } \text{lossless}_n' \)

**End**

**Theory** Pseudo-Random-Function

**Imports** CryptHOL.Computational-Model

**Begin**

### 1.9 Pseudo-random function

**Locale** random-function =

**Fixes** \(p :: 'a \text{ spmf} \)

**Begin**

**Type-synonym** \(('b,'a) \text{ dict } = 'b \Rightarrow 'a' \)

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

**Where**

random-oracle \(\sigma x = \)

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

**Definition** forgetful-random-oracle :: \(\text{unit } \Rightarrow 'b \Rightarrow ('a \times \text{unit}) \text{ spmf} \)

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

**lemma** weight-random-oracle [simp]:
weight-spm$ f$ $p = 1 \implies$ weight-spm$ f$ (random-oracle $\sigma x) = 1$
⟨proof⟩

**lemma** lossless-random-oracle [simp]:
lossless-spm$ f$ $p \implies$ lossless-spm$ f$ (random-oracle $\sigma x)$
⟨proof⟩

**sublocale** finite: callee-invariant-on random-oracle $\lambda \sigma. \text{finite} (\text{dom } \sigma)$ $\mathcal{F}$-full
⟨proof⟩

**lemma** card-dom-random-oracle:
assumes interaction-any-bounded-by $\not\not q$
and $(y, \sigma') \in \text{set-spm} f$ (exec-gpv random-oracle $\not\not \sigma$)
and fin: finite (dom $\sigma$)
shows card (dom $\sigma'$) $\leq q +$ card (dom $\sigma$)
⟨proof⟩
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 ⟨proof⟩

**definition** prf-oracle :: ′key $\Rightarrow$ unit $\Rightarrow$ ′domain $\Rightarrow$ (′range $\times$ unit) spmf
where prf-oracle key $\sigma$ x = return-spm$ f$ (prf key x, ())

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

**definition** game-0 :: (′domain, ′range) adversary $\Rightarrow$ bool spmf
where
game-0 $\not\not$ = do {
  key $\leftarrow$ key-gen;
  (b, -) $\leftarrow$ exec-gpv (prf-oracle key) $\not\not$ ();
  return-spm$ f$ b
}

**definition** game-1 :: (′domain, ′range) adversary $\Rightarrow$ bool spmf
where
game-1 $\not\not$ = do {
  (b, -) $\leftarrow$ exec-gpv random-oracle $\not\not$ Map.empty;
\begin{verbatim}
    return-spmf b

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

lemma advantage-nonneg: advantage $A \geq 0
(proof)

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 $\sigma$ $x$ =
\begin{cases}
    \text{case } $\sigma$ $x$ of Some $y$ ⇒ return-spmf ($y$, $\sigma$) \\
    None ⇒ spmf-of-set ($A - \text{ran } $\sigma$) \gg= (\lambda y. \text{return-spmf } (y, \sigma(x \mapsto y)))
\end{cases}

lemma weight-random-oracle [simp]:
\[ \text{finite } A; A - \text{ran } \sigma \neq \{\} \implies \text{weight-spmf } (\text{random-permutation } \sigma x) = 1 \]
(proof)

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

sublocale finite: callee-invariant-on random-permutation $\lambda \sigma. \text{finite } (\text{dom } \sigma) \; $I$-full
(proof)

lemma card-dom-random-oracle:
assumes interaction-any-bounded-by $A q$
and ($y$, $\sigma'$) ∈ set-spmf (exec-gpv random-permutation $A$ $\sigma$
\end{verbatim}
and fin: finite (dom σ)
shows card (dom σ') ≤ q + card (dom σ)
⟨proof⟩
end
end

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

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

locale guessing-many-one =
  fixes init :: ('c-o × 'c-a × 's) spmf
  and oracle :: 'c-o ⇒ 's ⇒ 'call ⇒ ('ret × 's) spmf
  and eval :: 'c-o ⇒ 'c-a ⇒ 's ⇒ 'guess ⇒ bool spmf
begin

type-synonym ('c-a', 'guess', 'call', 'ret) adversary-single = 'c-a' ⇒ ('guess', 'call', 'ret')
gpv
definition game-single :: ('c-a', 'guess', 'call', 'ret) adversary-single ⇒ bool spmf
where
game-single A =
do {
  (c-o, c-a, s) ← init;
  (guess, s') ← exec-gpv (oracle c-o) (A c-a) s;
  eval c-o c-a s' guess
}
definition advantage-single :: ('c-a', 'guess', 'call', 'ret) adversary-single ⇒ real
where
advantage-single A = spmf (game-single A) True


type-synonym ('c-a', 'guess', 'call', 'ret) adversary-many = 'c-a' ⇒ (unit, 'call' + 'guess', 'ret' + unit) gpv
definition eval-oracle :: 'c-o ⇒ 'c-a ⇒ bool × 's ⇒ 'guess ⇒ (unit × (bool × 's)) spmf
where
eval-oracle c-o c-a = (λ(b, s') guess. map-spmf (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;

(\cdot, (b, -)) \leftarrow \text{exec-gpv}
\leftarrow (\text{oracle c-o}) \oplus_\alpha \text{eval-oracle c-o c-a})
\leftarrow (\not\in c-a)
(False, s);
\text{return-spmf b}
\}

definition \text{advantage-multi} :: ('c-a, 'guess, 'call, 'ret) \text{adversary-many} \Rightarrow \text{real}
\text{where} \text{advantage-multi} \not\in = \text{spmff} (\text{game-multi} \not\in) \text{True}

type-synonym 'guess' \text{reduction-state} = 'guess' + \text{nat}

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

\text{primrec} \text{process-guess} :: 'guess \text{reduction-state} \Rightarrow 'guess \Rightarrow (\text{unit option} \times 'guess \text{reduction-state},
\text{call}, 'ret) \text{gpv}
\text{where}
\text{process-guess} (\text{Inr} j) guess = \text{Done} (\text{if} j > 0 \text{then} (\text{Some} (), \text{Inr} (j - 1)) \text{else} (\text{None, Inl} guess))
\text{process-guess} (\text{Inl} guess) = \text{Done} (\text{None, Inl} guess)

\text{abbreviation} \text{reduction-oracle} :: 'guess + \text{nat} \Rightarrow 'call + 'guess \Rightarrow ((\text{ret + unit}) \text{option} \times
('guess + \text{nat}), 'call, 'ret) \text{gpv}
\text{where} \text{reduction-oracle} \equiv \text{plus-intercept-stop process-call process-guess}

definition \text{reduction} :: \text{nat} \Rightarrow ('c-a, 'guess, 'call, 'ret) \text{adversary-many} \Rightarrow ('c-a, 'guess,
\text{call}, 'ret) \text{adversary-single}
\text{where}
\text{reduction} q \not\in c-a = \text{do} \{ 
\text{j-star} \leftarrow \text{lift-spmf} (\text{spmff-of-set} \{..<q\});
(\cdot, s) \leftarrow \text{inline-stop reduction-oracle} (\not\in c-a) (\text{Inr} j-star);
\text{Done} (\text{projl} s)
\}

\text{lemma} \text{many-single-reduction}: 
\text{assumes bound}: \forall c-a \ c-o s. (c-o, c-a, s) \in \text{set-spmf init} \Rightarrow \text{interaction-bounded-by} (\text{Not}
\text{o isl}) (\not\in c-a) q
\text{and} \text{lossless-oracle}: \forall c-a \ c-o s s' x. (c-o, c-a, s) \in \text{set-spmf init} \Rightarrow \text{lossless-spmf} (\text{oracle}
c-o c-a s' x)
\text{and} \text{lossless-eval}: \forall c-a \ c-o s s' \text{guess}. (c-o, c-a, s) \in \text{set-spmf init} \Rightarrow \text{lossless-spmf} (\text{eval}
c-o c-a s' \text{guess})
shows \( \text{advantage-multi} \not\leq \text{advantage-single} \) (reduction \( q \not\leq \)) \( \ast q \) including lifting-syntax

\( \langle \text{proof} \rangle \)

end

end

1.13 Unpredictable function

theory Unpredictable-Function imports

Guessing-Many-One

begin

locale upf =
  fixes key-gen :: 'key spmf
  and hash :: 'key \( \Rightarrow \) 'x \( \Rightarrow \) 'hash

begin


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

definition oracle-hash :: 'key \( \Rightarrow \) ('x, 'hash, 'x set) callee

where

oracle-hash k = (\( \lambda \) L y. do {
  let t = hash k y;
  let L = insert y L;
  return-spmf (t, L)
})

definition oracle-flag :: 'key \( \Rightarrow \) ('x \times 'hash, unit, bool \times 'x set) callee

where

oracle-flag = (\( \lambda \) key (flg, L) (y, t).
  return-spmf (((), (flg \lor (t = (hash key y) \land y \not\in L), L)))

abbreviation oracle :: 'key \( \Rightarrow \) ('x + 'x \times 'hash, 'hash + unit, bool \times 'x set) callee

where oracle key \equiv \exists (oracle-hash key) \oplus oracle-flag key

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

where

game \not\leq = do {
  key \leftarrow \text{key-gen};
  (\( \cdot \), (flag, L)) \leftarrow \text{exec-gpv} (oracle key) \not\leq (False, \{}); 
  return-spmf flag
}

