Game-based cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar and Bhargav Bhatt

March 17, 2025

Abstract

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

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

Contents

1 Specifying security using games

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

1.1 The DDH game

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

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

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

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

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

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

lemma *lossless-ddh-0*:
⟦ *lossless* $\mathscr{A}$; *0* < *order* $\mathscr{G}$ ⟧
⟹ *lossless-spmf* (*ddh-0* $\mathscr{A}$)
⟨*proof*⟩

lemma *lossless-ddh-1*:
⟦ *lossless* $\mathscr{A}$; *0* < *order* $\mathscr{G}$ ⟧
⟹ *lossless-spmf* (*ddh-1* $\mathscr{A}$)
⟨*proof*⟩

end

1.2 The LCDH game

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

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

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

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

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

lemma *lossless-lcdh*:
 ⟦ *lossless* $\mathcal{A}$; *0* < *order* $\mathcal{G}$ ⟧
 ⟹ *lossless-spmf* (*lcdh* $\mathcal{A}$)
⟨*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*) *option*

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

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

lemma *ekey-oracle-conv*:
ekey-oracle η σ *x* =
(*case* σ *of None* ⇒ *map-spmf* (λ(*ekey*, *dkey*). (*ekey*, *Some* (*ekey*, *dkey*, []))) (*key-gen* η)
| *Some* (*ekey*, *rest*) ⇒ *return-spmf* (*ekey*, *Some* (*ekey*, *rest*)))
⟨*proof*⟩

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

end

1.3.1 Single-user setting

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

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

lemma $oracle_1\text{-}simps$ [$simp$]:
$oracle_1\ b\ \eta\ s\ (Inl\ x) = map\text{-}spmf\ (apfst\ Inl)\ (ekey\text{-}oracle\ \eta\ s\ x)$
$oracle_1\ b\ \eta\ s\ (Inr\ (Inl\ y)) = map\text{-}spmf\ (apfst\ (Inr \circ Inl))\ (decrypt\text{-}oracle\ \eta\ s\ y)$
$oracle_1\ b\ \eta\ s\ (Inr\ (Inr\ z)) = map\text{-}spmf\ (apfst\ (Inr \circ Inr))\ (encrypt\text{-}oracle\ b\ \eta\ s\ z)$
$\langle proof \rangle$

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

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

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

lemma $advantage_1\text{-}nonneg$: $advantage_1\ \mathscr{A}\ \eta \geq 0\ \langle proof \rangle$

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

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

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

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

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

lemma *lossless-decrypt-oracle* [*simp*]: *lossless-spmf* (*decrypt-oracle* η σ *cipher*)
⟨*proof*⟩

lemma *lossless-ekey-oracle* [*simp*]:
lossless-spmf (*ekey-oracle* η σ *x*) ⟷ (σ = *None* ⟶ *lossless-spmf* (*key-gen* η))
⟨*proof*⟩

lemma *lossless-encrypt-oracle* [*simp*]:
⟦ σ = *None* ⟹ *lossless-spmf* (*key-gen* η);
⋀*ekey m*. *valid-plain* η *m* ⟹ *lossless-spmf* (*encrypt* η *ekey m*) ⟧
⟹ *lossless-spmf* (*encrypt-oracle b* η σ (*m0*, *m1*)) ⟷ *valid-plain* η *m0* ∧ *valid-plain* η *m1*
⟨*proof*⟩

1.3.2 Multi-user setting

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

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

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

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

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

lemma *advantage*$_n$*-nonneg*: *advantage*$_n$ $\mathscr{A}$ η ≥ *0* ⟨*proof*⟩

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

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

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

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

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

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

lemma *cipher-queriesI*:
⟦ *ose n* = *Some* (*ek*, *dk*, *ciphers*); *x* ∈ *set ciphers* ⟧ ⟹ *x* ∈ *cipher-queries ose*
⟨*proof*⟩

lemma *cipher-queriesE*:
assumes *x* ∈ *cipher-queries ose*
obtains (*cipher-queries*) *n ek dk ciphers* **where** *ose n* = *Some* (*ek*, *dk*, *ciphers*) *x* ∈ *set ciphers*
⟨*proof*⟩

lemma *cipher-queries-updE*:
assumes *x* ∈ *cipher-queries* (*ose*(*n* ↦ (*ek*, *dk*, *ciphers*)))
obtains (*old*) *x* ∈ *cipher-queries ose x* ∉ *set ciphers* | (*new*) *x* ∈ *set ciphers*
⟨*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* $\Rightarrow$ *bool*
begin

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

definition *oracle-encrypt* :: *'key* $\Rightarrow$ *bool* $\Rightarrow$ (*'message* $\times$ *'message*, *'cipher option*, *'cipher set*) *callee*
where
oracle-encrypt k b L = (λ(*msg1*, *msg0*).
(*case msg-predicate msg1* $\wedge$ *msg-predicate msg0 of*
True $\Rightarrow$ *do* {
c $\leftarrow$ *encrypt k* (*if b then msg1 else msg0*);
return-spmf (*Some c*, {*c*} $\cup$ *L*)
}
| *False* $\Rightarrow$ *return-spmf* (*None*, *L*)))

lemma *lossless-oracle-encrypt* [*simp*]:
assumes *lossless-spmf* (*encrypt k m1*) **and** *lossless-spmf* (*encrypt k m0*)
shows *lossless-spmf* (*oracle-encrypt k b L* (*m1*, *m0*))
$\langle$*proof*$\rangle$

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

lemma *lossless-oracle-decrypt* [*simp*]: *lossless-spmf* (*oracle-decrypt k L c*)
$\langle$*proof*$\rangle$

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

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

lemma *advantage-nonneg*: *0* $\leq$ *advantage* $\mathscr{A}$ $\langle$*proof*$\rangle$

end

end

theory *IND-CPA* **imports**
CryptHOL.Generative-Probabilistic-Value

CryptHOL.Computational-Model
CryptHOL.Negligible
begin

1.5 The IND-CPA game for symmetric encryption schemes

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

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

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

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

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

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

lemma *advantage-nonneg*: *advantage* $\mathscr{A}$ ≥ *0* ⟨*proof*⟩

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

lemma *ibounded-byE* [*consumes 1*, *case-names ibounded-by*, *elim?*]:
assumes *ibounded-by* $(\mathscr{A}1, \mathscr{A}2)\ q$
obtains *q1 q2*
where $q1 + q2 \leq q$
and *interaction-any-bounded-by* $\mathscr{A}1\ q1$
and $\bigwedge cipher\ \sigma.$ *interaction-any-bounded-by* $(\mathscr{A}2\ cipher\ \sigma)\ q2$
⟨*proof*⟩

lemma *ibounded-byI* [*intro?*]:
$[\![$ *interaction-any-bounded-by* $\mathscr{A}1\ q1$; $\bigwedge cipher\ \sigma.$ *interaction-any-bounded-by* $(\mathscr{A}2\ cipher\ \sigma)\ q2$; $q1 + q2 \leq q\ ]\!]$
$\Longrightarrow$ *ibounded-by* $(\mathscr{A}1, \mathscr{A}2)\ q$
⟨*proof*⟩

