NTNU
Norwegian University of
Science and Technology

Hash Proof Based Encryption Schemes in EasyCrypt

Morten Rotvold Solberg

Master of Science
Submission date: May 2018
Supervisor: Kristian Gjøsteen, IMF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Acknowledgements

Before you lies the work concluding my time as a student at NTNU. The project I have been working on this semester have been both interesting, fun, educational, difficult and from time to time seemed to be rather overwhelming. I would not have been able to do this all by my self, and several people deserve my thanks for helping in this project.

First of all I want to thank my supervisor, Kristian Gjøsteen, for valuable feedback and reflections.

I also want to thank the EASYCRYPT developers, and especially François Dupressoir for being so very helpful in answering my questions.

Thanks to my family and my girlfriend for believing in me and for all the love and support and for proofreading my thesis.

Thank you to my fellow students for all the lunch time quizzes, for all the hours in the table tennis room and generally for making these five years of studying as good as they could possibly be.

Morten Rotvold Solberg
Trondheim, May 2018

Abstract

We describe the basics of the proof-assistant EASYCRYPT and explain how to use EASYCRYPT to model encryption schemes and game-based security proofs. Furthermore, we analyze a generic encryption scheme by Cramer and Shoup, which is based on hash proof systems and the difficulty of deciding whether or not an element of a finite set X lies in a certain subset $L \subset X$. We implement this scheme in EASYCRYPT and verify its security proof. We also present some simplifications of the scheme, making it easier to implement at the cost of less security. Finally, we concretize the simplified generic scheme using the Decision Diffie Hellman assumption.

Sammendrag

Vi beskriver de grunnleggende elementene i bevisprogrammet EASYCRYPT, og forklarer hvordan EASYCRYPT kan brukes til å modellere kryptosystemer og spillbaserte sikkerhetsbevis. Videre analyserer vi et generisk kryptosystem av Cramer og Shoup, basert på hash proof-systemer og vanskeligheten av å avgjøre om et element i en endelig mengde X ligger i en spesiell delmengde $L \subset X$ eller ikke. Vi implementerer kryptosystemet i EASYCRYPT og verifiserer systemets sikkerhetsbevis. Vi beskriver også noen forenklinger av systemet som gjør det enklere å implementere, men også mindre sikkert. Til slutt konkretiserer vi det forenklede systemet ved hjelp av Decision Diffie Hellman-antagelsen.

Table of Contents

Chapter 1

Introduction

Cryptography plays a huge role in modern society. We can find cryptography essentially everywhere, from secure communications and electronic elections to new methods of payment. As cryptography becomes more and more central in our modern society, we also demand more and more security. We want to be *certain* that the systems we use are secure against different kinds of adversaries. For this purpose, techniques such as *game hopping* [8] have been developed. Such techniques handle the complexities of cryptographic schemes and protocols and allow us to mathematically prove the security of a cryptosystem under reasonable assumptions.

However, cryptographic schemes and protocols have become so complex that their security proofs are becoming error prone and difficult to verify even with the use of well developed proof techniques. It has even been claimed that many proofs in cryptography have become essentially unverifiable [2]. To deal with this, computer-based proof-assistants designed for automatic proof-verification have been developed. One such proof assistant is EASYCRYPT.

EASYCRYPT is a proof-assistant designed for formal verification of probabilistic, game-based security proofs with unspecified adversarial code. In other words, this proof-assistant will not produce a proof itself. For any scheme and attack, we must first construct a pen-and-paper proof. This proof must then be implemented[1] in EASYCRYPT for verification. The scheme we will look at in this thesis is a generic cryptosystem based on subset membership problems and hash proof systems, proposed by Cramer and Shoup in [3]. We will also look at a concrete construction based on hash proof systems and the Decision Diffie-Hellman assumption.

As the scheme we analyze is well-studied and accepted as secure under the appropriate assumptions, the main goal of this thesis is not to actually find out whether or not the proof holds. Instead, we want to describe how EASYCRYPT is used for this purpose, and to find out whether or not EASYCRYPT is a useful tool for verifying the security of the (fairly simple) scheme in question.

