Labreport

Gamehop of ElGamal-Encryption via DDH implemented (and proven) in EasyCrypt

Jakob Nussbaumer

3 August 2018

Contents

1 Introduction

Mathematical proofs are difficult to verify by a human and even those verifications are error prone. This issue is everlasting and problematic in the field of cryptography. For this reason automatic provers and computer-aided toolsets are on the rise to achieve verification of proofs. In a project called miTLS [4] EasyCrypt was used to show that the TLS Handshake is secure [1]. We want to achieve something similar for IPsec. The first step, understanding EasyCrypt and implementing a small example that can be used, is presented in this report. We will go through the cryptographic language of games, similar to the one used by Katz and Lindell (2016) [3], and transfer this into a computer-aided toolset, called EasyCrypt.

Chapter 2 contains the games and schemes in the cryptographic-language that are later on implemented in EasyCrypt.
Parts of the Game Hopping Lemma are introduced together with the implemented gamehop in chapter 3.
Chapter 4 provides a short overview of the most basic things one needs to know to understand EasyCrypt and how it works.
In chapter 5 the full implementation is explained stepwise. An example how the Proof-Engine of EasyCrypt works can be seen at section 5.3.1 with a small overview of the used tactics.
Last there is a quick recap in chapter 6 for explaining pros and cons of using EasyCrypt.

2 Preliminaries

Before the Game Hopping Lemma can be introduced and applied, a few definitions are needed:
The ***advantage*** of an adversary $\mathcal{A}$ playing a game G is defined as

$$\text{adv}^G(A) = |\text{prob}(\mathcal{A} \text{ wins } G) - \text{prob}(\text{guess(G)})|$$

where guess(G) is just a random guess to try to win G. From this point onward every game is assumed to have a random guessing chance of 1/2.
The ***advantage*** of a game G is defined as the maximum advantage over all probabilistic polynomial time (short ***ppt***) adversaries that play G. If a game G has advantage 0, this is called ***null-game*** or ***guessing game***.
Let $\mathcal{K}$ be the space of the keys, $\mathcal{M}$ be the space of the messages and $\mathcal{C}$ the space of the ciphertexts. An encryption scheme is ***indistinguishable under chosen message attacks (IND-CPA secure)***, if every probabilistic polynomial time attacker has at most neglibile (meaning smaller than any inverse polynomial) advantage in winning the following game:

Game: IND-CPA

Parameters: encryption scheme

1. Prepare a key $k \in \mathcal{K}$.
2. Uniformly random pick hidden bit $h_{LOR} \in \{0, 1\}$.
3. Prepare encryption oracle $\mathcal{O}_{Enc}$.
   When called with $m \in \mathcal{M}$ the oracle returns $c \leftarrow Enc_k(m)$.
4. Prepare a one-time oracle $\mathcal{O}_{LOR}$.
   When called with $m_0, m_1 \in \mathcal{M}$ the oracle returns $c \leftarrow Enc_k(m_h)$.
5. Call the adversary with input $(\mathcal{O}_{Enc}, \mathcal{O}_{LOR})$ and await guess $h'_{LOR} \in \{0, 1\}$.
6. If $h_{LOR} = h'_{LOR}$ then return **ACCEPT** else return **REJECT** .

The implementation in EasyCrypt is used to show that the ElGamal encryption scheme is IND-CPA secure.

The ElGamal encryption scheme is defined with:

Encryption Scheme: ElGamal

Parameters: a group G, an element $g \in G$ of order p.

KeyGen of input $\kappa \in \mathcal{K}$:

1. Uniformly random pick $x \in \{0, \ldots, p\}$, this is the secret key.
2. Save g^x as the public key.

Encryption of input $m \in \mathcal{M}$:

1. Uniformly random pick $y \in \{0, \ldots, p\}$.
2. Output $c = (g^y, g^{x*y} * m)$.

Decryption of input $c \in \mathcal{C}$:

1. Read c as the tuple (c_0, c_1).
2. Compute $m = c_0^{-x} * c_1$.

To show the security property of ElGamal, one has to assume that the Decisional Diffie-Hellman problem is not solvable in polynomial time. This is equivalent to having at most negligible advantage for the following game:

Game: DDH

Parameters: a group G, an element $g \in G$ of order p.

1. Uniformly random pick three integers $x, y, z_1 \in \{0, \ldots, p\}$.
   Compute $z_0 = x * y$.
2. Uniformly random pick hidden bit $h_{DDH} \in \{0, 1\}$.
   Define $z = z_{h_{DDH}}$.
3. Call the adversary with input (g^x, g^y, g^z) and await guess $h'_{DDH} \in \{0, 1\}$.
4. If $h_{DDH} = h'_{DDH}$ then return **ACCEPT** else return **REJECT** .

For simplicity's sake, the game IND-CPA with ElGamal as the encryption scheme will combined into one game:

Game: CPA_{real}

Parameters: a group G, an element $g \in G$ of order p.

1. Uniformly random pick $x \in \{0, \ldots, p\}$, this is the secret key.
2. Save g^x as the public key.
3. Uniformly random pick hidden bit $h_{LOR} \in \{0, 1\}$.
4. Prepare encryption oracle $\mathcal{O}_{Enc}$.
   When called with $m \in \mathcal{M}$ the oracle uniformly random picks $y \in \{0, \ldots, p\}$ and returns $c = (g^y, g^{(x*y)} * m)$.
5. Prepare a one-time oracle $\mathcal{O}_{LOR}$.
   When called with $m_0, m_1 \in \mathcal{M}$ the oracle uniformly random picks $y \in \{0, \ldots, p\}$ and returns $c = (g^y, g^{(x*y)} * m_h)$.
6. Call the adversary with input $(\mathcal{O}_{Enc}, \mathcal{O}_{LOR})$ and await guess $h'_{LOR} \in \{0, 1\}$.
7. If $h_{LOR} = h'_{LOR}$ then return **ACCEPT** else return **REJECT** .

The following game is needed later on in the Game Hopping Lemma. It is the same as CPA_{real} besides the one-time oracle, which differs in the created ciphertext. Instead of $g^{x*y} * m_h$ as the output $g^z * m_h$ is used, where z is randomly generated.

Game: $\text{CPA}_{\text{random}}$

Parameters: a group G, an element $g \in G$ of order p.

1. Uniformly random pick $x \in \{0, \ldots, p\}$, this is the secret key.
2. Save g^x as the public key.
3. Uniformly random pick hidden bit $h_{LOR} \in \{0, 1\}$.
4. Prepare encryption oracle $\mathcal{O}_{Enc}$.
   When called with $m \in \mathcal{M}$ the oracle uniformly random picks $y \in \{0, \ldots, p\}$ and returns $c = (g^y, g^{(x*y)} * m)$.
5. Prepare a one-time oracle $\mathcal{O}_{LOR}$.
   When called with $m_0, m_1 \in \mathcal{M}$ the oracle uniformly random picks $y, z \in \{0, \ldots, p\}$ and returns $c = (g^y, g^z * m_h)$.
6. Call the adversary with input $(\mathcal{O}_{Enc}, \mathcal{O}_{LOR})$ and await guess $h'_{LOR} \in \{0, 1\}$.
7. If $h_{LOR} = h'_{LOR}$ then return **ACCEPT** else return **REJECT** .

Every notation and game is defined, thus the Game Hopping Lemma can be introduced.

3 Gamehopping Application

Game Hopping Lemma - shortened version

Let G_0 and G_1 be two games which call the same adversary $\mathcal{A}$ and $\mathcal{D}$ a distinguisher that plays another game called G, where G hides a uniformly random bit $h_G \in \{0,1\}$ from $\mathcal{D}$. Note that $\mathcal{D}(\mathcal{A})$ means the distinguisher $\mathcal{D}$ uses adversary $\mathcal{A}$ to play the game G. The Game Hopping Lemma states that if

$$\mathcal{A} \text{ wins } G_0 \quad \text{iff} \quad \mathcal{D}(\mathcal{A}) \text{ wins } G \text{ given } h_G = 0$$

and

$$\mathcal{A} \text{ wins } G_1 \quad \text{iff} \quad \mathcal{D}(\mathcal{A}) \text{ loses } G \text{ given } h_G = 1$$

are true, then

$$\text{prob}(A \text{ wins } G_0) - \text{prob}(A \text{ wins } G_1) \leq 2 * \text{adv}^G(\mathcal{D}(\mathcal{A}))$$

holds.