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

end

end

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

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

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

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

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

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

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

lemma *advantage-nonneg*: *advantage oracle* $\sigma\ \mathscr{A} \geq 0$ $\langle proof \rangle$

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

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

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

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

end

end

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

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

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

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

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

primrec *ind-cpa* :: ($'pub\text{-}key$, $'plain$, $'cipher$, $'state$) *adversary* $\Rightarrow$ *bool spmf*
where
ind-cpa ($\mathscr{A}1$, $\mathscr{A}2$) = *TRY do* {
(pk, sk) $\leftarrow$ *key-gen*;
(($m0$, $m1$), σ) $\leftarrow$ $\mathscr{A}1$ pk;
- :: *unit* $\leftarrow$ *assert-spmf* (*valid-plains m0 m1*);
$b \leftarrow$ *coin-spmf*;
cipher $\leftarrow$ *aencrypt pk* (*if b then m0 else m1*);
$b' \leftarrow \mathscr{A}2$ *cipher* σ;
return-spmf ($b = b'$)
} *ELSE coin-spmf*

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

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

definition *lossless* :: ($'pub\text{-}key$, $'plain$, $'cipher$, $'state$) *adversary* $\Rightarrow$ *bool*
where
lossless $\mathscr{A}$ $\longleftrightarrow$
(($\forall pk$. *lossless-spmf* (*fst* $\mathscr{A}$ pk)) $\wedge$
($\forall$*cipher* σ. *lossless-spmf* (*snd* $\mathscr{A}$ *cipher* σ)))

lemma *lossless-ind-cpa*:
⟦ *lossless* $\mathscr{A}$; *lossless-spmf* (*key-gen*) ⟧ $\Longrightarrow$ *lossless-spmf* (*ind-cpa* $\mathscr{A}$)
⟨*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'*

where
  sign-oracle η *None* $m = do$ { (-, σ) $\leftarrow$ *vkey-oracle* η *None* (); *sign-oracle* η σ *m* }
| $\bigwedge$*log*. *sign-oracle* η (*Some* (*vkey*, *skey*, *log*)) m =
  (*if valid-message* η *m then do* {
    sig $\leftarrow$ *sign* η *skey m*;
    return-spmf (*sig*, *Some* (*vkey*, *skey*, (*m*, *sig*) # *log*))
  } *else return-pmf None*)
$\langle$*proof*$\rangle$
termination $\langle$*proof*$\rangle$
end

lemma *lossless-vkey-oracle* [*simp*]:
  lossless-spmf (*vkey-oracle* η σ *x*) $\longleftrightarrow$ (σ = *None* $\longrightarrow$ *lossless-spmf* (*key-gen* η))
$\langle$*proof*$\rangle$

lemma *lossless-sign-oracle* [*simp*]:
  $[\![$ σ = *None* $\Longrightarrow$ *lossless-spmf* (*key-gen* η);
    $\bigwedge$*skey m*. *valid-message* η *m* $\Longrightarrow$ *lossless-spmf* (*sign* η *skey m*) $]\!]$
  $\Longrightarrow$ *lossless-spmf* (*sign-oracle* η σ *m*) $\longleftrightarrow$ *valid-message* η *m*
$\langle$*proof*$\rangle$

lemma *lossless-sign-oracle-Some*: **fixes** *log* **shows**
  lossless-spmf (*sign-oracle* η (*Some* (*vkey*, *skey*, *log*)) *m*) $\longleftrightarrow$ *lossless-spmf* (*sign* η *skey m*) $\wedge$ *valid-message* η *m*
$\langle$*proof*$\rangle$

1.8.1 Single-user setting

type-synonym *'message' call*$_1$ = *unit* + *'message'*
type-synonym (*'vkey'*, *'signature'*) *ret*$_1$ = *'vkey'* + *'signature'*

definition *oracle*$_1$ :: *security*
  $\Rightarrow$ ((*'vkey*, *'sigkey*, *'message*, *'signature*) *state-oracle*, *'message call*$_1$, (*'vkey*, *'signature*) *ret*$_1$) *oracle'*
where *oracle*$_1$ η = *vkey-oracle* η $\oplus_O$ *sign-oracle* η

lemma *oracle*$_1$*-simps* [*simp*]:
  oracle$_1$ η *s* (*Inl x*) = *map-spmf* (*apfst Inl*) (*vkey-oracle* η *s x*)
  oracle$_1$ η *s* (*Inr y*) = *map-spmf* (*apfst Inr*) (*sign-oracle* η *s y*)
$\langle$*proof*$\rangle$

type-synonym (*'vkey'*, *'message'*, *'signature'*) *adversary*$_1$*'* =
  ((*'message'* $\times$ *'signature'*), *'message' call*$_1$, (*'vkey'*, *'signature'*) *ret*$_1$) *gpv*
type-synonym (*'vkey'*, *'message'*, *'signature'*) *adversary*$_1$ =
  security $\Rightarrow$ (*'vkey'*, *'message'*, *'signature'*) *adversary*$_1$*'*

definition *suf-cma*$_1$ :: (*'vkey*, *'message*, *'signature*) *adversary*$_1$ $\Rightarrow$ *security* $\Rightarrow$ *bool spmf*
where
  $\bigwedge$*log*. *suf-cma*$_1$ $\mathscr{A}$ η = *do* {

$((m, sig), \sigma) \leftarrow$ *exec-gpv* ($oracle_1$ η) ($\mathscr{A}$ η) *None*;
return-spmf (
case σ *of None* $\Rightarrow$ *False*
| *Some* (*vkey*, *skey*, *log*) $\Rightarrow$ *verify* η *vkey m sig* $\wedge$ $(m, sig) \notin$ *set log*)
}

definition $advantage_1$ :: ('*vkey*, '*message*, '*signature*) $adversary_1$ $\Rightarrow$ *advantage*
where $advantage_1$ $\mathscr{A}$ η = *spmf* ($suf\text{-}cma_1$ $\mathscr{A}$ η) *True*

lemma $advantage_1$*-nonneg*: $advantage_1$ $\mathscr{A}$ $\eta \geq 0$ $\langle proof \rangle$