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

where advantage \not\leq = spmf (game \not\leq) \text{True}

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


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

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

lemma advantage-advantage1:
  assumes bound: interaction-bounded-by (Not ◦ isl) $A$ q
  shows advantage $A$ ≤ advantage1 (guessing-many-one.reduction q (λ- :: unit. $A$) ()) * q
  ⟨proof⟩
end
end

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

2 Cryptographic constructions and their security

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

locale elgamal-base =
  fixes $G$ :: 'grp cyclic-group (structure)
begin

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

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

lemma key-gen-alt:
  key-gen = map-spmf ($\lambda x. (g^x, x)) (sample-uniform (order $G$))
⟨proof⟩

definition aencrypt :: 'grp pub-key ⇒ 'grp ⇒ 'grp cipher spmf
where
  aencrypt $\alpha$ msg = do
  { y ← sample-uniform (order $G$);
  return-spmf ($g^{y}$, ($\alpha^{y}$)⊗msg) }

lemma aencrypt-alt:
  aencrypt $\alpha$ msg = map-spmf ($\lambda y. (g^{y}, (\alpha^{y})⊗msg)) (sample-uniform (order $G$))
⟨proof⟩

definition adecrypt :: 'grp priv-key ⇒ 'grp cipher ⇒ 'grp option
where
  adecrypt $x$ = ($\lambda (\beta, \zeta). \text{Some} (\zeta ⊗ (\text{inv} (\beta^{x}))))$

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

sublocale ind-cpa: ind-cpa key-gen aencrypt adecrypt valid-plains
sublocale ddp: ddp $G$ (proof)

fun elgamal-adversary :: ('grp pub-key, 'grp plain, 'grp cipher, 'state) ind-cpa.adversary ⇒ 'grp ddp.adversary
where
  elgamal-adversary ($\alpha$, $\beta$, $\gamma$) = TRY do
  { b ← coin-spmf;
  (($\text{msg1}$, $\text{msg2}$), $\sigma$) ← $\alpha$ $\beta$;
  — 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 ← 2 (β, γ ⊙ (if b then msg1 else msg2)) σ;
  return-spmf (guess = b)
) ELSE coin-spmf

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

theorem advantage-elgamal: ind-cpa.advantage $\mathcal{A}$ = ddh.advantage (elgamal-adversary $\mathcal{A}$)
  including monad-normalisation
⟨proof⟩

end

locale elgamal-asym =
  fixes $G$ :: security ⇒ 'grp cyclic-group
  assumes elgamal: $\forall \eta. \text{elgamal} (G \eta)$
begin

sublocale elgamal $G \eta$ for $\eta$ ⟨proof⟩

theorem elgamal-secure:
  negligible (λ $\eta. \text{ind-cpa.advantage} \eta (\mathcal{A} \eta))$ if negligible (λ $\eta. \text{ddh.advantage} \eta (\text{elgamal-adversary} \eta (\mathcal{A} \eta)))$
⟨proof⟩

end

context elgamal-base begin

lemma lossless-key-gen [simp]: lossless-spmf (key-gen) ←→ 0 < order $G$
⟨proof⟩

lemma lossless-aencrypt [simp]:
  lossless-spmf (aencrypt key plain) ←→ 0 < order $G$
⟨proof⟩

lemma lossless-elgamal-adversary:
  \[ \text{ind-cpa.lossless} \mathcal{A}; 0 < order \mathcal{G} \]
  ⇒ ddh.lossless (elgamal-adversary $\mathcal{A}$)
⟨proof⟩

end

end
2.2  Hashed Elgamal in the Random Oracle Model

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

type-synonym bitstring = bool list

locale hash-oracle = fixes len :: nat begin

type-synonym 'a state = 'a ⇒ bitstring

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

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

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

lemma invariant-initial [simp]; invariant initial
⟨proof⟩

lemma invariant-update [simp]: [ invariant σ; length bs = len ] ⇒ invariant (σ(x ↦→ bs))
⟨proof⟩

lemma invariant [intro!, simp]: callee-invariant oracle invariant
⟨proof⟩

lemma invariant-in-dom [simp]: callee-invariant oracle (λσ. x ∈ dom σ)
⟨proof⟩

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

lemma card-dom-state;
assumes σ': (x, σ') ∈ set-spmf (exec-gpv oracle gpv σ)
and ihbound: interaction-any-bounded-by gpv n
shows card (dom σ') ≤ n + card (dom σ)
locale elgamal-base = 
fixes \mathcal{G} :: 'grp cyclic-group (structure) 
and len-plain :: nat
begin
sublocale hash: hash-oracle len-plain (proof)
abbreviation hash :: 'grp ⇒ (bitstring, 'grp, bitstring) gpv
where hash x ≡ Pause x Done

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

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

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

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

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

lemma lossless-aencrypt [simp]: lossless-gpv \mathcal{I} (aencrypt α msg) ⟷ 0 < order \mathcal{G}
⟨proof⟩

lemma interaction-bounded-by-aencrypt [interaction-bound, simp]:
interaction-bounded-by (λ -. True) (aencrypt α msg) 1
⟨proof⟩
sublocale ind-cpa: ind-cpa-pk lift-spmf key-gen aencrypt adecrypt valid-plains ⟨proof⟩
sublocale lcdh: lcdh $G$ ⟨proof⟩

fun elgamal-adversary :: ('grp pub-key, plain, 'grp cipher, 'grp bitstring, 'state) ind-cpa.adversary
where
elgamal-adversary $(\mathcal{A}, \mathcal{S}) \alpha \beta = \{ 
\begin{align*}
& (\text{msg1, msg2}, \sigma, s) \leftarrow \text{exec-gpv hash.oracle} (\mathcal{A} \alpha) \text{ hash.initial}; \\
& \text{have to check that the attacker actually sends an element from the group; otherwise stop early}
\end{align*}
\}
\} \text{ ELSE return-spmf (dom s)}
\}
end

locale elgamal = elgamal-base +
assumes cyclic-group: cyclic-group $G$
begin
interpretation cyclic-group $G$ ⟨proof⟩

lemma advantage-elgamal:
includes lifting-syntax
assumes lossless: ind-cpa.lossless $\mathcal{A}$
shows ind-cpa.advantage hash.oracle hash.initial $\mathcal{A} \leq$ lcdh.advantage (elgamal-adversary $\mathcal{A}$)
⟨proof⟩
including monad-normalisation ⟨proof⟩
including monad-normalisation ⟨proof⟩
end

context elgamal-base begin

lemma lossless-key-gen [simp]: lossless-spmf key-gen $\iff 0 < \text{order } G$
⟨proof⟩

lemma lossless-elgamal-adversary:
\[
\text{ind-cpa.lossless } \mathcal{A} ; \forall \eta. 0 < \text{order } G \\
\implies \text{lcdh.lossless (elgamal-adversary } \mathcal{A})
\]
2.3 The random-permutation random-function switching lemma

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

lemma rp-resample:
  assumes B ⊆ A ∪ C A ∩ C = {} C ⊆ B and finB: finite B
  shows bind-spmf (spmf-of-set B) (λx. if x ∈ A then spmf-of-set C else return-spmf x) = spmf-of-set C
⟨proof⟩

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

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

definition game :: bool ⇒ 'a adversary ⇒ bool spmf where
  game b A = run-gpv (if b then rp.A else rf.A) A
Map.empty

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

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

lemma advantage-nonneg: 0 ≤ advantage A ⟨proof⟩

lemma advantage-le-1: advantage A ≤ 1 ⟨proof⟩

context includes I.lifting begin

lift-definition I :: (bool, 'a', 'a) gpv (λx. if x ∈ A then A else {}) ⟨proof⟩

lemma outs-I-I [simp]: outs-I I = A ⟨proof⟩

lemma responses-I-I [simp]: responses-I I x = (if x ∈ A then A else {}) ⟨proof⟩
lifting-update $\mathcal{F}$.lifting
lifting-forget $\mathcal{F}$.lifting
end

lemma rp-rf:
  assumes bound: interaction-any-bounded-by $\mathcal{A}$ $q$
    and lossless: lossless-gpv $\mathcal{A}$ $\mathcal{A}$
    and WT: $\mathcal{F} + g \mathcal{A}$ $\sqrt{\cdot}$
  shows advantage $\mathcal{A} \leq q * q / \text{card } A$
  including lifting-syntax
  \langle proof \rangle
end

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

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

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

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

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

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

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

lemma $\varepsilon$-uh-nonneg : $\varepsilon$-uh ≥ 0
  \langle proof \rangle
lemma hash-ineq-card:
assumes finite W
shows spmf (game-hash-set W) True ≤ ε-uh * card W * card W
(\proof)
end

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

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

lemma lossless-rand [simp]: lossless-spmf rand
(\proof)

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

interpretation prf: prf key-gen f rand (\proof)
interpretation hash: hash seed-gen h(\proof)