[1] By ”implementation” we will mean writing the EASYCRYPT code modeling mathematical definitions, encryption schemes, security proofs and so on.

The thesis will be organized as follows: in Chapter 2, we provide an introduction to the basic elements of the EASYCRYPT language; in Chapter 3, we look at the cryptographic and mathematical theory we will need in the rest of the thesis, as well as a description of how the definitions we make can be implemented in EASYCRYPT; in Chapter 4 we look at a generic encryption scheme by Cramer and Shoup [3], which uses hash proof systems, sketch its security proof, and describe how both scheme and security proof can be implemented in EASYCRYPT; in Chapter 5 we concretize the generic scheme using the Decision Diffie Hellman assumption as an underlying subset membership problem; in Chapter 6, we summarize our work with some concluding remarks, and make a few suggestions for interesting further work.

Files including the EASYCRYPT implementations of the schemes we work with and all lemmas we describe in the thesis, along with their EASYCRYPT proofs, can be found on GitHub.[2]

[2] `https://github.com/mortensol/EasyCrypt`

Chapter 2

Introductory EasyCrypt

EASYCRYPT is a proof assistant designed for verifying the security of cryptographic constructions [1]. In EASYCRYPT, both security goals and hardness assumptions are modeled as probabilistic games. This chapter will serve as an introduction to EASYCRYPT. We will describe the basic elements of EASYCRYPT as well as the syntax, and we will look at the different forms of logic used to reason about different types of statements. At the end of the chapter we will briefly discuss the run time of an adversary attacking a cryptographic construction. We assume the reader has some basic knowledge of cryptography, such as the notions of plaintexts and ciphertexts.

We will provide a rather short description of the various elements. More detailed explanations can be found in [4] and [9].

The first and perhaps one of the most important things to note, is that at the time of writing, EASYCRYPT is under heavy development. In practice, this means that there is a possibility that parts of the code we write may not work at a later time due to some update, even though it worked fine when the thesis was written.

It should also be noted that there exists an EASYCRYPT user manual [9]. However, at the time of writing, this user manual is also under development, meaning several of the chapters are either missing or incomplete. It does however include much of the basics of EASYCRYPT, and several of the examples provided in this chapter will either be inspired of or taken from the user manual.

As there is no complete user manual, learning EASYCRYPT has to a large extent been done by reading and trying to understand code examples developed by the EASYCRYPT team. In addition, there has been a lot of trial, error and experimenting. Lastly, there is an EASYCRYPT mailing list, where the developers of EASYCRYPT are very helpful in answering questions.[1]

[1] `https://lists.gforge.inria.fr/mailman/listinfo/easycrypt-club`

2.1 The Basics of EasyCrypt

In this section, we will look at the basic elements that make up the EASYCRYPT language. These elements are types, theories, operators, modules, procedures, module types, axioms and lemmas.

2.1.1 Types

We can view types as mathematical sets consisting of at least one element. EASYCRYPT has a few built in types, e.g. `bool, int, real` and `unit`.

In addition to the built-in types, we may define our own *abstract* types. This is done by simply typing

```
type plaintext.
type ciphertext.
```

When defining abstract types, it may also be necessary to provide axioms defining how these types should behave. We can also use type aliases, if we want to name our types for example for readability purposes. Say for example we are working with a group G, and we have plaintexts in G^2 and ciphertexts in G^4. Then using the type aliases

```
type plaintext  = group * group.
type ciphertext = group * group * group * group.
```

makes the code both much more readable and a lot easier to work with than constantly working with the types `group * group` and `group * group * group * group`.