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

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

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

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

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

1.8.2 Multi-user setting

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

lemma $oracle_n$*-apply* [*simp*]:
$oracle_n$ η *s* (*i*, *x*) = *map-spmf* (*apsnd* (*fun-upd s i*)) ($oracle_1$ η (*s i*) *x*)
$\langle proof \rangle$

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

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

| *Some* $(vkey, skey, log) \Rightarrow verify\ \eta\ vkey\ m\ sig \wedge (m, sig) \notin set\ log)$
}

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

lemma $advantage_n$*-nonneg*: $advantage_n\ \mathscr{A}\ \eta \geq 0\ \langle proof \rangle$

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

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

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

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

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

end

end

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

1.9 Pseudo-random function

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

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

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

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

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

lemma *weight-random-oracle* [*simp*]:
weight-spmf p = *1* $\Longrightarrow$ *weight-spmf* (*random-oracle* σ *x*) = *1*
⟨*proof*⟩

lemma *lossless-random-oracle* [*simp*]:
lossless-spmf p $\Longrightarrow$ *lossless-spmf* (*random-oracle* σ *x*)
⟨*proof*⟩

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

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

end

1.10 Pseudo-random function

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

sublocale *random-function rand* ⟨*proof*⟩

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

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

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

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

return-spmf b
}

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

lemma *advantage-nonneg*: *advantage* $\mathscr{A} \geq 0$
⟨*proof*⟩

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

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

end

end

1.11 Random permutation

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

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

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

lemma *weight-random-oracle* [*simp*]:
⟦*finite A*; *A* − *ran* $\sigma \neq \{\}$ ⟧ $\Longrightarrow$ *weight-spmf* (*random-permutation* σ *x*) = *1*
⟨*proof*⟩

lemma *lossless-random-oracle* [*simp*]:
⟦*finite A*; *A* − *ran* $\sigma \neq \{\}$ ⟧ $\Longrightarrow$ *lossless-spmf* (*random-permutation* σ *x*)
⟨*proof*⟩

sublocale *finite*: *callee-invariant-on random-permutation* $\lambda\sigma$. *finite* (*dom* σ) $\mathscr{I}$*-full*
⟨*proof*⟩

lemma *card-dom-random-oracle*:
assumes *interaction-any-bounded-by* $\mathscr{A}$ *q*
and (*y*, σ') ∈ *set-spmf* (*exec-gpv random-permutation* $\mathscr{A}$ σ)

and *fin*: *finite* (*dom* σ)
shows *card* (*dom* σ') $\leq q +$ *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* :: ($'c\text{-}o \times 'c\text{-}a \times 's$) *spmf*
and *oracle* :: $'c\text{-}o \Rightarrow 's \Rightarrow 'call \Rightarrow ('ret \times 's)$ *spmf*
and *eval* :: $'c\text{-}o \Rightarrow 'c\text{-}a \Rightarrow 's \Rightarrow 'guess \Rightarrow$ *bool spmf*
begin

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

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

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

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

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

definition *game-multi* :: ($'c\text{-}a$, $'guess$, $'call$, $'ret$) *adversary-many* $\Rightarrow$ *bool spmf*
where
game-multi $\mathscr{A}$ = *do* {
(*c-o*, *c-a*, *s*) $\leftarrow$ *init*;

$(\text{-}, (b, \text{-})) \leftarrow$ *exec-gpv*
  $(\dagger(\textit{oracle c-o}) \oplus_O \textit{eval-oracle c-o c-a})$
  $(\mathscr{A}\ \textit{c-a})$
  $(\textit{False}, s)$;
 return-spmf b
}

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

type-synonym $'\textit{guess}'$ *reduction-state* = $'\textit{guess}' + \textit{nat}$

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

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

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

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

lemma *many-single-reduction*:
 assumes *bound*: $\bigwedge$*c-a c-o s*. $(\textit{c-o}, \textit{c-a}, s) \in$ *set-spmf init* $\Longrightarrow$ *interaction-bounded-by* $(\textit{Not} \circ \textit{isl})\ (\mathscr{A}\ \textit{c-a})\ q$
 and *lossless-oracle*: $\bigwedge$*c-a c-o s s' x*. $(\textit{c-o}, \textit{c-a}, s) \in$ *set-spmf init* $\Longrightarrow$ *lossless-spmf* $(\textit{oracle c-o}\ s'\ x)$
 and *lossless-eval*: $\bigwedge$*c-a c-o s s' guess*. $(\textit{c-o}, \textit{c-a}, s) \in$ *set-spmf init* $\Longrightarrow$ *lossless-spmf* $(\textit{eval c-o c-a}\ s'\ \textit{guess})$

shows *advantage-multi* $\mathscr{A} \leq$ *advantage-single* (*reduction q* $\mathscr{A}$) $*$ *q*
including *lifting-syntax*
⟨*proof*⟩

end

end

1.13 Unpredictable function

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

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

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

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

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

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

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

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

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

definition *game1* :: $('x, 'hash)$ *adversary1* $\Rightarrow$ *bool spmf*
where
  game1 $\mathscr{A}$ = *do* {
    key $\leftarrow$ *key-gen*;
    $((m, h), L) \leftarrow$ *exec-gpv* (*oracle-hash key*) $\mathscr{A}$ {};
    return-spmf $(h = \textit{hash key } m \wedge m \notin L)$
  }

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

lemma *advantage-advantage1*:
  assumes *bound*: *interaction-bounded-by* (*Not* $\circ$ *isl*) $\mathscr{A}$ *q*
  shows *advantage* $\mathscr{A} \leq$ *advantage1* (*guessing-many-one.reduction q* (λ- :: *unit*. $\mathscr{A}$) ())
$*$ *q*
$\langle$*proof*$\rangle$

end

end

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

end

2 Cryptographic constructions and their security

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

2.1 Elgamal encryption scheme

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

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

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

lemma *key-gen-alt*:
key-gen = *map-spmf* (λx. (**g** [^] *x*, *x*)) (*sample-uniform* (*order* $\mathcal{G}$))
⟨*proof*⟩

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

lemma *aencrypt-alt*:
aencrypt α *msg* = *map-spmf* (λy. (**g** [^] *y*, (α [^] *y*) ⊗ *msg*)) (*sample-uniform* (*order* $\mathcal{G}$))
⟨*proof*⟩