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

interpretation prf': prf key-seed-gen f' rand (\proof)

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

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

}\)

**Theorem** \( prf \cdot prf \)'-advantage:

- **Assumes** \( prf \cdot lossless \)
- and **bounded**: \( prf \cdot ibounded-by \)
- **Shows** \( prf \cdot advantage \leq prf \cdot advantage \cdot (prf \cdot reduction) + hash \cdot \epsilon \cdot uh \cdot q \cdot q \)

**Including** lifting-syntax

**Including** monad-normalisation

**Proof**

end

end

### 2.5 IND-CPA from PRF

**Theory** PRF-IND-CPA

- **Imports** CryptHOL.GPV-Bisim
- **Imports** CryptHOL.List-Bits
- **Imports** Pseudo-Random-Function
- **Imports** 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** \( f : key \Rightarrow bool \Rightarrow bool \)
- and **len** :: nat

- **Assumes** \( length-f : \forall xs, ys. \; [ length xs = len; length ys = len ] \Rightarrow length (f xs ys) = len \)

**Begin**

**Definition** key-gen :: bool list spmf

**Where** key-gen = spmf-of-set (nlists UNIV len)

**Definition** valid-plain :: plain \( \Rightarrow bool \)

**Where** valid-plain plain \( \Rightarrow length \) plain = len

**Definition** encrypt :: key \( \Rightarrow plain \Rightarrow cipher \) spmf

**Where**

encrypt key plain = do { 
  \( r \leftarrow spmf-of-set \) (nlists UNIV len);
  return-spmf (r, xor-list plain (f key r)) 
}

32
fun decrypt :: key ⇒ cipher ⇒ plain option
where decrypt key (r, c) = Some (xor-list (f key r) c)

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)
⟨proof⟩

interpretation ind-cpa: ind-cpa key-gen encrypt decrypt valid-plain ⟨proof⟩
interpretation prf: prf key-gen f spmf-of-set (nlists UNIV len) ⟨proof⟩

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

lemma interaction-bounded-by-prf-encrypt-oracle [interaction-bound];
interaction-any-bounded-by (prf-encrypt-oracle σ plain) 1
⟨proof⟩

lemma lossless-prf-encrypt-oracle [simp]: lossless-gpv \ll-top (prf-encrypt-oracle s x)
⟨proof⟩

definition prf-adversary :: (plain, cipher, 'state) ind-cpa.adversary ⇒ (plain, plain) prf.adversary
where
prf-adversary A = do {let (A₁, A₂) = A;
  (((p₁, p₂), σ), n) ← inline prf-encrypt-oracle A₁ ();
  if valid-plain p₁ ∧ valid-plain p₂ then do {
    b ← lift-spmf coin-spmf;
    let pb = (if b then p₁ else p₂);
    r ← lift-spmf (spmf-of-set (nlists UNIV len));
    pad ← Pause r Done;
    let c = (r, xor-list pb pad);
    (b',-) ← inline prf-encrypt-oracle (A₂ c σ) n;
    Done (b' = b)
  } else lift-spmf coin-spmf
}

theorem prf-encrypt-advantage:
assumes ind-cpa.ibounded-by A q
and lossless-gpv \ll-top (fst A)
and \langle cipher σ, lossless-gpv \ll-top (snd A cipher σ)
shows ind-cpa.advantage A ≤ prf.advantage (prf-adversary A) + q / 2 ^ \text{len}
⟨proof⟩

33
including monad-normalisation ⟨proof⟩ including monad-normalisation ⟨proof⟩

including monad-normalisation ⟨proof⟩

lemma interaction-bounded-prf-adversary:
  fixes q :: nat
  assumes ind-cpa ibounded-by A q
  shows prf ibounded-by (prf-adversary A) (1 + q)
⟨proof⟩

lemma lossless-prf-adversary: ind-cpa lossless A =⇒ prf lossless (prf-adversary A)
⟨proof⟩
end

locale otp-η =
  fixes f :: security ⇒ key ⇒ bool list ⇒ bool list
  and len :: security ⇒ nat
  assumes length-f: ∀η xs ys. [length xs = len η; length ys = len η] =⇒ length (f η xs ys) = len η
  and negligible-len [negligible-intros]: negligible (λη. 1 / 2 ^ (len η))
begin
interpretation otp f η len η for η ⟨proof⟩
interpretation ind-cpa: ind-cpa key-gen η encrypt η decrypt η valid-plain η for η ⟨proof⟩
interpretation prf: prf key-gen η f η spmf-of-set (nlists UNIV (len η)) for η ⟨proof⟩

lemma prf-encrypt-secure-for:
  assumes [negligible-intros]: negligible (λη. prf.advantage η (prf-adversary η (A η)))
  and q: ∀η. ind-cpa.ibounded-by (A η) (q η) and [negligible-intros]: polynomial q
  and lossless: ∀η. ind-cpa.lossless (A η)
  shows negligible (λη. ind-cpa.advantage η (A η))
⟨proof⟩
end

end

2.6 IND-CCA from a PRF and an unpredictable function

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

Formalisation of Shoup’s construction of an IND-CCA secure cipher from a PRF
and an unpredictable function [19, §7].

**type-synonym** bitstring = bool list

**locale** simple-cipher =

PRF: prf prf-key-gen prf-fun spmf-of-set (nlists UNIV prf-clen) +
UPF: upf upf-key-gen upf-fun

for prf-key-gen :: 'prf-key spmf
and prf-fun :: 'prf-key ⇒ bitstring ⇒ bitstring
and prf-domain :: bitstring set
and prf-range :: bitstring set
and prf-dlen :: nat
and prf-clen :: nat
and upf-key-gen :: 'upf-key spmf
and upf-fun :: 'upf-key ⇒ bitstring ⇒ 'hash

+ assumes prf-domain-finite: finite prf-domain
  assumes prf-domain-nonempty: prf-domain ≠ {}
  assumes prf-domain-length: x ∈ prf-domain ⇒ length x = prf-dlen
  assumes prf-codomain-length:
     [ key-prf ∈ set-spmf prf-key-gen; m ∈ prf-domain ] ⇒
     length (prf-fun key-prf m) = prf-clen
  assumes prf-key-gen-lossless: lossless-spmf prf-key-gen
  assumes upf-key-gen-lossless: lossless-spmf upf-key-gen

begin

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

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

lemma lossless-key-gen [simp]: lossless-spmf key-gen
⟨proof⟩

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

lemma lossless-encrypt [simp]: lossless-spmf (encrypt k m)
⟨proof⟩
fun decrypt :: ('prf-key × 'upf-key) ⇒ 'hash cipher-text ⇒ bitstring option

where
decrypt (k-prf, k-upf) (x, c, t) = (
  if up-fun k-upf (x @ c) = t ∧ length x = prf-dlen then
    Some (prf-fun k-prf x [⊕] c)
  else
    None
)

lemma cipher-correct:
[ k ∈ set-spmf key-gen; length m = prf-clen ]
⇒ encrypt k m >>= (λc. return-spmf (decrypt k c)) = return-spmf (Some m)
⟨proof⟩

declare encrypt.simps[simp def]

sublocale ind-cca: ind-cca key-gen encrypt decrypt λm. length m = prf-clen ⟨proof⟩
interpretation ind-cca'': ind-cca key-gen encrypt λ - -. None λm. length m = prf-clen ⟨proof⟩

definition intercept-upf-enc :: 'prf-key ⇒ bool ⇒ 'hash cipher-text set × 'hash cipher-text set ⇒ bitstring option × ('hash cipher-text set × 'hash cipher-text set), bitstring + (bitstring × 'hash), 'hash + unit) gpv

where
intercept-upf-enc k b = (λ(L, D) (m1, m0).
  (case (length m1 = prf-clen ∧ length m0 = prf-clen) of
    False ⇒ Done (None, L, D)
    True ⇒ do
      x ← lift-spmf (spmf-of-set prf-domain);
      let c = prf-fun k x [⊕] (if b then m1 else m0);
      t ← Pause (Inl (x @ c)) Done;
      Done ((Some (x, c, projl t)), (insert (x, c, projl t) L, D))
    )))

definition intercept-upf-dec :: 'hash cipher-text set × 'hash cipher-text set ⇒ 'hash cipher-text
⇒ (bitstring option × ('hash cipher-text set × 'hash cipher-text set), bitstring + (bitstring × 'hash), 'hash + unit) gpv

where
intercept-upf-dec = (λ(L, D) (x, c, t).
  if (x, c, t) ∈ L ∨ length x ≠ prf-dlen then Done (None, (L, D)) else do
    Pause (Inr (x @ c, t)) Done;
    Done (None, (L, insert (x, c, t) D))
  )
)

definition intercept-upf ::
  'prf-key ⇒ bool ⇒ 'hash cipher-text set × 'hash cipher-text set ⇒ bitstring × bitstring + 'hash cipher-text
⇒ (′hash cipher-text option + bitstring option) × (′hash cipher-text set × ′hash cipher-text set),
bitstring + (bitstring × ′hash), ′hash + unit gpv

where
intercept-upf k b = plus-intercept (intercept-upf-enc k b) intercept-upf-dec

lemma intercept-upf-simps [simp]:
intercept-upf k b (L, D) (Inr (x, c, t)) =
(if (x, c, t) ∈ L ∨ length x ≠ prf-dlen then Done (Inr None, (L, D)) else do {
  Pause (Inr (x @ c, t)) Done;
  Done (Inr None, (L, insert (x, c, t) D))
})