2.1.2 Operators

Operators can be viewed as mathematical functions, defined along with input types and output types. For example, we can define the absolute value operator for integers as

```
op "`|_|" : int -> int = fun x, (0 <= x) ? x : -x.
```

meaning both input and output are integers. Here, we have also used EASYCRYPT's anonymous function **fun**, i.e. a function not bound to a variable or constant, and the built in conditional expression. The conditional expression has the form

```
b ? t : f
```

where `b` is a boolean. If `b` is true, the expression evaluates to `t`, and if `b` is false, it evaluates to `f`. Note that this is equivalent to

```
if b then t else f.
```

We can also use operators to define distributions or finite sets. As an example, consider the following distribution of plaintexts and set of ciphertexts.

```
op dPlain  : plaintext distr.
op sCipher : ciphertext fset.
```

We may add some properties to these distributions and sets. If we want to make sure the distribution is uniform, we define it as follows:

```
op dPlain : { plaintext distr | is_uniform dPlain }
  as dPlain_uni.
```

2.1.3 Theories

As mentioned, EASYCRYPT has some built-in types. These built-in types are defined in files referred to as *theories*. As an example, basic usage of integers is defined in the theory called `Int.ec`. Before we can use the built-in types, we need to load the corresponding theories into our context. This is done by using the keywords **`require`** and **`import`**. Loading the theory into our context by typing

```
require Int
```

will allow us to work with integers. For example, writing

```
const x : int = Int.(+) 10 24
```

will make `x` evaluate to 34.

If we in addition to "requiring" a theory also import it, by

```
require import Int
```

we can use the operators defined in the `Int` theory in a more familiar way and without specifying which theory we are using. For example, writing

```
const x : int = 10 + 24
```

will also make `x` evaluate to 34.

2.1.4 Modules, Procedures and Module Types

Modules are used to model algorithms like encryption and decryption, as well as cryptographic games. Games will be discussed in greater detail in §3.2. A module consists of global variables and procedures, and a procedure consists of local variables, random assignments, regular assignments and calls to other procedures. A global variable may be used by any procedure inside the module, as long as it is defined prior to the procedure. A local variable however, can only be used inside the procedure where it is defined. Similarly, a procedure can make a call to any other procedure, as long as the procedure called is defined prior to the procedure calling it. We now provide an example of how to define a module with a few procedures.

```
module M = {
  var x   : int
  var bnd : int
  var b   : bool
```

```
  proc init_x() : unit = {
    x <$ [-bnd .. bnd];
  }

  proc get_x() : int = {
    init_x();
    return x;
  }

  proc get_b() : bool = {
    b <$ {0,1};
    return b;
  }
}.

module Get_b_from_M = {
  proc main() : bool = {
    var b : bool;
    b <@ M.get_b();
    return b;
  }
}.
```

The first module consists of three global variables: the integer `x`, the integer `bnd` defining the interval from which we can sample x, and the boolean `b`. The first procedure shows how we can define a procedure that does something without returning anything (i.e. it has return type `unit`). The second procedure makes a call to the first procedure and then returns the sampled value. The third procedure shows how we can perform both those actions in a single procedure, and samples a boolean before returning it. The second module has no global variables, but has a procedure consisting of a local variable and a procedure call to a procedure from a different module. The two procedures `M.get_b()` and `Get_b_from_M.main()` are equivalent.

A *module type* is similar to a module. In a module type, we also define different procedures along with their input and output types. However, we do not concretely state what the procedures should do. This is particularly useful when defining adversaries. In this way, we can define an abstract adversary, making sure the proof holds for any adversary with given properties. Concrete examples of this will be given in §3.1.2.

2.1.5 Axioms, Lemmas and SMT Solvers

Lemmas and axioms are statements that are saved and can be used later. The main difference between a lemma and an axiom is that axioms are trusted by EASYCRYPT, while lemmas need to be proved before they can be used in other settings. Other than that, axioms and lemmas are stated equivalently. To exemplify, we look at the fact that for all integers $i \geq 0$, we have $i + 1 > 0$. In EASYCRYPT, we can state this as an axiom or a lemma in the following way.

```
axiom a1 : forall (i:int), 0 <= i => 0 < i + 1.
axiom a2 (i:int) : 0 <= i => 0 < i + 1.

lemma l1 : forall (i:int), 0 <= i => 0 < i + 1.
proof.
...
qed.
```

The two axioms show two slightly different but equivalent ways of stating an axiom, the only difference being the notation for telling EASYCRYPT what variables we are using. These two ways also work for lemmas. Starting a proof will result in a *goal*. A goal consists of *assumptions* and a *conclusion*. In `lemma l1` above, the assumption list would consist of `i : int` and `0 <= i` and the conclusion (what we want to prove) would be `0 < i + 1`.