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

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

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

fun *elgamal-adversary* :: (*'grp pub-key*, *'grp plain*, *'grp cipher*, *'state*) *ind-cpa.adversary*
⇒ *'grp ddh.adversary*
where
elgamal-adversary ($\mathcal{A}1$, $\mathcal{A}2$) α β γ = *TRY do* {
$b \leftarrow$ *coin-spmf*;
((*msg1*, *msg2*), σ) $\leftarrow$ $\mathcal{A}1$ α;
— have to check that the attacker actually sends two elements from the group; otherwise

flip a coin
  - :: *unit* ← *assert-spmf* (*valid-plains msg1 msg2*);
  guess ← $\mathscr{A}$*2* (β, $\gamma \otimes$ (*if b then msg1 else msg2*)) σ;
  return-spmf (*guess* = *b*)
} *ELSE coin-spmf*

end

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

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

end

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

sublocale *elgamal* $\mathscr{G}$ η **for** η ⟨*proof*⟩

theorem *elgamal-secure*:
 negligible ($\lambda\eta$. *ind-cpa.advantage* η ($\mathscr{A}$ η)) **if** *negligible* ($\lambda\eta$. *ddh.advantage* η (*elgamal-adversary* η ($\mathscr{A}$ η)))
 ⟨*proof*⟩

end

context *elgamal-base* **begin**

lemma *lossless-key-gen* [*simp*]: *lossless-spmf* (*key-gen*) ⟷ *0* < *order* $\mathscr{G}$
⟨*proof*⟩

lemma *lossless-aencrypt* [*simp*]:
 lossless-spmf (*aencrypt key plain*) ⟷ *0* < *order* $\mathscr{G}$
⟨*proof*⟩

lemma *lossless-elgamal-adversary*:
 ⟦ *ind-cpa.lossless* $\mathscr{A}$; *0* < *order* $\mathscr{G}$ ⟧
 ⟹ *ddh.lossless* (*elgamal-adversary* $\mathscr{A}$)
⟨*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 *ibound*: *interaction-any-bounded-by gpv n*
shows *card* (*dom* σ') ≤ *n* + *card* (*dom* σ)

⟨*proof*⟩

end

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* $\mathscr{G}$ ⟨*proof*⟩

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

end

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

interpretation *cyclic-group* $\mathscr{G}$ ⟨*proof*⟩

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

end

context *elgamal-base* **begin**

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

lemma *lossless-elgamal-adversary*:
⟦ *ind-cpa.lossless* $\mathscr{A}$; $\bigwedge\eta$. *0* < *order* $\mathscr{G}$ ⟧
⟹ *lcdh.lossless* (*elgamal-adversary* $\mathscr{A}$)
⟨*proof*⟩

end

end

2.3 The random-permutation random-function switching lemma

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

lemma *rp-resample*:
assumes $B \subseteq A \cup C$ $A \cap C = \{\}$ $C \subseteq B$ **and** *finB*: *finite B*
shows *bind-spmf* (*spmf-of-set B*) (λx. *if* $x \in A$ *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*: $A \neq \{\}$
begin

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

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

lemma *advantage-nonneg*: $0 \leq$ *advantage* $\mathscr{A}$ ⟨*proof*⟩

lemma *advantage-le-1*: *advantage* $\mathscr{A} \leq 1$
⟨*proof*⟩

context includes $\mathscr{I}$*.lifting* **begin**
lift-definition $\mathscr{I}$ :: (*'a*, *'a*) $\mathscr{I}$ **is** (λx. *if* $x \in A$ *then A else* $\{\}$) ⟨*proof*⟩
lemma *outs-*$\mathscr{I}$*-*$\mathscr{I}$ [*simp*]: *outs-*$\mathscr{I}$ $\mathscr{I}$ = *A* ⟨*proof*⟩
lemma *responses-*$\mathscr{I}$*-*$\mathscr{I}$ [*simp*]: *responses-*$\mathscr{I}$ $\mathscr{I}$ *x* = (*if* $x \in A$ *then A else* $\{\}$) ⟨*proof*⟩

lifting-update $\mathscr{I}$*.lifting*
lifting-forget $\mathscr{I}$*.lifting*
end

lemma *rp-rf*:
assumes *bound*: *interaction-any-bounded-by* $\mathscr{A}$ *q*
and *lossless*: *lossless-gpv* $\mathscr{I}$ $\mathscr{A}$
and *WT*: $\mathscr{I} \vdash g\ \mathscr{A}\ \surd$
shows *advantage* $\mathscr{A} \leq q * q\ /\ card\ A$
including *lifting-syntax*
⟨*proof*⟩

end

end

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

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

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

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

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

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

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

lemma ε*-uh-nonneg* : ε*-uh* ≥ *0*
⟨*proof*⟩

lemma *hash-ineq-card*:
  assumes *finite W*
  shows *spmf* (*game-hash-set W*) *True* $\leq$ ε*-uh* $*$ *card W* $*$ *card W*
⟨*proof*⟩

end

locale *prf-hash* =
  fixes *f* :: $'key \Rightarrow {}'\alpha \Rightarrow {}'\gamma$
  and *h* :: $'seed \Rightarrow {}'\beta \Rightarrow {}'\alpha$
  and *key-gen* :: *'key spmf*
  and *seed-gen* :: *'seed spmf*
  and *range-f* :: $'\gamma$ *set*
  assumes *lossless-seed-gen*: *lossless-spmf seed-gen*
  and *range-f-finite*: *finite range-f*
  and *range-f-nonempty*: *range-f* $\neq \{\}$
begin

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

interpretation *prf'*: *prf key-seed-gen f' rand* ⟨*proof*⟩

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

definition *prf'-reduction* :: $({}'\beta, {}'\gamma)$ *prf'.adversary* $\Rightarrow$ $({}'\alpha, {}'\gamma)$ *prf.adversary*
where
  prf'-reduction $\mathscr{A}$ = *do* {
    seed ← *lift-spmf seed-gen*;
    (*b*, σ) ← *inline* (*reduction-oracle seed*) $\mathscr{A}$ ();

Done b
}

theorem *prf-prf′-advantage*:
assumes *prf′.lossless* $\mathscr{A}$
and *bounded*: *prf′.ibounded-by* $\mathscr{A}$ *q*
shows *prf′.advantage* $\mathscr{A}$ $\leq$ *prf.advantage* (*prf′-reduction* $\mathscr{A}$) + *hash.*ε*-uh* ∗ *q* ∗ *q*
including *lifting-syntax*
⟨*proof*⟩ **including** *monad-normalisation*
⟨*proof*⟩

end

end

2.5 IND-CPA from PRF

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

Formalises the construction from [16].