As another perspective think of the game G being a bridge between G_0 and G_1 with $\mathcal{D}$ as the support of the bridge. If h_G is 0, the distinguisher can make the game G look like G_0 for the adversary $\mathcal{A}$. Same holds for $h_g = 1$ and G_1. The lemma is used for an analysis of the difference between the left and right side of the bridge.

Assume now that G_1 is a guessing game. Then the Game Hopping Lemma states that the advantage for winning G_0 is at most two times the advantage for playing G. This helps by analyzing and understanding the game G_0 with the help of game G.

In the case of the implementation, G is the game DDH, G_0 is CPA_{real} and G_1 is $\text{CPA}_{\text{random}}$. Therefore by showing that $\text{CPA}_{\text{random}}$ is a null-game, it results in ElGamal encryption being IND-CPA secure if the game DDH has at most negligible advantage. The used distinguisher is defined in the following way:

Distinguisher $\mathcal{D}$

Parameters: a group G, an element $g \in G$ of order p.

Input: g^x, g^y, g^z from DDH game.

1. Uniformly random pick hidden bit $h_{LOR} \in \{0,1\}$.
2. Prepare encryption oracle $\mathcal{O}_{Enc}$.
   When called with $m \in \mathcal{M}$ the oracle uniformly random picks $t \in \{0,\ldots,p\}$ and returns $c = (g^t, (g^x)^t * m)$.
3. Prepare a one-time oracle $\mathcal{O}_{LOR}$.
   When called with $m_0, m_1 \in \mathcal{M}$ the oracle returns $c = (g^y, g^z * m_{h_{LOR}})$.
4. Call the adversary $\mathcal{A}$ with input $(\mathcal{O}_{Enc}, \mathcal{O}_{LOR})$ and await guess $h'_{LOR} \in \{0,1\}$.
5. If $h_{LOR} = h'_{LOR}$ return 0 (REAL), else return 1 (RANDOM).

Note that the distinguisher cannot read the variables x, y and z which are generated by the game DDH. In the encryption oracle only the input g^x is used, while in the test oracle g^y and g^z are used, where z can be either $x * y$ or random.

Apply the Game Hopping Lemma to this scenario. Before doing so one has to show that the precondition holds, meaning that the following two sides have to be true.
Left side:

$$\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{real}} \quad \text{iff} \quad \mathcal{D}(\mathcal{A}) \text{ wins DDH given } h_{DDH} = 0$$

Right side:

$$\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{random}} \quad \text{iff} \quad \mathcal{D}(\mathcal{A}) \text{ loses DDH given } h_{DDH} = 1$$

The left and right side hold respectively by looking at the case for $h_{DDH} = 0$ and $h_{DDH} = 1$. As a small sketch: if the hidden bit is 0 the distinguisher generates the test oracle with $z = x * y$, which makes everything adversary $\mathcal{A}$ can see look like the game $\mathrm{CPA}_{\mathrm{real}}$. If it is 1, z has to be random, hence being same as the game $\mathrm{CPA}_{\mathrm{random}}$.

Thus

$$\mathrm{prob}(\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{real}}) - \mathrm{prob}(\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{random}}) \leq 2 * \mathrm{adv}^{DDH}(\mathcal{D}(\mathcal{A}))$$

holds.

Proving that $\mathrm{prob}(\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{random}}) = 1/2$ for every ppt $\mathcal{A}$ can be done by checking that the ciphertext generated by the test oracle $(g^y, g^z * m_h)$ has the same distribution over $\mathcal{C}$ as (g^y, g^t) for some uniformly random $t \in \{0, \ldots, p\}$. This means that the adversary gains no knowledge from this call, which means he has no information about the hidden bit, hence guessing randomly is the best he can do.
Considering the definition of advantage for $\mathcal{A}$ playing $\mathrm{CPA}_{\mathrm{real}}$:

$$\mathrm{adv}^{\mathrm{CPA}_{\mathrm{real}}}(\mathcal{A}) \quad = \quad \left|\mathrm{prob}(A \text{ wins } \mathrm{CPA}_{\mathrm{real}}) - \frac{1}{2}\right| \quad \leq \quad 2 * \mathrm{adv}^{DDH}(\mathcal{D}(\mathcal{A}))$$

Since $\mathrm{CPA}_{\mathrm{real}}$ is just the combination of the game IND-CPA with the ElGamal encryption scheme, this result is the same as:
The ElGamal encryption scheme is IND-CPA secure under the assumption that the Decisional Diffie-Hellman problem is hard (meaning every probabilistic polynomial time adversary playing DDH game has at most negligible advantage).

Another notation and game that is interesting is the ***LOR-CPA*** game. This is similar to IND-CPA, where the encryption oracle is removed and the test oracle is not a one-time oracle anymore but can be called as many times as the adversary wants to. In the beginning the idea was to implement the proof that ElGamal is IND-LOR secure, which resulted in a problem during the proof. One can show that there exists an equality between LOR-CPA and IND-CPA. This equality can then be used for writing a proof for ElGamal being LOR-CPA secure. To keep the implementation simple IND-CPA was chosen instead of LOR-CPA.

4 Simple introduction to EasyCrypt

EasyCrypt is a computer-aided toolset for cryptographic proofs. One can find more details about Easycrypt at "https://www.easycrypt.info/" [2].

In EasyCrypt a lot of termonologies are needed. The following description is just a shortened explanation of the used language. If one is interested in the correct and complete version, please check the reference manual that can be found in the link given above.

A `type` can be understood as a set or theoretical construct which at first holds no specification. With additional statements one can refine a type, for example as a group. For every type (set) t one can also call the (probability) distribution over t with `t distr` . This way one can formulate an uniform distribution for the key generation.

The term `op` is used for the definition of expressions and operations. If one wants to be able to call the value 0 of type int with the name "x0" one would write `op x0 : int = 0` . This operator may be used from this point onwards in new definitions. The definition of the function $f(x, y) = x$ for integers can be written as `op f (x : int) (y : int) = x` .
To pick an element a of a set s with respect to a distribution the operation `op pick_s : s distr` is needed. For picking it now `a <$ pick_s` has to be called.

A `module` consists of typed variables (`var`) and ***procedures*** (`proc`). Each procedure has an output or return value (where type unit is used for the fixed return of empty element) and can have input parameters. They can be thought of similar to functions in C/C++, where the whole module would be similar to a class. The variables defined inside a module but outside a procedure are called ***global*** and can be used by every module. If the return value of a procedure `main()` shall be stored inside the variable a (where a is of same type as the return value of the procedure), one has to use the `<@` operator, i.e. `a <@ main()` (the @-symbol is only used for procedure assignments).
Example for a procedure that outputs a randomly generated boolean (`{0,1}` is predefined and returns uniformly a boolean):

```
module Coinflip = {
     proc main() : bool = {
          var coin;
          coin <$ {0,1};
          return coin;
     }
}.
```

For a module a `module type` can be defined. It only consists of declarations of procedures. For a module to be of a specific module type it has to have every procedure defined by its type with the same types as in- and output.

The probability expression `Pr[`$M.p(e_1, \ldots, e_n)$ `@ &m : `ϕ`]` for a procedure p of module M that has inputs e_1 to e_n is just a statement about the probability that the formula ϕ holds. The variable &m is a memory variable, which is needed to refer to global variables and modules. Inside the statement ϕ one can use the term `res` , which stands for the output of the procedure $M.p$.

As example consider the probability expression that in the above `module Coinflip` the procedure returns "true" with probability 1/2 (note that `%r` typecasts to a rational number):

```
Pr[Coinflip.main() @ &m :  res = true] = 1%r/2%r.
```

An `axiom` is a formula that is trusted by EasyCrypt, while a `lemma` is a formula that has to be proven to be true. Such a proof starts with `proof` and has to end with `qed` to declare where it is done. The proof engine works with goal lists. A goal consists of a ***context***, which contains all variables, assumptions and information about those, and a ***conclusion***, which is a single formula with the same constraints as the assumption formulas. If every goal of the goal list is solved, the whole lemma is proven. To solve one goal ***tactics*** are used. They are predefined in EasyCrypt and apply logical transformations to the conclusion and context. The transformations can also produce more smaller goals that need to be solved individually. Since the list of those tactics is quite large, checking the reference manual is recommended for further insight.
In the next chapter there are several proofs written in EasyCrypt, where only one is explained thoroughly with the corresponding tactics. To understand the way the other proofs are influencing the goals in detail one should use Proof-General and run through the code in interactive mode.
The definition of a procedure p being ***lossless*** means that p will terminate with probability 1. In the case of an adversary or oracle one can assume that to be always true, because if that would not be true, then the game would not be finished.