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

lemma interaction-bounded-by-upf-enc-Inr [interaction-bound]:
interaction-bounded-by (Not ◦ isl) (intercept-upf-enc k b LD mm) 0
⟨proof⟩

lemma interaction-bounded-by-upf-dec-Inr [interaction-bound]:
interaction-bounded-by (Not ◦ isl) (intercept-upf-dec LD c) 1
⟨proof⟩

lemma interaction-bounded-by-intercept-upf-Inr [interaction-bound]:
interaction-bounded-by (Not ◦ isl) (intercept-upf k b LD x) 1
⟨proof⟩

lemma interaction-bounded-by-intercept-upf-Inl [interaction-bound]:
isl x ⇒ interaction-bounded-by (Not ◦ isl) (intercept-upf k b LD x) 0
⟨proof⟩

lemma lossless-intercept-upf-enc [simp]: lossless-gpv (I-full ⊕, I-full) (intercept-upf-enc k b LD mm)
⟨proof⟩

lemma lossless-intercept-upf-dec [simp]: lossless-gpv (I-full ⊕, I-full) (intercept-upf-dec LD mm)
⟨proof⟩

lemma lossless-intercept-upf [simp]: lossless-gpv (I-full ⊕, I-full) (intercept-upf k b

37
lemma results-gpv-intercept-upf [simp]: results-gpv (I-full $\oplus$ I-full) (intercept-upf k b LD x) $\subseteq$ responses-I (I-full $\oplus$ I-full) x $\times$ UNIV

⟨proof⟩

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

lemma lossless-reduction-upf [simp]:
lossless-gpv (I-full $\oplus$ I-full) $\mathcal{A}$ $\Rightarrow$ lossless-gpv (I-full $\oplus$ I-full) (reduction-upf $\mathcal{A}$)
⟨proof⟩

context includes lifting-syntax begin

lemma round-1:
assumes lossless-gpv (I-full $\oplus$ I-full) $\mathcal{A}$
shows $\mid$ spmf (ind-cca.game $\mathcal{A}$) True $-$ spmf (ind-cca'.game $\mathcal{A}$) True $\leq$ UPF.advantage (reduction-upf $\mathcal{A}$)
⟨proof⟩ including monad-normalisation
⟨proof⟩

definition oracle-encrypt2 ::
('prf-key $\times$ 'upf-key) $\Rightarrow$ bool $\Rightarrow$ (bitstring, bitstring) PRF.dict $\Rightarrow$ bitstring $\times$ bitstring \\
$\Rightarrow$ ('hash cipher-text option $\times$ (bitstring, bitstring) PRF.dict) spmf
where
oracle-encrypt2 = (\lambda (k-prf, k-upf) b D (msg1, msg0). (case (length msg1 = prf-clen $\land$ length msg0 = prf-clen) of \\
  False $\Rightarrow$ return-spmf (None, D) \\
  True $\Rightarrow$ do \\
    x $\leftarrow$ spmf-of-set prf-domain; \\
    P $\leftarrow$ spmf-of-set (nlists UNIV prf-clen); \\
    let p = (case D x of Some r $\Rightarrow$ r | None $\Rightarrow$ P); \\
    let c = p [x] (if b then msg1 else msg0); \\
    let t = upf-fun k-upf (x @ c); \\
    return-spmf (Some (x, c, t), D(x $\mapsto$ p)) \\
  )
)

definition oracle-decrypt2 :: ('prf-key $\times$ 'upf-key) $\Rightarrow$ ('hash cipher-text, bitstring option, 'state) callee
where
oracle-decrypt2 = (\lambda key D cipher. return-spmf (None, D))
lemma lossless-oracle-decrypt2 [simp]: lossless-spmf (oracle-decrypt2 k Dbad c) ⟨proof⟩

lemma callee-invariant-oracle-decrypt2 [simp]: callee-invariant (oracle-decrypt2 key) fst ⟨proof⟩

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 ⟨proof⟩

definition game2 :: (bitstring, 'hash cipher-text) ind-cca.adversary ⇒ bool spmf
where
  game2 ≡ do {
    key ← key-gen;
    b ← coin-spmf;
    (b′, D) ← exec-gpv (oracle-encrypt2 key b ⊕ 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: $\text{spmf } (\text{ind-cca}. \text{game } A) \text{True} - \text{spmf } (\text{game2 } A) \text{True} = \text{PRF}$.advantage
(reduction-prf A)

(proof)

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

where
oracle-encrypt3 = (λ(k-prf, k-upf) b (bad, D) msg10)
  (case (length msg1 = prf-clen ∧ length msg0 = prf-clen) of
  False ⇒ return-spmf (None, (bad, D))
  True ⇒ do {
    x ← \text{spmf-of-set } prf-domain;
    P ← \text{spmf-of-set } (nlists \text{UNIV } prf-clen);
    let (p, F) = (case D x of Some r ⇒ (P, True) | None ⇒ (P, False));
    let c = p [⊕] (if b then msg1 else msg0);
    let t = upf-fun k-upf (x @ c);
    return-spmf (Some (x, c, t), (bad ∨ F, D(x ↦ p)))
  }))

lemma lossless-oracle-encrypt3 [simp];
lossless-spmf (oracle-encrypt3 k b D m10)
(proof)

lemma callee-invariant-oracle-encrypt3 [simp]; callee-invariant (oracle-encrypt3 key b) fst
(proof)

definition game3 :: (bitstring, 'hash cipher-text) ind-cca.adversary ⇒ (bool × bool) \text{spmf}

where
game3 A ≡ do {
  key ← key-gen;
  b ← coin-spmf;
  (b', (bad, D)) ← exec-gpv (oracle-encrypt3 key b ⊕ oracle-decrypt2 key) A (False, Map-empty);
  return-spmf (b = b', bad)
}

lemma round-3:
assumes lossless-gpv (A.full ⊕ A.full) A
shows \text{measure } (measure-spmf (game3 A)) ((b, bad), b) = \text{spmf } (game2 A) \text{True}
  ≤ \text{measure } (measure-spmf (game3 A)) ((b, bad), bad)
(proof)

lemma round-4:
assumes lossless-gpv (A.full ⊕ A.full) A

40
shows map-spmf \text{fst} \ (\text{game3} \ \mathcal{A}) = \text{coin-spmf}
\langle \text{proof} \rangle \ \text{including} \ \text{monad-normalisation}
\langle \text{proof} \rangle

\textbf{lemma} \ \text{game3-bad}:
\begin{itemize}
\item \textbf{assumes} interaction-bounded-by \text{isl} \ \mathcal{A} q
\item \textbf{shows} measure \ (\text{measure-spmf} \ (\text{game3} \ \mathcal{A})) \ \{(b, \ \text{bad}, \ \text{bad}\} \leq q / \ \text{card prf-domain} * q
\end{itemize}
\langle \text{proof} \rangle

\textbf{theorem} \ \text{security}:
\begin{itemize}
\item \textbf{assumes} lossless: lossless-gpv \ (\mathcal{I} - \text{full} \oplus \mathcal{I} - \text{full}) \ \mathcal{A}
\item \textbf{and} bound: interaction-bounded-by \text{isl} \ \mathcal{A} q
\item \textbf{shows} ind-cca.\text{advantage} \ \mathcal{A} \leq
PRF.\text{advantage} \ (\text{reduction-prf} \ \mathcal{A}) + UPF.\text{advantage} \ (\text{reduction-upf} \ \mathcal{A}) +
\ \text{real} q / \ \text{real} (\ \text{card prf-domain}) * \ \text{real} q \ \text{is} ?LHS \leq -
\end{itemize}
\langle \text{proof} \rangle