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

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

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

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

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

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

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

lemma *encrypt-decrypt-correct*:
 ⟦ *length key* = *len*; *length plain* = *len* ⟧
 ⟹ *encrypt key plain* ⤜ (λ*cipher*. *return-spmf* (*decrypt key cipher*)) = *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 (λ*pad*. *Done* ((*r*, *xor-list plain pad*), ()))
 }

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

lemma *lossless-prf-encyrpt-oracle* [*simp*]: *lossless-gpv* ℐ*-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 (𝒜*1*, 𝒜*2*) = 𝒜;
  (((*p1*, *p2*), σ), *n*) ← *inline prf-encrypt-oracle* 𝒜*1* ();
  if valid-plain p1 ∧ *valid-plain p2 then do* {
   b ← *lift-spmf coin-spmf*;
   let pb = (*if b then p1 else p2*);
   r ← *lift-spmf* (*spmf-of-set* (*nlists UNIV len*));
   pad ← *Pause r Done*;
   let c = (*r*, *xor-list pb pad*);
   (*b*′, -) ← *inline prf-encrypt-oracle* (𝒜*2 c* σ) *n*;
   Done (*b*′ = *b*)
  } *else lift-spmf coin-spmf*
 }

theorem *prf-encrypt-advantage*:
 assumes *ind-cpa.ibounded-by* 𝒜 *q*
 and *lossless-gpv* ℐ*-full* (*fst* 𝒜)
 and ⋀*cipher* σ. *lossless-gpv* ℐ*-full* (*snd* 𝒜 *cipher* σ)
 shows *ind-cpa.advantage* 𝒜 ≤ *prf*.*advantage* (*prf-adversary* 𝒜) + *q* / *2* ^ *len*
⟨*proof*⟩

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* $\mathscr{A}$ *q*
shows *prf.ibounded-by* (*prf-adversary* $\mathscr{A}$) (*1* + *q*)
⟨*proof*⟩

lemma *lossless-prf-adversary*: *ind-cpa.lossless* $\mathscr{A}$ $\Longrightarrow$ *prf.lossless* (*prf-adversary* $\mathscr{A}$)
⟨*proof*⟩

end

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

interpretation *otp f* η *len* η **for** η ⟨*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* ($\lambda\eta$. *prf.advantage* η (*prf-adversary* η ($\mathscr{A}$ η)))
and *q*: $\bigwedge\eta$. *ind-cpa.ibounded-by* ($\mathscr{A}$ η) (*q* η) **and** [*negligible-intros*]: *polynomial q*
and *lossless*: $\bigwedge\eta$. *ind-cpa.lossless* ($\mathscr{A}$ η)
shows *negligible* ($\lambda\eta$. *ind-cpa.advantage* η ($\mathscr{A}$ 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* :: $('prf\text{-}key \times 'upf\text{-}key) \Rightarrow 'hash\ cipher\text{-}text \Rightarrow bitstring\ option$
where
  decrypt $(k\text{-}prf, k\text{-}upf)\ (x, c, t) = ($
    if upf-fun k-upf $(x\ @\ c) = t \wedge$ *length* $x =$ *prf-dlen then*
      Some $(prf\text{-}fun\ k\text{-}prf\ x\ [\oplus]\ c)$
    else
      None
  $)$

lemma *cipher-correct*:
  $[\![\ k \in set\text{-}spmf\ key\text{-}gen;\ length\ m = prf\text{-}clen\ ]\!]$
  $\Longrightarrow encrypt\ k\ m \gg\!= (\lambda c.\ return\text{-}spmf\ (decrypt\ k\ c)) = return\text{-}spmf\ (Some\ m)$
$\langle proof \rangle$

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

sublocale *ind-cca*: *ind-cca key-gen encrypt decrypt* $\lambda m.\ length\ m = prf\text{-}clen$ $\langle proof \rangle$
interpretation *ind-cca'*: *ind-cca key-gen encrypt* λ - -. *None* $\lambda m.\ length\ m = prf\text{-}clen$
$\langle proof \rangle$

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

definition *intercept-upf-dec*
  :: $'hash\ cipher\text{-}text\ set \times 'hash\ cipher\text{-}text\ set \Rightarrow 'hash\ cipher\text{-}text$
  $\Rightarrow (bitstring\ option \times ('hash\ cipher\text{-}text\ set \times 'hash\ cipher\text{-}text\ set),$
    $bitstring + (bitstring \times 'hash), 'hash + unit)\ gpv$
where
  $intercept\text{-}upf\text{-}dec = (\lambda (L, D)\ (x, c, t).$
    $if\ (x, c, t) \in L \vee length\ x \neq prf\text{-}dlen\ then\ Done\ (None, (L, D))\ else\ do\ \{$
      $Pause\ (Inr\ (x\ @\ c, t))\ Done;$
      $Done\ (None, (L, insert\ (x, c, t)\ D))$
    $\})$

definition *intercept-upf* ::
  $'prf\text{-}key \Rightarrow bool \Rightarrow 'hash\ cipher\text{-}text\ set \times 'hash\ cipher\text{-}text\ set \Rightarrow bitstring \times bitstring$
$+\ 'hash\ cipher\text{-}text$

$\Rightarrow$ (('*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* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) (*intercept-upf-enc k b LD mm*)
⟨*proof*⟩

lemma *lossless-intercept-upf-dec* [*simp*]: *lossless-gpv* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) (*intercept-upf-dec LD mm*)
⟨*proof*⟩

lemma *lossless-intercept-upf* [*simp*]: *lossless-gpv* ($\mathscr{I}$*-full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$*-full*) (*intercept-upf k b*

LD x)
⟨*proof*⟩

lemma *results-gpv-intercept-upf* [*simp*]: *results-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) (*intercept-upf k b LD x*) $\subseteq$ *responses-*$\mathcal{I}$ ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) *x* $\times$ *UNIV*
⟨*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 ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) $\mathcal{A}$ $\Longrightarrow$ *lossless-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) (*reduction-upf* $\mathcal{A}$)
⟨*proof*⟩

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

lemma *round-1*:
 assumes *lossless-gpv* ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) $\mathcal{A}$
 shows |*spmf* (*ind-cca.game* $\mathcal{A}$) *True* − *spmf* (*ind-cca'.game* $\mathcal{A}$) *True*| $\leq$ *UPF.advantage* (*reduction-upf* $\mathcal{A}$)
⟨*proof*⟩ **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 = (λ(*k-prf*, *k-upf*) *b D* (*msg1*, *msg0*). (*case* (*length msg1* = *prf-clen* $\wedge$ *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* [$\oplus$] (*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* = (λ*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 $\mathscr{A}$ ≡ *do* {
key ← *key-gen*;
b ← *coin-spmf*;
(*b*′, *D*) ← *exec-gpv*
(*oracle-encrypt2 key b* $\oplus_O$ *oracle-decrypt2 key*) $\mathscr{A}$ *Map-empty*;
return-spmf (*b* = *b*′)
}

fun *intercept-prf* ::
′*upf-key* ⇒ *bool* ⇒ *unit* ⇒ (*bitstring* × *bitstring*) + ′*hash cipher-text*
⇒ ((′*hash cipher-text option* + *bitstring option*) × *unit*, *bitstring*, *bitstring*) *gpv*
where
intercept-prf - - - (*Inr* -) = *Done* (*Inr None*, ())
| *intercept-prf k b* - (*Inl* (*m1*, *m0*)) = (*case* (*length m1*) = *prf-clen* ∧ (*length m0*) = *prf-clen of*
False ⇒ *Done* (*Inl None*, ())
| *True* ⇒ *do* {
x ← *lift-spmf* (*spmf-of-set prf-domain*);
p ← *Pause x Done*;
let c = *p* [⊕] (*if b then m1 else m0*);
let t = *upf-fun k* (*x* @ *c*);
Done (*Inl* (*Some* (*x*, *c*, *t*)), ())
})