Adversaries are modeled by abstract modules. Those are modules where the code is unknown and EasyCrypt quantifies over adversaries. An abstract module that has to be of `module type Adversary_` is defined by `declare module Adv : Adversary_` . Another way to use an abstract module for a single formula is by calling `forall (Adv <: Adversary_)` . Nonetheless the adversaries have to be controlled in some way. Two considerations need to be done: first the procedures that are allowed to call, second the global variables that are allowed to manipulate.

1. The adversary has only access to those procedures that are given to him directly, which are stated in the module type. For a procedure p of module M to be allowed to be called by an adversary's procedure pa, the definition of pa has to have $\{M.p\}$ written after the output type of pa. This then states that if pa is called the adversary can use the procedure $M.p$ for doing whatever he wants.

2. The set of all global variables of a module M is the union of all selfdeclared global variables and all other global variables that can be read or written by a series of procedure calls, beginning with a call of a procedure p which M has access to.
   This means for an adversary, that by giving him access to an oracle, he gets access to the used secret variables (like secret key or hidden bit). Here having access means that he cannot only read, but even write it. Because of this sometimes one wants an adversary to not have access to the stored public key, thus giving him a copy of that instead.
   To make sure that an adversary is not able to access the global variables of M there are two possible ways. During the declaration: `declare module Adv : Adversary_{M}` and in a formula, where ϕ is a statement for the formula: `forall (Adv <: Adversary_{M}, ` ϕ
   Both have the module M written inbetween braces directly after the module type as some kind of assignment. The interpreation means that the global variables of Adv and the global variables of M have to be disjoint. By doing that one can control the global variables given to the adversary.

5 Implementation

To help differentiate between variables, types, modules and module types note that:

1. Types and module types are named with _ as last letter.
2. Variables are starting with a lower case letter.
3. Modules and module types start with a capital letter.

5.1 Premise

All definitions done in chapter 2 need to be implemented.

To specify which game is played, instead of using booleans for every game the following types are used:

```
type DDHhidden_ = [ REAL | RANDOM ].
type LORhidden_ = [ LEFT | REAL ].
type gamedecision_ = [ ACCEPT | REJECT ].
```

The games DDH, CPA_{real} and $\text{CPA}_{\text{random}}$ pick a hidden bit uniformly random, which is implemented with the corresponding distribution-operations:

```
(* For DDHhidden_ *)
op distr_DDHhidden :  DDHhidden_ distr.
axiom distr_DDHhidden_1E : forall (s :  DDHhidden_), mu1 distr_DDHhidden s = (1%r/2%r).
axiom distr_DDHhidden_ll :  is_lossless distr_DDHhidden.
lemma distr_DDHhidden_uni:  is_uniform distr_DDHhidden.
     proof.  by move=> ??; rewrite !distr_DDHhidden_1E. qed.
(* For LORhidden_ *)
op distr_LORhidden :  LORhidden_ distr.
axiom distr_LORhidden_1E : forall (s :  LORhidden_), mu1 distr_LORhidden s = (1%r/2%r).
axiom distr_LORhidden_ll :  is_lossless distr_LORhidden.
lemma distr_LORhidden_uni:  is_uniform distr_LORhidden.
     proof.  by move=> ??; rewrite !distr_LORhidden_1E. qed.
```

Parameters of ElGamal encryption scheme consist of a group G with group element g of order p. For the group the predefined version from EasyCrypt is used, where G is a cyclic group, g is its generator and the set of all integers from 0 to p is a type called t. Meaning the space of the public and secret keys, the plaintext and the ciphertext are the following types:

```
type pk_ = group.
type sk_ = t.
type ptxt_ = group.
type ctxt_ = group * group.
```

Next the templates for the distinguisher (or adversary for playing game DDH) and adversary playing games CPA_{real} and $\text{CPA}_{\text{random}}$need to be defined.

The module type for the distinguisher contains only one procedure, which gets g^x, g^y and g^z as input and has to return a hidden bit for the game DDH.

```
module type DDH_Adversary_ = {
     proc solve(gx gy gz :  group) :  DDHhidden_
}.
```

The module Oracle will contain everything that has to be stored long term, the test oracle (called `enc_lor`) and the encryption oracle (called `enc`). Furtheron there are two procedures, one for having a single call to initialize the oracle and one called public_knowledge, which is used to give the adversary all the information that is publicly known.

```
module type Oracle_ = {
     proc init(pk :  pk_, gy gz :  group) :  unit
     proc public_knowledge() :  pk_
     proc enc_lor(m0 m1 :  ptxt_) :  ctxt_
     proc enc(m :  ptxt_) :  ctxt_
}.
```

The module type for the adversary playing the IND-CPA games contains two procedures, even though in the games he will only be called once. The reason for this is the one-time test oracle. To make sure the adversary gets to call enc_lor only once there are two possible ways to implement this. One is by calling the adversary itself once and handling it the way as done in this implementation, or by allowing to call enc_lor infinitly many times and having a counter variable for control purpose. This second approach leads to another problem, since a procedure always has to return the same type. Therefore either there has to be some workaround or the procedure enc_lor has to return some kind of ciphertext, which could be done by a uniformly random distribution over the ciphertextspace.
The procedures choose (gets no input and has to return two messages) and guess (gets a ciphertext as input and has to return the hidden bit for a IND-CPA game) are called in this order, where both have access to the procedures public_knowledge and enc from the oracle that is used as input. The adversary therefore has no access to the procedures init or enc_lor.

```
module type CPA_Adversary_ (Or :  Oracle_) = {
     proc choose() :  ptxt_ * ptxt_ {Or.public_knowledge Or.enc}
     proc guess(c :  ctxt_) :  LORhidden_ {Or.public_knowledge Or.enc}
}.
```

That contains all the module types. In the following the chapter the modules are explained.

5.2 Games

As the first module the game DDH is implemented, second the Oracle, followed by the games CPA_{real} and $\text{CPA}_{\text{random}}$.

Implementation of game DDH:
Considering how the Game Hopping Lemma is applied, the DDH game needs only to be looked at with fixed hidden bit. Therefore the procedure main will expect a hidden bit already as input and is not chosen randomly. Everything else is just like one would expect from the game definition.

```
module DDH_Game_input (A : DDH_Adversary_) = {
     (* expect hidden bit as input *)
     proc main(ddh_h :  DDHhidden_) :  gamedecision_ = {

          var ddh_h', x, y, z0, z1, gamedecision;
          (* The values which are the exponents for DDH *)
          x <$ FDistr.dt;
          y <$ FDistr.dt;
          z0 = x * y;
          z1 <$ FDistr.dt;

          (* If we are in the REAL-case, we use z0 *)
          if (ddh_h = REAL) {
               ddh_h' <@ A.solve(g ^ x, g ^ y, g ^ z0);}
          else {
               ddh_h' <@ A.solve(g ^ x, g ^ y, g ^ z1);}

          (* If the adversary gave same output, the game ACCEPTS *)
          if (ddh_h = ddh_h') {
               gamedecision = ACCEPT;}
          else {
               gamedecision = REJECT;}

          return gamedecision;
     }
}.
```

Implementation of the oracles $\mathcal{O}_{Enc}$ and $\mathcal{O}_{LOR}$:
As stated at the module type Oracle_, this module has four procedures and stores every long term value it has to know. The hidden bit used by the distinguisher and the games CPA_{real} and $\text{CPA}_{\text{random}}$ are generated inside the procedure init. This is no different than generating it themselves and then storing it inside the module Oracle.

```
module Oracle : Oracle_ = {
    (* global variables *)
    var lor_h : LORhidden_
    var pk : pk_
    var gy : group
    var gz : group

    proc init(pkI : pk_, gyI gzI : group) : unit = {
        pk = pkI;
        gy = gyI;
        gz = gzI;
        lor_h <$ distr_LORhidden;
    }

    proc public_knowledge() : pk_ = {
        return pk;
    }

    proc enc_lor(m0 m1 : ptxt_) : ctxt_ = {
        var c;

        if (lor_h = LEFT) {
            c = (gy, (gz) * m0);}
        else {
            c = (gy, (gz) * m1);}

        return c;
    }

    proc enc(m : ptxt_) : ctxt_ = {
        var t;
        t <$ FDistr.dt;
        return (g ^ t, (pk ^ t) * m);
    }
}.
```

Recommended way of implementation for bigger projects is to split the procedures into two modules. One for storing purpose (would mean having procedure init) and another containing all oracles. In both the procedure public_knowledge could be written. Then it is possible for every module to have Oracle as input instead of calling it strictly. The only downside is in the proof section where more control over the adversary has to be stated.