\textbf{theorem} \ \text{security1}:
\begin{itemize}
\item \textbf{assumes} lossless: lossless-gpv \ (\mathcal{I} - \text{full} \oplus \mathcal{I} - \text{full}) \ \mathcal{A}
\item \textbf{assumes} \ q; \ \text{interaction-bounded-by} \ \text{isl} \ \mathcal{A} q
\item \textbf{and} \ q'\; \text{interaction-bounded-by} \ (\text{Not} \circ \text{isl}) \ \mathcal{A} q'
\item \textbf{shows} ind-cca.\text{advantage} \ \mathcal{A} \leq
\ PRF.\text{advantage} \ (\text{reduction-prf} \ \mathcal{A}) +
\ \text{UPF.}\text{advantage} \ (\text{guessing-many-one.reduction} \ q' \ (\lambda -. \text{reduction-upf} \ \mathcal{A}) ()) * q' +
\ \text{real} q * \text{real} q / \ \text{real} (\ \text{card prf-domain})
\end{itemize}
\langle \text{proof} \rangle

end

end

locale simple-cipher' =
\begin{itemize}
\item \textbf{fixes} prf-key-gen :: \text{security} \Rightarrow ' \text{prf-key spmf}
\item \textbf{and} prf-fun :: \text{security} \Rightarrow ' \text{prf-key} \Rightarrow \text{bitstring} \Rightarrow \text{bitstring}
\item \textbf{and} prf-domain :: \text{security} \Rightarrow \text{bitstring set}
\item \textbf{and} prf-range :: \text{security} \Rightarrow \text{bitstring set}
\item \textbf{and} prf-dlen :: \text{security} \Rightarrow \text{nat}
\item \textbf{and} prf-clen :: \text{security} \Rightarrow \text{nat}
\item \textbf{and} upf-key-gen :: \text{security} \Rightarrow ' \text{upf-key spmf}
\item \textbf{and} upf-fun :: \text{security} \Rightarrow ' \text{upf-key} \Rightarrow \text{bitstring} \Rightarrow ' \text{hash}
\item \textbf{assumes} simple-cipher; \ \langle \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) \rangle
\end{itemize}
\textbf{begin}

\textbf{sublocale} simple-cipher
\begin{itemize}
\item prf-key-gen \ \eta \ prf-fun \ \eta \ prf-domain \ \eta \ prf-range \ \eta \ prf-dlen \ \eta \ prf-clen \ \eta \ upf-key-gen \ \eta \ upf-fun \ \eta
\item for \ \eta
\end{itemize}
\textbf{theorem} security-asymptotic:
\begin{itemize}
  \item \textbf{fixes} q q' :: security ⇒ nat
  \item \textbf{assumes} lossless: \( \forall \eta. \text{lossless-gpv} (\mathcal{A} \text{-full} \oplus \mathcal{A} \text{-full}) (\mathcal{A} \eta) \)
  \item \textbf{and} bound: \( \forall \eta. \text{interaction-bounded-by} \text{isl} (\mathcal{A} \eta) (q \eta) \)
  \item \textbf{and} bound': \( \forall \eta. \text{interaction-bounded-by} (\text{Not} \circ \text{isl}) (\mathcal{A} \eta) (q' \eta) \)
  \item \textbf{and} \[ \text{negligible-intros}\]:
    \begin{itemize}
      \item polynomial q polynomial q
      \item negligible \( (\lambda \eta. \text{PRF}.\text{advantage} \eta \text{(reduction-prf} \eta \mathcal{A} \eta)) \)
      \item negligible \( (\lambda \eta. \text{UPF}.\text{advantage1} \eta \text{(guessing-many-one.reduction} (q' \eta) (\lambda -. \text{reduction-upf} \eta \mathcal{A} \eta))()) \)
      \item negligible \( (\lambda \eta. 1 / \text{card} \text{(prf-domain} \eta)) \)
    \end{itemize}
  \item \textbf{shows} negligible \( (\lambda \eta. \text{ind-cca}.\text{advantage} \eta \mathcal{A} \eta) \)
\end{itemize}
\textbf{⟨proof⟩}
\begin{itemize}
  \item \textbf{end}
\end{itemize}
\textbf{theory} Cryptographic-Constructions \textbf{imports}
\begin{itemize}
  \item Elgamal
  \item Hashed-Elgamal
  \item RP-RF
  \item PRF-UHF
  \item PRF-IND-CPA
  \item PRF-UPF-IND-CCA
\end{itemize}
\textbf{begin}
\begin{itemize}
  \item \textbf{end}
\end{itemize}
\textbf{theory} Game-Based-Crypto \textbf{imports}
\begin{itemize}
  \item Security-Spec
  \item Cryptographic-Constructions
\end{itemize}
\textbf{begin}
\begin{itemize}
  \item \textbf{end}
\end{itemize}
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.

```isabelle
theory CryptHOL-Tutorial imports CryptHOL begin
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.\footnote{The cryptographic literature sometimes abstracts over the adversary and defines the advantage to be the advantage of the best “feasible” adversary against a game. Such abstraction would require a formalization of feasibility, for which CryptHOL currently does not offer any support. We therefore always consider the advantage of a specific adversary.} We return to asymptotic security statements in §6.

A formalization of a security property must therefore specify all of the following:
• 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 $G$ be a group of order $q$ with generator $g$. The List Computational Diffie-Hellman (LCDH) assumption holds for $G$ 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 $G$ using Isabelle’s module system based on locales. The locale `list-cdh` fixes $G$ to be a finite cyclic group that has elements of type `′grp cyclic-group` and comes with a generator $g_G$. Basic facts about finite groups are formalized in the CryptHOL theory `CryptHOL.Cyclic-Group`.

```isabelle
locale list-cdh = cyclic-group $G$
  for $G$ :: ′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

---

3The syntax directive `structure` tells Isabelle that all group operations in the context of the locale refer to the group $G$ unless stated otherwise. For example, $g_G$ can be written as $g$ inside the locale. Isabelle automatically adds the locale parameters and the assumptions on them to all definitions and lemmas inside that locale. Of course, we could have made the group $G$ an explicit argument of all functions ourselves, but then we would not benefit from Isabelle’s module system, in particular locale instantiation.
probability may be “lost”. A subprobability distribution is called lossless, written \textit{lossless-spmf}, if its probability mass is 1.) We define the following abbreviation as a shorthand for the type of LCDH adversaries.\footnote{Actually, the type of group elements has already been fixed in the locale \textit{list-cdh} to the type \texttt{'grp}. Unfortunately, such fixed type variables cannot be used in type declarations inside a locale in Isabelle2018. The \texttt{type-synonym adversary} is therefore parametrized by a different type variable \texttt{'grp'}, but it will be used below only with \texttt{'grp}.}

\texttt{type-synonym ‘grp’ adversary = ‘grp’ \Rightarrow ‘grp’ \Rightarrow ‘grp’ set spmf}

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

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

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

- \texttt{order $G$} denotes the order of $G$ and $([\cdot]) :: ‘grp \Rightarrow nat \Rightarrow ‘grp$ 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 $g^{x \cdot y}$, where $g$ is the generator of $G$. The game finally returns a boolean that indicates whether the adversary produced a right guess. Formally, \texttt{game $A$} is a boolean random variable.

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

The advantage of the adversary is equivalent to its probability of winning the LCDH game. The function \texttt{spmf :: ‘a spmf \Rightarrow ‘a \Rightarrow real} returns the probability of an elementary event under a given subprobability distribution.

\texttt{definition advantage :: ‘grp adversary \Rightarrow real where advantage $A$ = spmf (game $A$) True}
This completes the formalisation of the LCDH game and we close the locale `list-cdh` with `end`. The above definitions are now accessible under the names `game` and `advantage`. Furthermore, when we later instantiate the locale `list-cdh`, they will be specialized to the given parameters. We will return to this topic in §4.5.

4.2 A Random Oracle

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

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

```
type-synonym bitstring = bool list

locale random-oracle =
  fixes len :: nat
begin

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

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

```
type-synonym 'a state = 'a ⇒ 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))
```

Initially, the state of a random oracle is the empty map `empty`, as no queries have been asked. For readability, we introduce an abbreviation:
abbreviation (input) initial :: 'a state where initial ≡ 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)
⟨proof⟩

lemma fresh:
oracle σ x =
(do { bs ← spmf-of-set (nlists UNIV len);
return-spmf (bs, σ(x↦→ bs)) })
if σ x = None
⟨proof⟩
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, \(\text{ind-cpa-pk}\).

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

\[
\text{locale } \text{ind-cpa-pk} = \\
\text{fixes } \text{key-gen} :: (\text{pubkey} \times \text{privkey} \times \text{query} \times \text{response}) \text{gpv} \\
\text{and } \text{encrypt} :: \text{pubkey} \Rightarrow \text{plain} \Rightarrow (\text{cipher}, \text{query}, \text{response}) \text{gpv} \\
\text{and } \text{decrypt} :: \text{privkey} \Rightarrow \text{cipher} \Rightarrow (\text{plain}, \text{query}, \text{response}) \text{gpv} \\
\text{and } \text{valid-plains} :: \text{plain} \Rightarrow \text{plain} \Rightarrow \text{bool}
\]

begin

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

### 4.4 Security notions with oracles: IND-CPA security

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

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

**Stage 1 (find):**
1. The challenger generates a public key $pk$ using $\text{key-gen}$ 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 $\text{assert-gpv}(\text{valid-plains } m_0 \ m_1)$ 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 $\text{ELSE}$ branch.