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

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

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

lemma *lossless-oracle-encrypt3* [*simp*]:
$lossless\text{-}spmf\ (oracle\text{-}encrypt3\ k\ b\ D\ m10)$
$\langle proof \rangle$

lemma *callee-invariant-oracle-encrypt3* [*simp*]: $callee\text{-}invariant\ (oracle\text{-}encrypt3\ key\ b)\ fst$
$\langle proof \rangle$

definition *game3* :: $(bitstring, 'hash\ cipher\text{-}text)\ ind\text{-}cca.adversary \Rightarrow (bool \times bool)\ spmf$

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

lemma *round-3*:
assumes $lossless\text{-}gpv\ (\mathscr{I}\text{-}full \oplus_{\mathscr{I}} \mathscr{I}\text{-}full)\ \mathscr{A}$
shows $|measure\ (measure\text{-}spmf\ (game3\ \mathscr{A}))\ \{(b, bad).\ b\} - spmf\ (game2\ \mathscr{A})\ True|$
$\leq measure\ (measure\text{-}spmf\ (game3\ \mathscr{A}))\ \{(b, bad).\ bad\}$
$\langle proof \rangle$

lemma *round-4*:
assumes $lossless\text{-}gpv\ (\mathscr{I}\text{-}full \oplus_{\mathscr{I}} \mathscr{I}\text{-}full)\ \mathscr{A}$

shows *map-spmf fst* (*game3* $\mathscr{A}$) = *coin-spmf*
⟨*proof*⟩ **including** *monad-normalisation*
⟨*proof*⟩

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

theorem *security*:
assumes *lossless*: *lossless-gpv* ($\mathscr{I}$-*full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$-*full*) $\mathscr{A}$
and *bound*: *interaction-bounded-by isl* $\mathscr{A}$ *q*
shows *ind-cca.advantage* $\mathscr{A} \leq$
PRF.advantage (*reduction-prf* $\mathscr{A}$) + *UPF.advantage* (*reduction-upf* $\mathscr{A}$) +
real q / *real* (*card prf-domain*) $*$ *real q* (**is** *?LHS* $\leq$ -)
⟨*proof*⟩

theorem *security1*:
assumes *lossless*: *lossless-gpv* ($\mathscr{I}$-*full* $\oplus_{\mathscr{I}}$ $\mathscr{I}$-*full*) $\mathscr{A}$
assumes *q*: *interaction-bounded-by isl* $\mathscr{A}$ *q*
and q': *interaction-bounded-by* (*Not* $\circ$ *isl*) $\mathscr{A}$ q'
shows *ind-cca.advantage* $\mathscr{A} \leq$
PRF.advantage (*reduction-prf* $\mathscr{A}$) +
UPF.advantage1 (*guessing-many-one.reduction* q' (λ-. *reduction-upf* $\mathscr{A}$) ()) $* q'$ +
real q $*$ *real q* / *real* (*card prf-domain*)
⟨*proof*⟩

end

end

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

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

⟨*proof*⟩

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

end

end

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

end

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

end

A Tutorial Introduction to CryptHOL

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

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

Abstract

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

3 Introduction

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

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

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

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

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

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

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

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

3.1 Getting started

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

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

3.2 Getting started

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

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

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

4 Modelling cryptography using CryptHOL

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

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

4.1 Security notions without oracles: the CDH assumption

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

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

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

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

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

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

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

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

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

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

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

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

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

type-synonym $'grp'$ *adversary* $= 'grp' \Rightarrow 'grp' \Rightarrow 'grp'$ *set spmf*

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

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

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

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

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

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

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

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

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

end

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

4.2 A Random Oracle

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

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

type-synonym *bitstring* = *bool list*

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

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

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

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

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

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

abbreviation (*input*) *initial* :: *'a state* **where** *initial* ≡ *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, *ind-cpa-pk*.

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

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

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

4.4 Security notions with oracles: IND-CPA security

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

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

Stage 1 (find):

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

Stage 2 (guess):

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

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

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

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

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

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

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

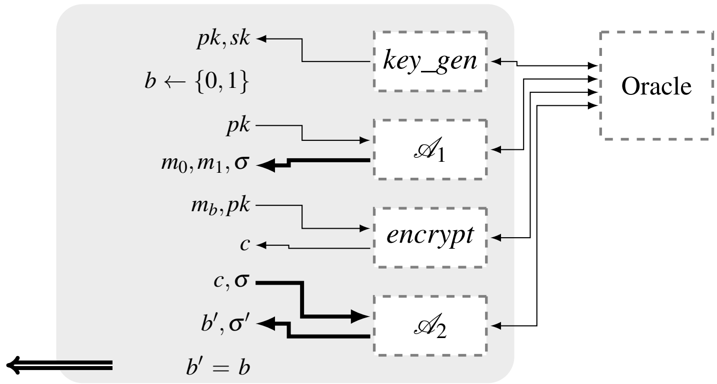

Figure 1: Graphic representation of the generic IND-CPA game.

$\quad$ *cipher* $\leftarrow$ *encrypt pk* (*if b then* m_0 *else* m_1);
$\quad$ $b' \leftarrow \mathscr{A}_2$ *cipher* σ;
$\quad$ *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* :: ($'\sigma$, $'query$, $'response$) $oracle'$ $\times$ $'\sigma$
$\quad\Rightarrow$ ($'pubkey$, $'plain$, $'cipher$, $'query$, $'response$, $'state$) *adversary* $\Rightarrow$ *real*
where
$\quad$ *advantage* (*oracle*, σ) $\mathscr{A}$ = |*spmf* (*run-gpv oracle* (*game* $\mathscr{A}$) σ) *True* $-$ *1/2*|

end

4.5 Concrete cryptographic constructions: the hashed ElGamal encryption scheme

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

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

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

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

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

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

Key generation Pick an element x randomly from the set $\{0,\ldots,q-1\}$ 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 (α,β), output $h(\alpha^{sk})\oplus\beta$.