Implementation of game CPA_{real} :
Calling the adversary first with procedure *choose* and then procedure *guess* is due to the one-time oracle $\mathcal{O}_{LOR}$. Since he is able to call the encryption oracle $\mathcal{O}_{Enc}$ at every time, even before choosing messages m_0 and m_1 (or more specifically during the procedure choose), this is the same as giving him $\mathcal{O}_{LOR}$.
Typically one should expect a loss in security by restricting an adversary, though this can be neglected here.

```
module CPA_Game_real (Adv : CPA_Adversary_) = {
     module A = Adv(Oracle)

     proc main() : gamedecision_ = {
          var lor_h' : LORhidden_;
          var gamedecision : gamedecision_;
          var x, y, z;
          var m0, m1, c;

          (* Sample and define everything we need for the encryption. *)
          x <$ FDistr.dt;
          y <$ FDistr.dt;
          z = x * y;

          (* Initalize the oracle *)
          Oracle.init(g ^ x, g ^ y, g ^ z);

          (* Call the adversary *)
          (m0, m1) <@ A.choose(g ^ x);
          c = Oracle.enc_lor(m0, m1);
          lor_h' <@ A.guess(c);

          (* If the adversary guessed correctly, he will win the game, else he loses. *)
          if (Oracle.lor_h = lor_h'){
          gamedecision = ACCEPT;}
          else {
          gamedecision = REJECT;}

          return gamedecision;
     }
}.
```

Implementation of game $\text{CPA}_{\text{random}}$:
Note that the only thing changed from module CPA_Game_random is the variable z, which is now random. This is exactly as defined by the game notation.

```
module CPA_Game_random (Adv :  CPA_Adversary_) = {
     module A = Adv(Oracle)

     proc main() :  gamedecision_ = {
          var lor_h' :  LORhidden_;
          var gamedecision :  gamedecision_;
          var x, y, z;
          var m0, m1, c;

          x <$ FDistr.dt;
          y <$ FDistr.dt;
          z <$ FDistr.dt; (* this is changed *)

          Oracle.init(g ^ x, g ^ y, g ^ z);

          (m0, m1) <@ A.choose(g ^ x);
          c = Oracle.enc_lor(m0, m1);
          lor_h' <@ A.guess(c);

          if (Oracle.lor_h = lor_h'){
          gamedecision = ACCEPT;}
          else {
          gamedecision = REJECT;}

          return gamedecision;
     }
}.
```

Implementation of the distinguisher $\mathcal{D}$:
The module Distinguisher gets (g^x, g^y, g^z) as input from the game DDH. From this information he generates everything needed to call an adversary for the game IND-CPA. The output then is used for stating if z is real or random.

```
module Distinguisher (Adv : CPA_Adversary_) : DDH_Adversary_ = {
     module A = Adv(Oracle)

     (* Input is from the DDH-Game. *)
     proc solve(gx : group, gy : group, gz : group) : DDHhidden_ = {
          var lor_h' : LORhidden_;
          var ddh_h' : DDHhidden_;
          var m0, m1, c;

          (* Initalize the oracle such that we can let the adversary play the CPA-Game.
          *)
          Oracle.init(gx,gy,gz);

          (* Call the adversary as distinguisher *)
          (m0, m1) <@ A.choose(gx);
          c = Oracle.enc_lor(m0, m1);
          lor_h' <@ A.guess(c);

          (* Use the adversaries output *)
          if (Oracle.lor_h = lor_h'){
          ddh_h' = REAL;}
          else {
          ddh_h' = RANDOM;}

          (* Return guess for the game DDH. *)
          return ddh_h';
     }
}.
```

5.3 Proofs

Finding a proof just by using tactics from EasyCrypt can be quite bothersome. Especially if one is new to this language. To make it simpler, one can use more modules which are closer to each other. For example instead of the module DDH_Game_input calling the distinguisher another two modules called DDH0 and DDH1 were created. This way the problems that arised were easier to find and fix, therefore recommending that approach.

The adversary is not defined yet, just its module type. To be able to call him and define that he gets no access to the module Oracle, the following is needed:

```
declare module Adv :  CPA_Adversary_{Oracle}.
```

Because Adv only has access to the module Oracle, the only global variables he can read/write are either defined by Oracle or used in one of its procedures. Every procedure defined in Oracle computes with its own known variables and does not call any other procedure. Therefore Adv has no global variables, just the information that was given to him beforehand.
To check that this really is the case, one can remove the braces with its context and check the proof that is given below. Then there would be an error message stating that variables are writeable by the adversary and restrictions are required.

For applying Game Hopping Lemma the left side

$$\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{real}} \quad \text{iff} \quad \mathcal{D}(\mathcal{A}) \text{ wins DDH given } h_{DDH} = 0$$

is needed to be shown to be true. This is done the following way (explanation for this proof is below):

```
lemma left_side &m:
     Pr[DDH_Game_input(Distinguisher(Adv)).main(REAL) @ &m :  (res = ACCEPT)] =
     Pr[CPA_Game_real(Adv).main() @ &m :  (res = ACCEPT)].

     proof.
          byequiv => //.
          proc.
          rcondt{1} 5.
          by auto.
          inline*.
          swap{1} 4 19.
          rnd{1}.
          wp; simplify.
          call(_:  ={pk, gy, gz, lor_h}(Oracle, Oracle)); try sim.
          auto.
          call(_:  ={pk, gy, gz, lor_h}(Oracle, Oracle)); try sim.
          auto.
          simplify.
          progress; apply dt_ll.
     qed.
```

In the next section the tactics used in the proof above are explained and shown in the interactive mode of EasyCrypt via ProofGeneral.

5.3.1 Goal-List and Tactics

The goal from the lemma itself looks like:

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Pr[DDH_Game_input(Distinguisher(Adv)).main(REAL) @ &m : res = ACCEPT] =
Pr[CPA_Game_real(Adv).main() @ &m : res = ACCEPT]
```

With the tactic `byequiv` the probability expression is transformed into an equal equality expression. The addition `=> //` means that for every goal created by this the tactic `trivial` is applied, which really just checks if the goal can be solved trivial. In this case `byequiv` would create 3 goals. Two of them are solved by the tactic `trivial` , therefore just one is created.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

    DDH_Game_input(Distinguisher(Adv)).main ~ CPA_Game_real(Adv).main

post = ={res}
```

By applying `proc` the equality will now be presented by the corresponding procedures. The equality has to be shown (stated in the post condition) by showing that both sides have the same output. By writing {1} for the left side and {2} for the right side one can distinguish if a variable or tactic is called for one side.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                    (1--)  x <$ dt
y <$ dt                                    (2--)  y <$ dt
z0 <- x * y                                (3--)  z <- x * y
z1 <$ dt                                   (4--)  Oracle.init(g ^ x, g ^ y, g ^ z)
if (ddh_h = REAL) {                        (5--)  (m0, m1) <@
                                           (  -)    CPA_Game_real(Adv).A.choose(g ^ x)
  ddh_h' <@                                (5.1)
    Distinguisher(Adv).solve(g ^ x,        (   )
    g ^ y, g ^ z0)                         (   )
) else {                                   (5--)
  ddh_h' <@                                (5?1)
    Distinguisher(Adv).solve(g ^ x,        (   )
    g ^ y, g ^ z1)                         (   )
}                                          (5--)
if (ddh_h = ddh_h') {                      (6--)  c <@ Oracle.enc_lor(m0, m1)
  gamedecision <- ACCEPT                   (6.1)
) else {                                   (6--)
  gamedecision <- REJECT                   (6?1)
}                                          (6--)
                                           (7--)  lor_h' <@ CPA_Game_real(Adv).A.guess(c)
                                           (8--)  if (Oracle.lor_h = lor_h') {
                                           (8.1)    gamedecision <- ACCEPT
                                           (8--)  else {
                                           (8?1)    gamedecision <- REJECT
                                           (8--)

post = ={gamedecision}
```

Using `rcondt` makes the current goal with an if statement into two goals. One where the if statement (here in line 5 on the left side) has to be shown to be true and the other goal is the one from befure where the statement is assumed to be true. The newly generated one looks like this:

```
Current goal (remaining: 2)

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
forall &m0,
  hoare[ x <$ $dt; ...; z1 <$ $dt :
           (glob Adv){m0} = (glob Adv){m} /\
           ddh_h = REAL /\ (glob Adv) = (glob Adv){m} ==> ddh_h = REAL]
```

If a tactic starts with `by` this means that the current goal has to be closed with the following tactic, else an error would occur. The tactic `auto` tries a few tactics itself, for example `trivial` and `simplify` (is explained in the step it is used). Since this also closes the current goal, the only goal one can see at this ponit is the other goal that was generated by `rcondt` .

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                    (1--)  x <$ dt
y <$ dt                                    (2--)  y <$ dt
z0 <- x * y                                (3--)  z <- x * y
z1 <$ dt                                   (4--)  Oracle.init(g ^ x, g ^ y, g ^ z)
ddh_h' <@                                  (5--)  (m0, m1) <@
  Distinguisher(Adv).solve(g ^ x,          (  -)    CPA_Game_real(Adv).A.choose(g ^ x)
  g ^ y, g ^ z0)                           (  -)
if (ddh_h = ddh_h') {                      (6--)  c <@ Oracle.enc_lor(m0, m1)
  gamedecision <- ACCEPT                   (6.1)
) else {                                   (6--)
  gamedecision <- REJECT                   (6?1)
}                                          (6--)
                                           (7--)  lor_h' <@ CPA_Game_real(Adv).A.guess(c)
                                           (8--)  if (Oracle.lor_h = lor_h') {
                                           (8.1)    gamedecision <- ACCEPT
                                           (8--)  else {
                                           (8?1)    gamedecision <- REJECT
                                           (8--)

post = ={gamedecision}
```

Applying `inline*` checks both sides for a procedure call. If any is found and the procedure is defined beforehand it is replaced by the procedure itself. This makes the whole goal a lot longer.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                          ( 1--)  x <$ dt
y <$ dt                                          ( 2--)  y <$ dt
z0 <- x * y                                      ( 3--)  z <- x * y
z1 <$ dt                                         ( 4--)  pkI <- g ^ x
gx <- g ^ x                                      ( 5--)  gyI <- g ^ y
gy <- g ^ y                                      ( 6--)  gzI <- g ^ z
gz <- g ^ z0                                     ( 7--)  Oracle.pk <- pkI
pkI <- gx                                        ( 8--)  Oracle.gy <- gyI
gyI <- gy                                        ( 9--)  Oracle.gz <- gzI
gzI <- gz                                        (10--)  Oracle.lor_h <$ distr_LORhidden
Oracle.pk <- pkI                                 (11--)  (m0, m1) <@
                                                 (   -)    CPA_Game_real(Adv).A.choose(g ^ x)
Oracle.gy <- gyI                                 (12--)  m00 <- m0
Oracle.gz <- gzI                                 (13--)  m10 <- m1
Oracle.lor_h <$ distr_LORhidden                  (14--)  if (Oracle.lor_h = LEFT) {
                                                 (14.1)    c0 <- (Oracle.gy, Oracle.gz * m00)
                                                 (14--)  else {
                                                 (14?1)    c0 <- (Oracle.gy, Oracle.gz * m10)
                                                 (14--)
(m0, m1) <@                                      (15--)  c <- c0
  Distinguisher(Adv).A.choose(gx)                (   -)
m00 <- m0                                        (16--)  lor_h' <@ CPA_Game_real(Adv).A.guess(c)
m10 <- m1                                        (17--)  if (Oracle.lor_h = lor_h') {
                                                 (17.1)    gamedecision <- ACCEPT
                                                 (17--)  else {
                                                 (17?1)    gamedecision <- REJECT
                                                 (17--)
if (Oracle.lor_h = LEFT) {                       (18--)
  c0 <- (Oracle.gy, Oracle.gz * m00)             (18.1)
) else {                                         (18--)
  c0 <- (Oracle.gy, Oracle.gz * m10)             (18?1)
}                                                (18--)
c <- c0                                          (19--)
lor_h' <@ Distinguisher(Adv).A.guess(c)          (20--)
if (Oracle.lor_h = lor_h') {                     (21--)
  ddh_h'0 <- REAL                                (21.1)
) else {                                         (21--)
  ddh_h'0 <- RANDOM                              (21?1)
}                                                (21--)
ddh_h' <- ddh_h'0                                (22--)
if (ddh_h = ddh_h') {                            (23--)
  gamedecision <- ACCEPT                         (23.1)
) else {                                         (23--)
  gamedecision <- REJECT                         (23?1)
}                                                (23--)

post = ={gamedecision}
```

The left side has in line 4 the variable $z1$ which is not used on the right side. With `swap` the line moves to the end (which is done manually by moving it 19 lines downwards). This is done so both sides are similar before that part.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                          ( 1--)  x <$ dt
y <$ dt                                          ( 2--)  y <$ dt
z0 <- x * y                                      ( 3--)  z <- x * y
gx <- g ^ x                                      ( 4--)  pkI <- g ^ x
gy <- g ^ y                                      ( 5--)  gyI <- g ^ y
gz <- g ^ z0                                     ( 6--)  gzI <- g ^ z
pkI <- gx                                        ( 7--)  Oracle.pk <- pkI
gyI <- gy                                        ( 8--)  Oracle.gy <- gyI
gzI <- gz                                        ( 9--)  Oracle.gz <- gzI
Oracle.pk <- pkI                                 (10--)  Oracle.lor_h <$ distr_LORhidden
Oracle.gy <- gyI                                 (11--)  (m0, m1) <@
                                                 (   -)    CPA_Game_real(Adv).A.choose(g ^ x)
Oracle.gz <- gzI                                 (12--)  m00 <- m0
Oracle.lor_h <$ distr_LORhidden                  (13--)  m10 <- m1
(m0, m1) <@                                      (14--)  if (Oracle.lor_h = LEFT) {
  Distinguisher(Adv).A.choose(gx)                (   -)
                                                 (14.1)    c0 <- (Oracle.gy, Oracle.gz * m00)
                                                 (14--)  else {
                                                 (14?1)    c0 <- (Oracle.gy, Oracle.gz * m10)
                                                 (14--)
m00 <- m0                                        (15--)  c <- c0
m10 <- m1                                        (16--)  lor_h' <@ CPA_Game_real(Adv).A.guess(c)
if (Oracle.lor_h = LEFT) {                       (17--)  if (Oracle.lor_h = lor_h') {
  c0 <- (Oracle.gy, Oracle.gz * m00)             (17.1)    gamedecision <- ACCEPT
) else {                                         (17--)  else {
  c0 <- (Oracle.gy, Oracle.gz * m10)             (17?1)    gamedecision <- REJECT
}                                                (17--)
c <- c0                                          (18--)
lor_h' <@ Distinguisher(Adv).A.guess(c)          (19--)
if (Oracle.lor_h = lor_h') {                     (20--)
  ddh_h'0 <- REAL                                (20.1)
) else {                                         (20--)
  ddh_h'0 <- RANDOM                              (20?1)
}                                                (20--)
ddh_h' <- ddh_h'0                                (21--)
if (ddh_h = ddh_h') {                            (22--)
  gamedecision <- ACCEPT                         (22.1)
) else {                                         (22--)
  gamedecision <- REJECT                         (22?1)
}                                                (22--)
z1 <$ dt                                         (23--)

post = ={gamedecision}
```