```haskell
fun game :: ('pubkey', 'plain', 'cipher', 'query', 'response', 'state) adversary
  ⇒ (bool, 'query', 'response') gpv
where
  game (σ₁, σ₂) = TRY do {
    (pk, sk) ← key-gen;
    ((m₀, m₁), σ) ← σ₁ pk;
    assert-gpv (valid-plains m₀ m₁);
    b ← lift-spmf coin-spmf;
```
cipher ← encrypt pk (if b then m₀ else m₁);
  b' ← spmf cipher σ;
  Done (b' = b)
} ELSE lift-spmf coin-spmf

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

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

fun advantage :: (′σ, ′query, ′response) oracle′ × ′σ
⇒ (′pubkey, ′plain, ′cipher, ′query, ′response, ′state) adversary ⇒ real
where
  advantage (oracle, σ) σ' = |spmf (run-gpv oracle (game σ') σ) True - 1/2|
end

4.5 Concrete cryptographic constructions: the hashed ElGamal encryption scheme

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

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

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

```
locale hashed-elgamal =
  lcdh: list-cdh \texttt{G} +
  ro: random-oracle \texttt{len-plain}
  for \texttt{G} :: \texttt{grp cyclic-group (structure)}
  and \texttt{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.\(^5\)

For readability, we introduce meaningful abbreviations for these concepts.

\[\text{type-synonym 'grp' pub-key = 'grp'}\]

\(^5\)More precisely, the private key ranges between 0 and \( q - 1 \) and the bitstrings are of length \texttt{len-plain}. However, Isabelle/HOL’s type system cannot express such properties that depend on locale parameters.
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 $\text{hash}$. 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 hash :: '$grp' ⇒ (bitstring, '$grp', bitstring) gpv
where
  hash x ≡ Pause x Done

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

definition encrypt :: '$grp' pub-key ⇒ plain ⇒ ('$grp' cipher, '$grp', bitstring) gpv
where
  encrypt α msg = do { y ← lift-spmf (sample-uniform (order $\mathcal{G}$)); h ← hash (α $\hat{\cdot}$ y); Done (g $\hat{\cdot}$ y, h $\oplus$ msg) }

definition decrypt :: '$grp' priv-key ⇒ '$grp' cipher ⇒ (plain, '$grp', bitstring) gpv
where
  decrypt x = (λ(β, ζ). do { h ← hash (β $\hat{\cdot}$ x); Done (ζ $\oplus$ h) }))

definition valid-plains :: plain ⇒ plain ⇒ bool
where
  valid-plains msg1 msg2 ≡ length msg1 = len-plain ∧ length msg2 = len-plain
```

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

```
sublocale ind-cpa: ind-cpa-pk (lift-spmf key-gen) encrypt decrypt valid-plains ⟨proof⟩
```

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

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 for hashed Elgamal that depends on another adversary’s advantage in the LCDH 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 $\text{empty}$.

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), s) ← 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.⁶

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.

⁶A security proof often involves several reductions. The bound then depends on several advantages, one for each reduction.
Our proof for the concrete security theorem needs the side condition that the adversary is lossless. Losslessness for adversaries is similar to losslessness for subprobability distributions. It ensures that the adversary always terminates and returns an answer to the challenger. For the IND-CPA game, we define losslessness as follows:

**definition** (in ind-cpa-pk) **lossless**

\[ (\text{\textquotesingle}pubkey, \text{\textquotesingle}plain, \text{\textquotesingle}cipher, \text{\textquotesingle}query, \text{\textquotesingle}response, \text{\textquotesingle}state) \text{ adversary} \Rightarrow \text{bool} \]

where

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

\[ \wedge (\forall cipher \sigma. \text{lossless-gpv} F \text{-full} (A_2 cipher \sigma))) \]

So now let’s start with the proof.

**proof**

As a preparatory step, we split the adversary \( A \) into its two phases \( A_1 \) and \( A_2 \). 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 \( A \).

\[ \text{obtain } A \text{ } \text{where } \text{\textquotesingle}A\text{ \textquotesingle simp} : A = (A_1, A_2) \text{ by (cases } A) \]

\[ \text{from } \text{lossless have } \text{lossless1 \text{ simp} : } \text{\textquotesingle}A\text{ \textquotesingle simp} : \lambda pk. \text{lossless-gpv} F \text{-full} (A_1 pk) \]

\[ \text{and } \text{lossless2 \text{ simp} : } \lambda \sigma cipher. \text{lossless-gpv} F \text{-full} (A_2 cipher \sigma) \]

\[ \text{by (auto simp add: ind-cpa.lossless-def)} \]

### 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 **interceptor** between the adversary and the oracle that records all adversary’s queries.

**define** **interceptor** :: (grp set) ⇒ (bitstring × (grp set)) gpv

**where**

\[ \text{interceptor } \sigma x = (\text{do } \{ \]

\[ h \leftarrow \text{hash } x; \]

\[ \text{Done (h, insert } x \sigma) \]

\[ )) \text{ for } \sigma x \]

We integrate this interceptor into the **game** using the **inline** function as illustrated in Fig. 4 and name the result **game**\(_0\).

**define** **game**\(_0\) where

---

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

game_0 = TRY do
  (pk, -) ← lift-spmf key-gen;
  (((m_1, m_2), σ), s) ← inline interceptor (A_1 pk) {};
  assert-gpv (valid-plains m_1 m_2);
  b ← lift-spmf coin-spmf;
  c ← encrypt pk (if b then m_1 else m_2);
  (b’, s’) ← inline interceptor (A_2 c σ) s;
  Done (b’ = b)
} ELSE lift-spmf coin-spmf

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

More precisely, we have been very careful that the two games game A and game_0 have identical structure. They differ only in that game_0 uses the adversary (λpk. inline interceptor (A_1 pk) θ, λcipher σ. inline interceptor (A_2 cipher σ)) instead of A. The formal justification for this replacement happens in two steps:

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

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

The first step is automated using Isabelle’s Transfer package [9], which is based on Mitchell’s representation independence [14]. The replacement is controlled by so-called transfer rules of the form R x y which indicates that x shall replace y; the correspondence relation R captures the kind of replacement. The transfer proof method then constructs a constraint system with one constraint for each atom in the
proof goal where the correspondence relation and the replacement are unknown. It then tries to solve the constraint system using the rules that have been declared with the attribute \texttt{[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: \texttt{interceptor} records all queries in a set whereas \texttt{id-oracle} has no state, which is modelled with the singleton type \texttt{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 \texttt{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 (\texttt{unfolding}) makes the two oracle transformers structurally identical except for the state update function. Having proved that the state update function \(\lambda\cdot\sigma.\sigma\) is a correct replacement for \texttt{insert} w.r.t. \(cr\), the \texttt{transfer-prover} then lifts this replacement to the bisimulation rule. Here, \texttt{transfer-prover} is similar to \texttt{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 \texttt{transfer} works.

\begin{verbatim}
{ define cr :: unit \to 'grp set \to bool where cr \sigma \sigma' = True for \sigma \sigma'
 have [transfer-rule]: cr () {} by (simp add: cr-def) — initial states
 have [transfer-rule]: ((=) =====> cr =====> cr) (\lambda\cdot\sigma.\sigma) insert — state update
  by (simp add: rel-fun-def cr-def)
 have [transfer-rule]: — cr is a bisimulation for the oracle transformers
  (cr =====> ( =) =====> rel-gpv (rel-prod ( =) cr) ( =)) id-oracle interceptor
 unfolding interceptor-def id-oracle-def bind-gpv-Pause bind-rpv-Done
  by transfer-prover
 have ind-cpa.game \not\equiv = game0 unfolding game0-def \not\equiv = ind-cpa.game.simps
   by transfer (simp add: bind-map-gpv o-def ind-cpa.game.simps split-def)
}
\end{verbatim}

5.4 Equational program transformations

Before we move on, we need to simplify \texttt{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 \texttt{interceptor} with the random oracle \texttt{oracle} into a new oracle \texttt{oracle'} with which the adversary interacts.

\begin{verbatim}
define oracle' :: 'grp set \times ('grp \to bitstring) \to 'grp \Rightarrow -
where oracle' = (\lambda(s, \sigma) x. do { (h, \sigma') \leftarrow case \sigma x of
\end{verbatim}

60
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 s) σ =
  map-spmf (λ(a, b, c). ((a, b), c)) (exec-gpv oracle' s, σ) for s σ
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 Γ);
  y ← sample-uniform (order Γ);
  b ← coin-spmf;
  ((msg1, msg2), σ), (s, s-h) ←
    exec-gpv oracle' (λ( g ^ x) ) ({}, ro.initial);
  TRY do {
    - :: unit ← assert-spmf (valid-plains msg1 msg2);
    (h, s-h') ← ro.oracle s-h (g ^ (x * y));
    let cipher = (g ^ y), h (if b then msg1 else msg2));
    (b', (s', s-h'')) ← exec-gpv oracle' (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 Is-
abelle, 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.
Figure 5: The IND-CPA game after flattening. The blue box around the encryption algorithm and the random oracle represents the expanded definition of them.

- The equations `exec-gpv-bind`, `try-bind-assert-gpv`, and `try-gpv-bind-lossless` ensure that the operator `exec-gpv`, which connects the `game0` with the random oracle, is distributed over the sequencing. Together with `*`, this gives the adversary access to `oracle'` instead of the interceptor and the random oracle, and makes the call to the random oracle in the encryption of the chosen message explicit.

- The theorem `lcdh.nat-pow-pow` rewrites the iterated exponentiation $(g[^x]y)$ to $(g[^x*y])$.

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

Note that the state of the oracle `oracle'` is changed between $A_1$ and $A_2$. Namely, the random oracle’s part `s-h` may change when the chosen message is encrypted, but the state that records the adversary’s queries `s` is passed on unchanged.

### 5.5 Capturing a failure event

Suppose that two games behave the same except when a so-called failure event occurs [19]. Then the chance of an adversary distinguishing the two games is bounded by the probability of the failure event. In other words, the simulation of the reduction is allowed to break if the failure event occurs. In the running example, such an argument is a key step to derive the bound on the adversary’s advantage.

But to reason about failure events, we must first introduce them into the games we consider. This is because in CryptHOL, the probabilistic programs describe probability distributions over what they return (return-spmf). The variables that are used internally in the program are not accessible from the outside, i.e., there is
no memory to which these are written. This has the advantage that we never have
to worry about the names of the variables, e.g., to avoid clashes. The drawback is
that we must explicitly introduce all the events that we are interested in.
Introducing a failure event into a game is straightforward. So far, the games game
and game0 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 \( g[^\hat{\cdot}](x \ast y) \) that is used for encrypting
the chosen message \( m_b \). So we simply change the return type of the game to return
whether the adversary guessed right and whether the failure event has occurred.
The next definition game1 does so. (Recall that oracle' stores in its first state com-
ponent \( s \) 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 game1.