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

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

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

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

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

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

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

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

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

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

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

sublocale *ind-cpa*: *ind-cpa-pk* (*lift-spmf key-gen*) *encrypt decrypt valid-plains* $\langle$*proof*$\rangle$

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

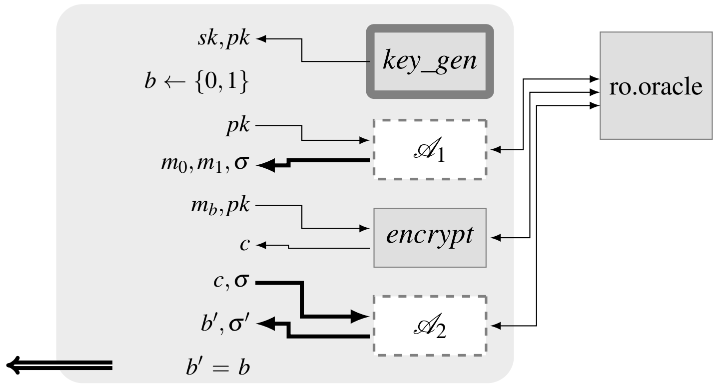

Figure 2: The IND-CPA game instantiated with the Hashed-ElGamal encryption scheme and accessing a random oracle.

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

5 Cryptographic proofs in CryptHOL

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

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

5.1 The reduction

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

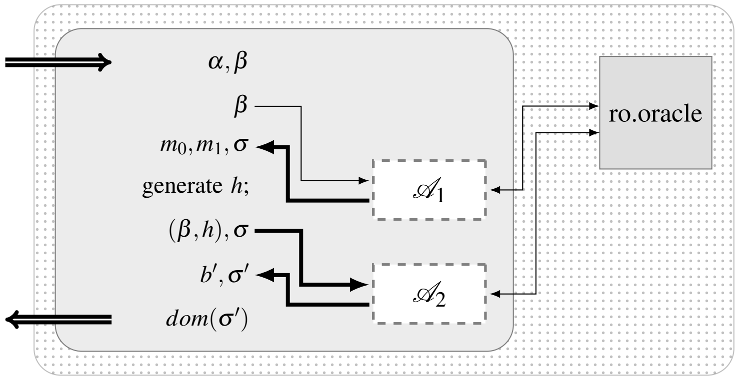

Figure 3: The reduction for the Elgamal security proof.

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

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

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

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

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

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

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

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

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

5.2 Concrete security statement

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

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

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

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

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

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

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

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

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

So now let's start with the proof.

proof –

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

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

5.3 Recording adversary queries

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

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

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

define $game_0$ **where**

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

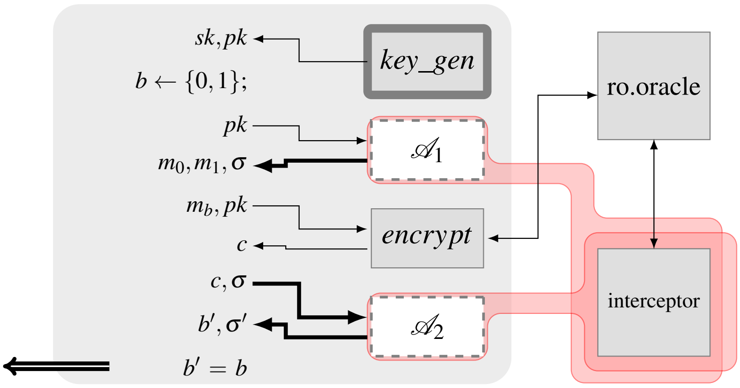

Figure 4: The IND-CPA game after expanding the key generation algorithm's definition and inlining the query-recording hash oracle. The red boxes represent the inline operator.

$game_0 = TRY\ do\ \{$
$\quad (pk, \text{-}) \leftarrow lift\text{-}spmf\ key\text{-}gen;$
$\quad (((m_1, m_2), \sigma), s) \leftarrow inline\ interceptor\ (\mathscr{A}_1\ pk)\ \{\};$
$\quad assert\text{-}gpv\ (valid\text{-}plains\ m_1\ m_2);$
$\quad b \leftarrow lift\text{-}spmf\ coin\text{-}spmf;$
$\quad c \leftarrow encrypt\ pk\ (if\ b\ then\ m_1\ else\ m_2);$
$\quad (b', s') \leftarrow inline\ interceptor\ (\mathscr{A}_2\ c\ \sigma)\ s;$
$\quad Done\ (b' = b)$
$\}\ ELSE\ lift\text{-}spmf\ coin\text{-}spmf$

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

More precisely, we have been very careful that the two games *game* $\mathscr{A}$ and $game_0$ have identical structure. They differ only in that $game_0$ uses the adversary $(\lambda pk.\ inline\ interceptor\ (\mathscr{A}_1\ pk)\ \emptyset,\ \lambda cipher\ \sigma.\ inline\ interceptor\ (\mathscr{A}_2\ cipher\ \sigma))$ instead of $\mathscr{A}$. The formal justification for this replacement happens in two steps:

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

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

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

The second step is automated using Isabelle's simplifier.

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

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

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

5.4 Equational program transformations

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

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

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

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

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

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

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

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

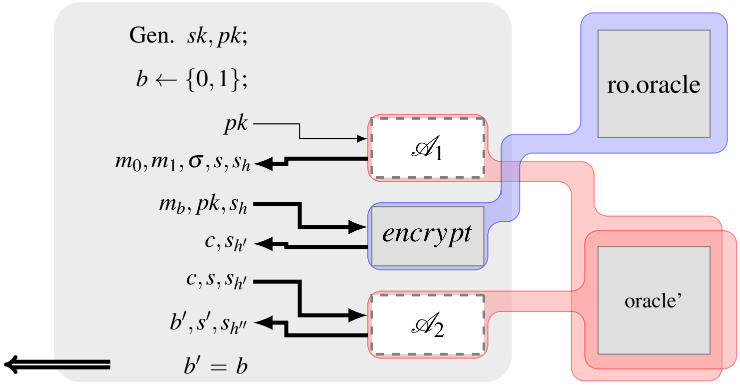

Figure 5: The IND-CPA game after flattening. The blue box around the encryption algorithm and the random oracle represents the expanded definition of them.