The tactic `rnd` moves the random assignment for $z1$ into the post condition, removing it from the procedure step on the left side.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                          ( 1--)  x <$ dt
y <$ dt                                          ( 2--)  y <$ dt
z0 <- x * y                                      ( 3--)  z <- x * y
gx <- g ^ x                                      ( 4--)  pkI <- g ^ x
gy <- g ^ y                                      ( 5--)  gyI <- g ^ y
gz <- g ^ z0                                     ( 6--)  gzI <- g ^ z
pkI <- gx                                        ( 7--)  Oracle.pk <- pkI
gyI <- gy                                        ( 8--)  Oracle.gy <- gyI
gzI <- gz                                        ( 9--)  Oracle.gz <- gzI
Oracle.pk <- pkI                                 (10--)  Oracle.lor_h <$ distr_LORhidden
Oracle.gy <- gyI                                 (11--)  (m0, m1) <@
                                                 (   -)    CPA_Game_real(Adv).A.choose(g ^ x)
Oracle.gz <- gzI                                 (12--)  m00 <- m0
Oracle.lor_h <$ distr_LORhidden                  (13--)  m10 <- m1
(m0, m1) <@                                      (14--)  if (Oracle.lor_h = LEFT) {
  Distinguisher(Adv).A.choose(gx)                (   -)
                                                 (14.1)    c0 <- (Oracle.gy, Oracle.gz * m00)
                                                 (14--)  else {
                                                 (14?1)    c0 <- (Oracle.gy, Oracle.gz * m10)
                                                 (14--)
m00 <- m0                                        (15--)  c <- c0
m10 <- m1                                        (16--)  lor_h' <@ CPA_Game_real(Adv).A.guess(c)
if (Oracle.lor_h = LEFT) {                       (17--)  if (Oracle.lor_h = lor_h') {
  c0 <- (Oracle.gy, Oracle.gz * m00)             (17.1)    gamedecision <- ACCEPT
) else {                                         (17--)  else {
  c0 <- (Oracle.gy, Oracle.gz * m10)             (17?1)    gamedecision <- REJECT
}                                                (17--)
c <- c0                                          (18--)
lor_h' <@ Distinguisher(Adv).A.guess(c)          (19--)
if (Oracle.lor_h = lor_h') {                     (20--)
  ddh_h'0 <- REAL                                (20.1)
) else {                                         (20--)
  ddh_h'0 <- RANDOM                              (20?1)
}                                                (20--)
ddh_h' <- ddh_h'0                                (21--)
if (ddh_h = ddh_h') {                            (22--)
  gamedecision <- ACCEPT                         (22.1)
) else {                                         (22--)
  gamedecision <- REJECT                         (22?1)
}                                                (22--)

post = is_lossless dt && forall (z1_0 : t), z1_0 \in dt => ={gamedecision}
```

If one wants to apply a tactic t_2 to every goal produced by a tactic t_1 write t_1; t_2 . The tactic `wp` moves all assignments that are not random or a procedure callinto the post condition and `simplify` removes a lot of trivial statements by shortening it (for example instead of writing "ACCEPT = ACCEPT" shortens to "true"), thus making it easier to read, and fusing if-statements into a simpler form.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                        ( 1--)  x <$ dt
y <$ dt                                        ( 2--)  y <$ dt
z0 <- x * y                                    ( 3--)  z <- x * y
gx <- g ^ x                                    ( 4--)  pkI <- g ^ x
gy <- g ^ y                                    ( 5--)  gyI <- g ^ y
gz <- g ^ z0                                   ( 6--)  gzI <- g ^ z
pkI <- gx                                      ( 7--)  Oracle.pk <- pkI
gyI <- gy                                      ( 8--)  Oracle.gy <- gyI
gzI <- gz                                      ( 9--)  Oracle.gz <- gzI
Oracle.pk <- pkI                               (10--)  Oracle.lor_h <$ distr_LORhidden
Oracle.gy <- gyI                               (11--)  (m0, m1) <@
                                               (   -)    CPA_Game_real(Adv).A.choose(g ^ x)
Oracle.gz <- gzI                               (12--)  m00 <- m0
Oracle.lor_h <$ distr_LORhidden                (13--)  m10 <- m1
(m0, m1) <@                                    (14--)  if (Oracle.lor_h = LEFT) {
  Distinguisher(Adv).A.choose(gx)              (   -)
                                               (14.1)    c0 <- (Oracle.gy, Oracle.gz * m00)
                                               (14--)  else {
                                               (14?1)    c0 <- (Oracle.gy, Oracle.gz * m10)
                                               (14--)
m00 <- m0                                      (15--)  c <- c0
m10 <- m1                                      (16--)  lor_h' <@ CPA_Game_real(Adv).A.guess(c)
if (Oracle.lor_h = LEFT) {                     (17--)
  c0 <- (Oracle.gy, Oracle.gz * m00)           (17.1)
) else {                                       (17--)
  c0 <- (Oracle.gy, Oracle.gz * m10)           (17?1)
}                                              (17--)
c <- c0                                        (18--)
lor_h' <@ Distinguisher(Adv).A.guess(c)        (19--)
```

The post-condition is on the next page.

```
post =
  if Oracle.lor_h{2} = lor_h'{2} then
    if Oracle.lor_h{1} = lor_h'{1} then
      if ddh_h{1} = REAL then is_lossless dt
      else is_lossless dt && forall (z1_0 : t), z1_0 \notin dt
    else
      if ddh_h{1} = RANDOM then is_lossless dt
      else is_lossless dt && forall (z1_0 : t), z1_0 \notin dt
  else
    if Oracle.lor_h{1} = lor_h'{1} then
      if ddh_h{1} = REAL then
        is_lossless dt && forall (z1_0 : t), z1_0 \notin dt
      else is_lossless dt
    else
      if ddh_h{1} = RANDOM then
        is_lossless dt && forall (z1_0 : t), z1_0 \notin dt
      else is_lossless dt
```

From this point onwards the post-condition will not be shown since it is too large without much information one can gain from it.

The tactic `call` produces looks at the current procedure assignment (this can only be done if both sides end with such an assignment) and produces new goals which are stating assumptions about the equality. Here this means that 2 goals would be produced, one for every procedure the adversary is allowed to call from the oracles. Inside the parantheses of `call` every variable that should be assumed to be the same has to be written. In this case the variables pk, gy, gz and lor_h are assumed to be equivalent in both sides. This changes the current post condition such that one has to show that those variables are truly equivalent.
By using `try` before a tactic, it is only applied it if it does not fail. Together with `sim` , which reduces an equality statement into a simpler form and can even solve it if they are similar enough, the goals produced by `call` are solved instantly.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                    ( 1--)  x <$ dt
y <$ dt                                    ( 2--)  y <$ dt
z0 <- x * y                                ( 3--)  z <- x * y
gx <- g ^ x                                ( 4--)  pkI <- g ^ x
gy <- g ^ y                                ( 5--)  gyI <- g ^ y
gz <- g ^ z0                               ( 6--)  gzI <- g ^ z
pkI <- gx                                  ( 7--)  Oracle.pk <- pkI
gyI <- gy                                  ( 8--)  Oracle.gy <- gyI
gzI <- gz                                  ( 9--)  Oracle.gz <- gzI
Oracle.pk <- pkI                           (10--)  Oracle.lor_h <$ distr_LORhidden
Oracle.gy <- gyI                           (11--)  (m0, m1) <@
                                           (   -)    CPA_Game_real(Adv).A.choose(g ^ x)
Oracle.gz <- gzI                           (12--)  m00 <- m0
Oracle.lor_h <$ distr_LORhidden            (13--)  m10 <- m1
(m0, m1) <@                                (14--)  if (Oracle.lor_h = LEFT) {
  Distinguisher(Adv).A.choose(gx)          (   -)
                                           (14.1)    c0 <- (Oracle.gy, Oracle.gz * m00)
                                           (14--)  else {
                                           (14?1)    c0 <- (Oracle.gy, Oracle.gz * m10)
                                           (14--)
m00 <- m0                                  (15--)  c <- c0
m10 <- m1                                  (16--)
if (Oracle.lor_h = LEFT) {                 (17--)
  c0 <- (Oracle.gy, Oracle.gz * m00)       (17.1)
) else {                                   (17--)
  c0 <- (Oracle.gy, Oracle.gz * m10)       (17?1)
}                                          (17--)
c <- c0                                    (18--)
```

After applying `auto` again:

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                    ( 1)  x <$ dt
y <$ dt                                    ( 2)  y <$ dt
z0 <- x * y                                ( 3)  z <- x * y
gx <- g ^ x                                ( 4)  pkI <- g ^ x
gy <- g ^ y                                ( 5)  gyI <- g ^ y
gz <- g ^ z0                               ( 6)  gzI <- g ^ z
pkI <- gx                                  ( 7)  Oracle.pk <- pkI
gyI <- gy                                  ( 8)  Oracle.gy <- gyI
gzI <- gz                                  ( 9)  Oracle.gz <- gzI
Oracle.pk <- pkI                           (10)  Oracle.lor_h <$ distr_LORhidden
Oracle.gy <- gyI                           (11)  (m0, m1) <@
                                           (  )    CPA_Game_real(Adv).A.choose(g ^ x)
Oracle.gz <- gzI                           (12)
Oracle.lor_h <$ distr_LORhidden            (13)
(m0, m1) <@                                (14)
  Distinguisher(Adv).A.choose(gx)          (  )
```