```quantum
define game1 :: nat ⇒ nat ⇒ (bool × bool) spmf
where
    game1 x y = do 
        b ← coin-spmf;
        ((\( (m_1, m_2), \sigma) \), (s, s-h)) ← exec-gpv oracle' (A1 (g[^\hat{\cdot}] x)) ({}, ro.initial);
        TRY do 
            -(:: unit ← assert-spmf (valid-plains m_1 m_2);
                (h, s-h') ← ro.oracle s-h (g[^\hat{\cdot}] (x \ast y));
               let c = (g[^\hat{\cdot}] y, h \oplus (if b then m_1 else m_2));
               (b', (s', s-h'')) ← exec-gpv oracle' (A2 c \sigma) (s, s-h');
               return-spmf (b' = b, g[^\hat{\cdot}] (x \ast y) ∈ s')
        ELSE do 
            b ← coin-spmf;
            return-spmf (b, g[^\hat{\cdot}] (x \ast y) ∈ s)
        }
    for x y
```

It is easy to prove that game0 combined with the random oracle is a projection of
game1 with the sampling added, as formalized in game0-game1.

```quantum
let ?sample = λf :: nat ⇒ nat ⇒ - spmf. do 
    x ← sample-uniform (order G);
    y ← sample-uniform (order G);
    f x y 
have game0-game1:
    run-gpv ro.oracle game0 ro.initial = map-spmf fst (?sample game1) 
    by (simp add: game0_game1_def o_def split_def map-try-spmf map-scale-spmf)
```

### 5.6 Game hop based on a failure event

A game hop based on a failure event changes one game into another such that they
behave identically unless the failure event occurs. The fundamental-lemma bounds
the absolute difference between the two games by the probability of the failure
event. In the running example, we would like to avoid querying the random oracle
when encrypting the chosen message. The next game game2 is identical except that
the call to the random oracle oracle is replaced with sampling a random bitstring.  

\[ \text{define game}_2 :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{bool} \times \text{bool}) \text{ spmf} \]

\[ \text{where game}_2 \ x \ y = \text{do} \ \{ \]

\[ b \leftarrow \text{coin-spmf}; \]

\[ ((m_1, m_2), \sigma), (s, s-h)) \leftarrow \text{exec-gpv oracle'} (\text{set}_1 (g \ [\ ] \ x)) (\{\}, \text{ro.initial}); \]

\[ \text{TRY do} \]

\[ :: \text{unit} \leftarrow \text{assert-spmf} (\text{valid-plains} m_1 m_2); \]

\[ h \leftarrow \text{spmfs-of-set (nlists UNIV len-plain)}; \]

\[ \text{— We do not query the random oracle for } g \ [\ ] (x * y), \text{ but instead sample a random bitstring } h \text{ directly. So the rest differs from } \text{game}_1 \text{ only if the adversary queries } g \ [\ ] (x * y). \]

\[ \text{let cipher} = (g \ [\ ] \ y, \ h [\ ] (if } b \text{ then } m_1 \text{ else } m_2); \]

\[ (b', (s', s-h')) \leftarrow \text{exec-gpv oracle'} (\text{set}_2 \text{ cipher } \sigma) (s, s-h); \]

\[ \text{return-spmf} (b' = b, g \ [\ ] (x * y) \in s') \]

\[ \} \text{ ELSE do} \]

\[ b \leftarrow \text{coin-spmf}; \]

\[ \text{return-spmf} (b, g \ [\ ] (x * y) \in s) \]

\[ \} \text{ for } x \ y \]

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

\[ \text{have rel-spmf } (\lambda (\text{win}, \text{bad}) (\text{win'}, \text{bad'}). \text{bad} = \text{bad'} \land (\neg \text{bad'} \rightarrow \text{win} = \text{win'}) (\text{game}_2 \ x \ y) (\text{game}_1 \ x \ y) \text{ for } x \ y \]

\[ \text{proof —} \]

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

\[ \text{interpret inv-oracle'} \cdot \text{callee-invariant-on oracle'} (\lambda (s, s-h), s = \text{dom s-h}) \cdot \mathcal{F} \text{-full} \]

\[ \text{by unfold-locales(auto simp add: oracle'-def split: option.split-asm if-split)} \]

\[ \text{interpret bad: callee-invariant-on oracle'} (\lambda (s, -), z \in s) \cdot \mathcal{F} \text{-full for } z \]

\[ \text{by unfold-locales(auto simp add: oracle'-def)} \]

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

\[ \text{let } ?X = \lambda (s, s-h) (s', s-h'), s = \text{dom s-h} \land s' = s \land s-h = s-h' (g \ [\ ] (x * y) := \text{None}) \]

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

\[ \text{let } ?\text{bad} = \lambda (s, s-h). g \ [\ ] (x * y) \in s \]

\[ \text{let } ?R = (\lambda (a, s1') (b, s2'). ?\text{bad} s1' = ?\text{bad} s2' \land (\neg ?\text{bad} s2' \rightarrow a = b \land ?X s1' s2')) \]

\[ \text{have bisim: rel-spmf } ?R (\text{oracle'} s1 \text{ plain}) (\text{oracle'} s2 \text{ plain}) \]

---

9 In Shoup’s terminology [19], such a step makes (a gnome sitting inside) the random oracle forgetting the query.
if ?X s1 s2 for s1 s2 plain using that
by (auto split: prod.split intro!: rel-spmf-bind-reflI simp add: oracle' def rel-spmf-return-spmf)

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

Now we are ready to prove that the two games \( \text{game}_1 \) and \( \text{game}_2 \) 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 \text{game}_1-def \text{game}_2-def
— Peel off the first phase of the game using the structural decomposition rules rel-spmf-bind-reflI and rel-spmf-try-spmf.
apply (clarsimp intro!: rel-spmf-bind-reflI simp del: bind-spmf-const)
apply (rule rel-spmf-try-spmf)

subgoal TRY for \( b \ m \_ \_ \_ \sigma \ s \ s-h \)
apply (rule rel-spmf-bind-reflI)
— Exploit that in the first phase of the game, the set \( s \) of queried strings and the map of the random oracle \( s-h \) are updated in lock step, i.e., \( s = \text{dom } s-h \).
apply (drule inv-oracle' exec-gpv-invariant; clarsimp)
— Has the adversary queried the random oracle with \( g [\hat{\_}] (x * y) \) during the first phase?
apply (cases \( g [\hat{\_}] (x * y) \in s \))

subgoal True — Then the failure event has already happened and there is nothing more to do. We just have to prove that the two games on both sides terminate with the same probability.
by (intro rel-spmf-bindI1 rel-spmf-bindI2 lossless-exec-gpv\[where \ I = I' \-full\] dest!: bad.exec-gpv-invariant)

subgoal False — Then let's see whether the adversary queries \( g [\hat{\_}] (x * y) \) in the second phase. Thanks to ro.fresh, the call to the random oracle simplifies to sampling a random bitstring.
apply (clarsimp iff del: domIff simp add: domIff ro.fresh intro!: rel-spmf-bindI)
apply (rule rel-spmf-bindI\[where \ R = \?R\])
— The lemma exec-gpv-oracle-bisim-bad-full lifts the bisimulation for oracle' to the adversary \( s/2 \) 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 \( \text{game}_1 \) and \( \text{game}_2 \), apply the fundamental-lemma.

hence rel-spmf \( (\lambda (\text{win}, \text{bad}) (\text{win'}, \text{bad'}) . (\text{bad} \leftrightarrow \text{bad'}) \land (\neg \text{bad'} \rightarrow \text{win} \leftrightarrow \text{win'}) \) \( (?\text{sample } \text{game}_2) (?\text{sample } \text{game}_1) \)
by (intro rel-spmf-bind-reflI)
hence \( \text{measure} \text{(measure spmf (?sample game\textsubscript{2})) \{(win, -), win\}} - \text{measure} \text{(measure spmf (?sample game\textsubscript{1})) \{(win, -), win\}} \]
\[ \leq \text{measure} \text{(measure spmf (?sample game\textsubscript{2})) \{ (-, bad), bad \}} \]

unfolding split-def by (rule fundamental-lemma)

moreover

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

have \( \text{measure} \text{(measure spmf (?sample game\textsubscript{2})) \{(win, -), win\}} = \text{spmf} \text{(map spmf fst (?sample game\textsubscript{2})) True} \)

and \( \text{measure} \text{(measure spmf (?sample game\textsubscript{1})) \{(win, -), win\}} = \text{spmf} \text{(map spmf fst (?sample game\textsubscript{1})) True} \)