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

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

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

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

5.5 Capturing a failure event

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

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

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

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

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

5.6 Game hop based on a failure event

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

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

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

have *rel-spmf* (λ(*win*, *bad*) (*win'*, *bad'*). *bad* = *bad'* ∧ (¬ *bad'* ⟶ *win* = *win'*)) ($game_2$ *x y*) ($game_1$ *x y*) **for** *x y*
proof −

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

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

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

let *?X* = λ(*s*, *s-h*) (*s'*, *s-h'*). *s* = *dom s-h* ∧ *s'* = *s* ∧ *s-h* = *s-h'*(**g** [^] (*x* ∗ *y*) := *None*)

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

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

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

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

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

show *?thesis*
unfolding *game*$_1$*-def game*$_2$*-def*
— Peel off the first phase of the game using the structural decomposition rules *rel-spmf-bind-reflI* and *rel-spmf-try-spmf*.
apply(*clarsimp intro*!: *rel-spmf-bind-reflI simp del*: *bind-spmf-const*)
apply(*rule rel-spmf-try-spmf*)
subgoal *TRY* **for** $b\ m_1\ m_2\ \sigma\ s$ *s-h*
apply(*rule rel-spmf-bind-reflI*)
— Exploit that in the first phase of the game, the set *s* of queried strings and the map of the random oracle *s-h* are updated in lock step, i.e., $s = dom$ *s-h*.
apply(*drule inv-oracle′.exec-gpv-invariant*; *clarsimp*)
— Has the adversary queried the random oracle with **g** [^] $(x * y)$ during the first phase?
apply(*cases* **g** [^] $(x * y) \in s$)
subgoal *True* — Then the failure event has already happened and there is nothing more to do. We just have to prove that the two games on both sides terminate with the same probability.
by(*auto intro*!: *rel-spmf-bindI1 rel-spmf-bindI2 lossless-exec-gpv*[**where** $\mathscr{I}=\mathscr{I}$*-full*] *dest*!: *bad.exec-gpv-invariant*)
subgoal *False* — Then let's see whether the adversary queries **g** [^] $(x * y)$ in the second phase. Thanks to *ro.fresh*, the call to the random oracle simplifies to sampling a random bitstring.
apply(*clarsimp iff del*: *domIff simp add*: *domIff ro.fresh intro*!: *rel-spmf-bind-reflI*)
apply(*rule rel-spmf-bindI*[**where** *R*=*?R*])
— The lemma *exec-gpv-oracle-bisim-bad-full* lifts the bisimulation for *oracle′* to the adversary $\mathscr{A}_2$ interacting with *oracle′*.
apply(*rule exec-gpv-oracle-bisim-bad-full*[*OF - - bisim inv inv*])
apply(*auto simp add*: *fun-upd-idem*)
done
done
subgoal *ELSE* **by**(*rule rel-spmf-reflI*) *clarsimp*
done
qed

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

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

hence $|$*measure* (*measure-spmf* (*?sample* $game_2$)) $\{$(*win*, -). *win*$\}$ $-$ *measure* (*measure-spmf* (*?sample* $game_1$)) $\{$(*win*, -). *win*$\}|$
$\leq$ *measure* (*measure-spmf* (*?sample* $game_2$)) $\{$(-, *bad*). *bad*$\}$
unfolding *split-def* **by**(*rule fundamental-lemma*)
moreover

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

have *measure* (*measure-spmf* (*?sample* $game_2$)) $\{$(*win*, -). *win*$\}$ = *spmf* (*map-spmf fst* (*?sample* $game_2$)) *True*
and *measure* (*measure-spmf* (*?sample* $game_1$)) $\{$(*win*, -). *win*$\}$ = *spmf* (*map-spmf fst* (*?sample* $game_1$)) *True*
and *measure* (*measure-spmf* (*?sample* $game_2$)) $\{$(-, *bad*). *bad*$\}$ = *spmf* (*map-spmf snd* (*?sample* $game_2$)) *True*
unfolding *spmf-conv-measure-spmf measure-map-spmf* **by**(*auto simp add*: *vimage-def split-def*)
ultimately have *hop12*:
$|$*spmf* (*map-spmf fst* (*?sample* $game_2$)) *True* $-$ *spmf* (*map-spmf fst* (*?sample* $game_1$)) *True*$|$
$\leq$ *spmf* (*map-spmf snd* (*?sample* $game_2$)) *True*
by *simp*

5.7 Optimistic sampling: the one-time-pad

This step is based on the one-time-pad, which is an instance of optimistic sampling. If two runs of the two games in an optimistic sampling step would use the same random bits, then their results would be different. However, if the adversary's choices are independent of the random bits, we may relate runs that use different random bits, as in the end, only the probabilities have to match. The previous game hop from $game_1$ to $game_2$ made the oracle's responses in the second phase independent from the encrypted ciphertext. So we can now change the bits used for encrypting the chosen message and thereby make the ciphertext independent of the message.

To that end, we parametrize $game_2$ by the part that does the optimistic sampling and call this parametrized version $game_3$.

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

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

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

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

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

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

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

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

5.8 Combining several game hops

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

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

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

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

qed

end

6 Asymptotic security

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

6.1 Introducing a security parameter

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

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

locale *ind-cpa-pk'* =
fixes *key-gen* :: *security* $\Rightarrow$ (*'pubkey* $\times$ *'privkey*, *'query*, *'response*) *gpv*
and *encrypt* :: *security* $\Rightarrow$ *'pubkey* $\Rightarrow$ *'plain* $\Rightarrow$ (*'cipher*, *'query*, *'response*) *gpv*
and *decrypt* :: *security* $\Rightarrow$ *'privkey* $\Rightarrow$ *'cipher* $\Rightarrow$ (*'plain*, *'query*, *'response*) *gpv*
and *valid-plains* :: *security* $\Rightarrow$ *'plain* $\Rightarrow$ *'plain* $\Rightarrow$ *bool*
begin
sublocale *ind-cpa-pk key-gen* η *encrypt* η *decrypt* η *valid-plains* η **for** η $\langle$*proof*$\rangle$
end

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

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

ro: *random-oracle′ len-plain*
for $\mathcal{G}$ :: *security* ⇒ *′grp cyclic-group*
and *len-plain* :: *security* ⇒ *nat*
begin
sublocale *hashed-elgamal* $\mathcal{G}$ η *len-plain* η **for** η ⟨*proof*⟩

6.2 Asymptotic security statements

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

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

theorem *asymptotic-security-elgamal*:
assumes *negligible* ($\lambda\eta$. *lcdh.advantage* η (*elgamal-reduction* η ($\mathcal{A}$ η)))
and *eventually* ($\lambda\eta$. *ind-cpa.lossless* ($\mathcal{A}$ η)) *at-top*
shows *negligible* ($\lambda\eta$. *ind-cpa.advantage* η (*ro.oracle* η, *ro.initial*) ($\mathcal{A}$ η))

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

⟨*proof*⟩

end

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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