Before `call` was used for the procedure *guess* of the adversary, now use it for procedure *choose*.

```
Current goal

Type variables: <none>

&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
&1 (left ) : DDH_Game_input(Distinguisher(Adv)).main
&2 (right) : CPA_Game_real(Adv).main

pre =
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m}

x <$ dt                                    ( 1)  x <$ dt
y <$ dt                                    ( 2)  y <$ dt
z0 <- x * y                                ( 3)  z <- x * y
gx <- g ^ x                                ( 4)  pkI <- g ^ x
gy <- g ^ y                                ( 5)  gyI <- g ^ y
gz <- g ^ z0                               ( 6)  gzI <- g ^ z
pkI <- gx                                  ( 7)  Oracle.pk <- pkI
gyI <- gy                                  ( 8)  Oracle.gy <- gyI
gzI <- gz                                  ( 9)  Oracle.gz <- gzI
Oracle.pk <- pkI                           (10)  Oracle.lor_h <$ distr_LORhidden
Oracle.gy <- gyI                           (11)
Oracle.gz <- gzI                           (12)
Oracle.lor_h <$ distr_LORhidden            (13)
```

After applying `auto` again, every step of the procedures on both sides are moved into the post condition. Therefore the only goal that is left is a single statement that has to be shown. To give a small overview how that looks like the next code (split onto two pages) already has the followup tactic `simplify` applied to it.

```
Current goal
Type variables: <none>
&m: memory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
forall &1 &2,
  (glob Adv){2} = (glob Adv){m} /\
  ddh_h{1} = REAL /\ (glob Adv){1} = (glob Adv){m} =>
  forall (xL : t),
    xL \in dt =>
    (xL \in dt) &&
    forall (yL : t),
      yL \in dt =>
      (yL \in dt) &&
      forall (lor_hL : LORhidden_),
        lor_hL \in distr_LORhidden =>
        (lor_hL \in distr_LORhidden) &&
        ={glob Adv} &&
        forall (result_L result_R : ptxt_ * ptxt_) (Adv_L
          Adv_R : (glob Adv)),
          result_L = result_R /\ Adv_L = Adv_R =>
          if lor_hL = LEFT then
            if lor_hL = LEFT then
              (g ^ (xL * yL) * result_L.`1 = g ^ (xL * yL) * result_R.`1 /\
               Adv_L = Adv_R) &&
              forall (result_L0 result_R0 : LORhidden_) (Adv_L0
                Adv_R0 : (glob Adv)),
                result_L0 = result_R0 /\ Adv_L0 = Adv_R0 =>
                if lor_hL = result_R0 then
                  if lor_hL = result_L0 then
                    if ddh_h{1} = REAL then is_lossless dt
                    else is_lossless dt && forall (z1 : t), z1 \notin dt
                  else
                    if ddh_h{1} = RANDOM then is_lossless dt
                    else is_lossless dt && forall (z1 : t), z1 \notin dt
                else
                  if lor_hL = result_L0 then
                    if ddh_h{1} = REAL then
                      is_lossless dt && forall (z1 : t), z1 \notin dt
                    else is_lossless dt
                  else
                    if ddh_h{1} = RANDOM then
                      is_lossless dt && forall (z1 : t), z1 \notin dt
                    else is_lossless dt
```

```
    else
      (g ^ (xL * yL) * result_L.`2 = g ^ (xL * yL) * result_R.`1 /\
       Adv_L = Adv_R) &&
      forall (result_L0 result_R0 : LORhidden_) (Adv_L0
        Adv_R0 : (glob Adv)),
        result_L0 = result_R0 /\ Adv_L0 = Adv_R0 =>
        if lor_hL = result_R0 then
          if lor_hL = result_L0 then
            if ddh_h{1} = REAL then is_lossless dt
            else is_lossless dt && forall (z1 : t), z1 \notin dt
          else
            if ddh_h{1} = RANDOM then is_lossless dt
            else is_lossless dt && forall (z1 : t), z1 \notin dt
        else
          if lor_hL = result_L0 then
            if ddh_h{1} = REAL then
              is_lossless dt && forall (z1 : t), z1 \notin dt
            else is_lossless dt
          else
            if ddh_h{1} = RANDOM then
              is_lossless dt && forall (z1 : t), z1 \notin dt
            else is_lossless dt
  else
    if lor_hL = LEFT then
      (g ^ (xL * yL) * result_L.`1 = g ^ (xL * yL) * result_R.`2 /\
       Adv_L = Adv_R) &&
      forall (result_L0 result_R0 : LORhidden_) (Adv_L0
        Adv_R0 : (glob Adv)),
        result_L0 = result_R0 /\ Adv_L0 = Adv_R0 =>
        if lor_hL = result_R0 then
          if lor_hL = result_L0 then
            if ddh_h{1} = REAL then is_lossless dt
            else is_lossless dt && forall (z1 : t), z1 \notin dt
          else
            if ddh_h{1} = RANDOM then is_lossless dt
            else is_lossless dt && forall (z1 : t), z1 \notin dt
        else
          if lor_hL = result_L0 then
            if ddh_h{1} = REAL then
              is_lossless dt && forall (z1 : t), z1 \notin dt
            else is_lossless dt
          else
            if ddh_h{1} = RANDOM then
              is_lossless dt && forall (z1 : t), z1 \notin dt
            else is_lossless dt
    else
      (g ^ (xL * yL) * result_L.`2 = g ^ (xL * yL) * result_R.`2 /\
       Adv_L = Adv_R) &&
      forall (result_L0 result_R0 : LORhidden_) (Adv_L0
        Adv_R0 : (glob Adv)),
        result_L0 = result_R0 /\ Adv_L0 = Adv_R0 =>
        if lor_hL = result_R0 then
          if lor_hL = result_L0 then
            if ddh_h{1} = REAL then is_lossless dt
            else is_lossless dt && forall (z1 : t), z1 \notin dt
          else
            if ddh_h{1} = RANDOM then is_lossless dt
            else is_lossless dt && forall (z1 : t), z1 \notin dt
        else
          if lor_hL = result_L0 then
            if ddh_h{1} = REAL then
              is_lossless dt && forall (z1 : t), z1 \notin dt
            else is_lossless dt
          else
            if ddh_h{1} = RANDOM then
              is_lossless dt && forall (z1 : t), z1 \notin dt
            else is_lossless dt
```

The tactic `progress` changes the whole goal, which has to be only one statement as in the last step, into several smaller ones that are easier to read and solve. In this case 4 goals would be produced which all are solved if one can show that the distribution call `dt` terminates with probability 1 (which one is allowed to assume, otherwise the key-generation would not even work). One of those goals looks like this:

```
Current goal (remaining:  4)
Type variables:  <none>
&m:  memory
&1:  memory <DDH_Game_input(Distinguisher(Adv)).main>
&2:  memory <CPA_Game_real(Adv).main>
xL: t
H : xL \in dt
H0:  xL \in dt
yL: t
H1:  yL \in dt
H2:  yL \in dt
result_R: ptxt_ * ptxt_
Adv_R: (glob Adv)
H3:  LEFT \in distr_LORhidden
H4:  LEFT \in distr_LORhidden
Adv_R0:  (glob Adv)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
is_lossless dt
```

This and the other goals are solved by the tactic `apply` , which needs an `axiom` or `lemma` as input. It gets the axiom `dt_ll` as input, which states that `dt` is lossless (exactly what is needed here).