and \( \text{measure} \text{(measure spmf (?sample game\textsubscript{2})) \{ (-, bad), bad \}} = \text{spmf} \text{(map spmf snd (?sample game\textsubscript{2})) True} \)

unfolding spmf-com-measure-spmf measure-map-spmf by (auto simp add: vimage-def split-def)

ultimately have hop12:

\[ |\text{spmf} \text{(map spmf fst (?sample game\textsubscript{2})) True} - \text{spmf} \text{(map spmf fst (?sample game\textsubscript{1})) True}| \leq \text{spmf} \text{(map spmf snd (?sample game\textsubscript{2})) True} \]

by simp

5.7 Optimistic sampling: the one-time-pad

This step is based on the one-time-pad, which is an instance of optimistic sampling. If two runs of the two games in an optimistic sampling step would use the same random bits, then their results would be different. However, if the adversary’s choices are independent of the random bits, we may relate runs that use different random bits, as in the end, only the probabilities have to match. The previous game hop from \( \text{game\textsubscript{1}} \) to \( \text{game\textsubscript{2}} \) 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 \( \text{game\textsubscript{2}} \) by the part that does the optimistic sampling and call this parametrized version \( \text{game\textsubscript{3}} \).

define \( \text{game\textsubscript{3}} :: (\text{bool} \Rightarrow \text{bitstring} \Rightarrow \text{bitstring} \Rightarrow \text{bitstring} \Rightarrow \text{spmf}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{bool} \times \text{bitstring}) \ => \text{spmf} \)

where \( \text{game\textsubscript{3}} \ f x y = \text{do} \{
\ b \leftarrow \text{coin spmf};\n\ ((m\textsubscript{1}, m\textsubscript{2}), \sigma), (s, s\text{-}h)) \leftarrow \text{exec gpv oracle}^\prime (\text{oracle}^\prime_1 (g \ [\ y \ ] x)) (\{ \}, \text{ro. initial});\n\ TRY \ do \{
\ \ :: \text{unit} \leftarrow \text{assert spmf (valid plains m\textsubscript{1} m\textsubscript{2})};\n\ h' \leftarrow f \ b \ m\textsubscript{1} \ m\textsubscript{2};\n\ let \ \text{cipher} = (g \ [\ y \ ] \ h');\n\ (b', (s', s\text{-}h')) \leftarrow \text{exec gpv oracle}^\prime (\text{cipher} \ \sigma) \ (s, s\text{-}h);
\\}
\\}
\\}
\\}
\\}

66
return-spmf \( b' = b, g \{ \} (x \ast y) \in s' \)
] ELSE do {
    b \leftarrow \text{coin-spmf};
    return-spmf \( b, g \{ \} (x \ast y) \in s \)
}

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

let \( ?f' = \lambda b m_1 m_2. \text{map-spmf} (\lambda h. (if b then m_1 else m_2) \{ \} h) (\text{spmf-of-set} (\text{nlists UNIV len-plain})) \)

have \( \text{game}_2 = \text{game}_3 \): \( \text{game}_2 \ x \ y = \text{game}_3 \ ?f \ x \ y \) for \( x \ y \)

by(simp add: \( \text{game}_2\text{-def} \ \text{game}_3\text{-def} \ \text{Let-def} \ \text{bind-map-spmf} \ \text{xor-list-commute} \ \text{o-def} \))

CryptHOL’s \( \text{one-time-pad} \) 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 \text{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.

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

have \( \text{game}_3 = \text{game}_3 \ ?f \ ?f' \) for \( x \ y \)

by(auto intro!: \( \text{try-spmf-cong} \ \text{bind-spmf-cong} \ \text{OF refl} \ \text{if-cong} \ \text{OF refl} \) simp add: \( \text{game}_3\text{-def} \ \text{split-def} \ \text{one-time-pad} \ \text{valid-plains-def} \))

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 \( \text{elgamal-reduction} \). 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.

have \( \text{game}_3\text{-bad}: \text{map-spmf} \ \text{snd} (\text{?sample} (\text{game}_3 \ ?f')) = \text{lcdh.game} (\text{elgamal-reduction} \) \)

have \( \text{game}_3\text{-guess}: \text{map-spmf} \ \text{fst} (\text{game}_3 \ ?f' \ x \ y) = \text{coin-spmf} \) for \( x \ y \)

5.8 Combining several game hops

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

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

True = 1 / 2

using \( \text{ind-cpa-game-eq-game}_0 \) by(simp add: \( \text{game}_0\text{-game}_1\text{-o-def} \))

67
also have \[ \frac{1}{2} = \text{spmf} (\text{map-spmf} \text{fst} (\text{sample game}_1)) \text{ True} \]
\[ \text{by (simp add: \text{abs-minus-commute})} \]
also have \[ \frac{1}{2} = \text{spmf} (\text{map-spmf} \text{fst} (\text{sample game}_2)) \text{ True} \]
\[ \text{by (simp add: \text{game}_2: \text{game}_3, \text{game}_3: \text{guess} \text{ spmf-of-set})} \]
also have \[ \ldots \leq \text{spmf} (\text{map-spmf} \text{snd} (\text{sample game}_1)) \text{ True} \]
\[ \text{by (rule hop12)} \]
also have \[ \ldots = \text{lcdh}.\text{advantage} (\text{elgamal-reduction} \Rightarrow) \]
\[ \text{by (simp add: \text{game}_2: \text{game}_3, \text{game}_3: \text{guess} \text{ spmf-of-set})} \]
finally show \[ \text{thesis} \]

This completes the concrete proof and we can end the locale \textit{hashed-elgamal}.

\textit{qed}

\textit{end}

\section{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.

\subsection{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 \textit{ind-cpa-pk}.

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

\textbf{locale} \textit{ind-cpa-pk'} =
\textbf{fixes} key-gen :: \textit{security} \Rightarrow \textit{'pubkey} \times \textit{'privkey} \times \textit{'query} \times \textit{'response} \textit{gpv}
\textbf{and} encrypt :: \textit{security} \Rightarrow \textit{'pubkey} \Rightarrow \textit{'plain} \Rightarrow \textit{'cipher} \times \textit{'query} \times \textit{'response} \textit{gpv}
\textbf{and} decrypt :: \textit{security} \Rightarrow \textit{'privkey} \Rightarrow \textit{'cipher} \Rightarrow \textit{'plain} \times \textit{'query} \times \textit{'response} \textit{gpv}
\textbf{and} valid-plains :: \textit{security} \Rightarrow \textit{'plain} \Rightarrow \textit{'plain} \Rightarrow \textit{bool}

\textbf{begin}
\textbf{sublocale} \textit{ind-cpa-pk'} \textit{key-gen} \textit{\eta} encrypt \textit{\eta} decrypt \textit{\eta} valid-plains \textit{\eta} for \textit{\eta} \textit{(proof)}
\textbf{end}

We do so similarly for \textit{list-cdh}, \textit{random-oracle}, and \textit{hashed-elgamal}.

\textbf{locale} \textit{hashed-elgamal'} =
\textit{lcdh}: \textit{list-cdh'} \Rightarrow
random-oracle \( G \):

\[
\text{len-plain} \Rightarrow \text{security} \quad \text{for } G
\]

and

\[
\text{len-plain} \Rightarrow \text{nat}
\]

begin

sublocale hashed-elgamal \( G \) \( \eta \) \( \text{len-plain} \) \( \eta \) for \( \eta \) (proof)

6.2 Asymptotic security statements

For asymptotic security statements, CryptHOL defines the predicate \textit{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 \textit{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 \textit{negligible}.

\textbf{theorem} asymptotic-security-elgamal:

\textbf{assumes} negligible \((\lambda \eta. \text{lcdh.advantage} \eta \text{ (elgamal-reduction } \eta (\mathcal{A} \eta)))\)

\textbf{and} eventually \((\lambda \eta. \text{ind-cpa.lossless} (\mathcal{A} \eta))\) at-top

\textbf{shows} negligible \((\lambda \eta. \text{ind-cpa.advantage} \eta (\mathcal{R}.\text{oracle} \eta, \mathcal{R}.\text{initial}) (\mathcal{A} \eta))\)

The proof is canonical, too: Using the lemmas about \textit{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.

(\textit{proof})

end

References