As the last step this produces an empty goal-list, thus the whole lemma can be saved with the command `qed` .

The following proofs are only shown without explanation for tactics or the goal-list.

5.3.2 Going on with Proofs

For the right side

$$\mathcal{A} \text{ wins } \mathrm{CPA}_{\mathrm{random}} \quad \text{iff} \quad \mathcal{D}(\mathcal{A}) \text{ loses DDH given } h_{DDH} = 1$$

the implementation looks like:

```
lemma right_side &m:
      Pr[DDH_Game_input(Distinguisher(Adv)).main(RANDOM) @ &m :  (res = ACCEPT)] =
      Pr[CPA_Game_random(Adv).main() @ &m :  (res = ACCEPT)].

      proof.
            byequiv => //.
            proc.
            rcondf{1} 5.
            by auto.
            inline*.
            wp.
            simplify; auto; simplify; auto.
            call(_:  ={pk, gy, gz, lor _h}(Oracle, Oracle)); try sim.
            auto.
            call(_:  ={pk, gy, gz, lor_h}(Oracle, Oracle)); try sim.
            auto.
      qed.
```

Since both sides are proven, the Game Hopping Lemma can be applied. The only thing to do now is to check that $\mathrm{CPA}_{\mathrm{random}}$ is a guessing game.

This is done via one intermediate step, where another module called `Oracle_nom` is used, which is nearly the same as the module Oracle. The only difference being inside the procedure `enc_lor`, where no message is used for encryption instead of the message m_h. This module looks like this:

```
module Oracle_nom : Oracle_ = {
    var lor_h : LORhidden_
    var pk : pk_
    var gy : group
    var gz : group

    proc init(pkI : pk_, gyI gzI : group) : unit = {
        pk = pkI;
        gy = gyI;
        gz = gzI;
        lor_h <$ distr_LORhidden;
    }
    proc public_knowledge() : pk_ = {
        return pk;
    }
    proc enc_lor(m0 m1 : ptxt_) : ctxt_ = {
        return (gy, gz); (* This line here is different. *)
    }
    proc enc(m : ptxt_) : ctxt_ = {
        var t;
        t <$ FDistr.dt;
        return (g ^ t, (pk ^ t) * m);
    }
}.
```

Due to the way CPA_Game_random is implemented, the call to Oracle is fixed. Thus another version, called CPA_Game_random_nom, is needed. It is the same as CPA_Game_random while calling Oracle_nom instead of Oracle:

```
module CPA_Game_random_nom (Adv : CPA_Adversary_) = {
     module A = Adv(Oracle_nom)

     proc main() : gamedecision_ = {
          var lor_h' : LORhidden_;
          var gamedecision : gamedecision_;
          var x, y, z;
          var m0, m1, c;

          x <$ FDistr.dt;
          y <$ FDistr.dt;
          z <$ FDistr.dt;

          Oracle_nom.init(g ^ x, g ^ y, g ^ z);

          (m0, m1) <@ A.choose(g ^ x);
          c = Oracle_nom.enc_lor(m0, m1);
          lor_h' <@ A.guess(c);

          if (Oracle_nom.lor_h = lor_h'){
          gamedecision = ACCEPT;}
          else {
          gamedecision = REJECT;}

          return gamedecision;
     }
}.
```

Contemplating at this point, it would already have been better for a small project like this to have Oracle as an input instead of a fixed module that is inserted.

First show that both modules are the same for an adversary that has disjoint global variables from both, Oracle and Oracle_nom:

```
declare module Adv :  CPA_Adversary_{Oracle, Oracle_nom}.
     Pr[CPA_Game_random(Adv).main() @ &m :  (res = ACCEPT)] =
     Pr[CPA_Game_random_nom(Adv).main() @ &m :  (res = ACCEPT)].

     proof.
          byequiv; auto.
          proc; inline*.
          auto.
          call(_:  ={pk, gy, lor_h}(Oracle, Oracle_nom)); try sim.
          auto.
          swap{1} 3 2.
          swap{1} [5..6] 2.
          swap{1} 11 -4.
          swap{1} 11 -4.
          swap{2} 3 2.
          swap{2} [5..6] 2.
          swap{2} 11 -4.
          swap{2} 11 -4.
          wp.
          rnd (fun z, z + log (if (Oracle.lor_h = LEFT) then m0 else m1){1})
               (fun z, z - log (if (Oracle.lor_h = LEFT) then m0 else m1){1}).
          auto.
          call(_:  ={pk, gy, lor_h}(Oracle, Oracle_nom)); try sim.
          auto.
          simplify.
          progress.
          algebra.
          smt.
          smt.
          algebra.
          algebra.
          algebra.
     qed.
```

This makes the proof a whole lot easier, since the probability of 1/2 can be shown for this new game easier.
Before doing that, a few axioms have to be defined. Those axioms all just state, that every procedure is lossless (meaning every procedure terminates with probability 1).

For Oracle_nom:

```
axiom or_nom_pk_ll &m:
     islossless Oracle_nom.public_knowledge.

axiom or_nom_enc_ll &m:
     islossless Oracle_nom.enc.
```

For adversary:

```
axiom adv_ll1 &m:
     forall (Or <:  Oracle_{Adv}),
          islossless Or.public_knowledge => islossless Or.enc =>
          islossless Adv(Or).guess.

axiom adv_ll2 &m:
     forall (Or <:  Oracle_{Adv}),
          islossless Or.public_knowledge => islossless Or.enc =>
          islossless Adv(Or).choose.
```

CPA_{real} is as good as a guessing game:

```
lemma nom_half &m:
     Pr[CPA_Game_random_nom(Adv).main() @ &m :  (res = ACCEPT)] = 1%r/2%r.

     proof.
          byphoare; auto.
          proc; inline*.
          auto.
          swap 10 5.
          rnd.
          call(_:  true).
          apply adv_ll1.
          apply or_nom_pk_ll.
          apply or_nom_enc_ll.
          auto.
          call(_:  true).
          apply adv_ll2.
          by proc; auto.
          by proc; auto => ??; apply dt_ll.
          auto.
          progress.
          apply dt_ll.
          apply distr_LORhidden_1E.
     qed.
```

Final result using both lemmas:

```
lemma zero_advantage &m:
     `| Pr[CPA_Game_random(Adv).main() @ &m :  (res = ACCEPT)] - 1%r/2%r | = 0%r.

     proof.
          rewrite (same_nom &m).
          rewrite (nom_half &m).
          auto.
     qed.
```

By using EasyCrypt it is shown that the ElGamal encryption scheme is IND-CPA secure if DDH is hard.

6 Conclusion

Using a computer-aided toolset for proving cryptographic statements is good at handling small details. Those details can easily be seen as trivial and later on discovered as a considerably large problem. Even though the workload rises by a great margin for using EasyCrypt, the result can be used to underline that a proof was checked formally and thoroughly.
The current problem for such toolsets, or automatic provers, lies not only in learning a new computer languag. Cryptography is such a new field that there exists no universal language standard. Therefore every scientific group uses their own style for writing handwritten proofs. That means if one wants to implement their own handwritten proof, they first have to translate it into the style the creators use and only then it can be used for transferring into code. The difference in human language arised as a challenge in this project, since the main issue was in transferring the game-style language from chapter 2 into the language of the authors of EasyCrypt, which their program fully supports.

In my opinion EasyCrypt was particularly helpful at getting a better understanding of structuring a proof in cryptography. Sadly it needed a lot of time to achieve even such a small example, mainly because EasyCrypt is not newcomer-friendly. As a next step the implementation of the pseudorandom generator would be good

Bibliography

[1] Karthikeyan Bhargavan et al. Proving the TLS Handshake Secure (as it is). URL: https://eprint.iacr.org/2014/182. (accessed: 03.08.2018).

[2] IMDEA Software Institute, Inria, and École Polytechnique. EasyCrypt: Computer-Aided Cryptographic Proofs. URL: https://www.easycrypt.info/trac/. (accessed: 03.08.2018).

[3] Jonathan Katz and Yehuda Lindell. Introduction to Modern Cryptography. Chapman & Hall/CRC Cryptography and Network Security Series. AAA, 2016. ISBN: 9781466570269.

[4] Microsoft, Inria, and the Joint Centre. miTLS: A Verified Reference Implementation of TLS. URL: https://mitls.org/. (accessed: 03.08.2018).