The proof body of the lemma (i.e. the dots between `proof` and `qed`) consists of so called *tactics*. Tactics are certain keywords that are used to either transform or break the conclusion down into simpler logic statements. We will not explain many tactics in detail in this thesis, but detailed explanations of (almost) all available tactics can be found in [9]. We will also omit the proofs of the lemmas we prove in EASYCRYPT in this text, but as mentioned, all the proofs can be found in the code files on GitHub[2]. The reason is that the proofs tend to be long and spacious and look more like noise than actual proofs. As an example (from the file `CCA-masters-finished.ec`), a typical lemma along with a proof may look as follows:

```
local equiv CCA_Exp1G0_main :
  CCA(Genscheme,A).main ~ Exp1G0(A).main :
  ={glob A} ==> ={res}.
proof.
proc; inline*.
call (_: CCA.log{1} = Exp1G0.log{2}
  /\ CCA.sk{1} = Exp1G0.sk{2}
  /\ CCA.cstar{1} = Exp1G0.cstar{2}).
proc; inline*.
if => //. sp 8 6.  if => //. smt.
wp;skip;first smt.
wp;skip;first smt.
wp;skip;first smt.
swap{1} [20..22] -4.
wp. rnd.
call (_: CCA.log{1} = Exp1G0.log{2}
  /\ CCA.sk{1} = Exp1G0.sk{2}
  /\ CCA.cstar{1} = Exp1G0.cstar{2}).
proc; inline*.
if => //=. sp 8 6. if => //=. smt.
wp. skip. smt. wp. skip. smt. wp; skip. smt.
wp. do 2! rnd. wp. sp. rnd; wp; rnd; skip.
move => ?????; split; first smt.
move => ?; split; first smt.
```

[2] https://github.com/mortensol/EasyCrypt

```
move => ???. split; first smt.
move => ????. split; first smt.
move => ???. split; first smt.
move => ???. split; first smt.
move => ?. split; first done.
move => ?. split. smt.
move => ??????????. split. by exact/H13.
move => ???????. split. split.
have -> : c_L =
  (x0L, e_L, proj_ (pk_L.`2, x0L, e_L, w0L)) by smt.
have -> : c_R =
  (x0L, estar_R, hash_ (sk_R.`2, x0L, estar_R)) by smt.
have -> : estar_R = e_L.
have -> :  estar_R =
  toY (toint (if bL then result_R.`2 else result_R.`1) +
  toint (hash (sk_R.`1, x0L))) by smt.
have -> : e_L =
  toY (toint (if bL then result_L.`2 else result_L.`1) +
  toint (proj (pk_L.`1, x0L, w0L))) by smt.
have -> : sk_R.`1 = sk_L.`1 by smt.
have -> : hash (sk_L.`1, x0L) =
  proj (pk_L.`1, x0L, w0L) by smt. smt.
have <- : sk_L.`2 = sk_R.`2 by smt. smt.
split; first smt.
have -> : sk_L = sk_R by smt.
have -> : log_L = log_R by smt.
have -> : cstar_L = cstar_R.
have -> : cstar_L = Some c_L by smt.
have -> : cstar_R = Some c_R by smt.
have -> : c_L =
  (x0L, e_L, proj_ (pk_L.`2, x0L, e_L, w0L)) by smt.
have -> : c_R =
  (x0L, estar_R, hash_ (sk_R.`2, x0L, estar_R)) by smt.
have -> : estar_R = e_L.
have -> :  estar_R =
  toY (toint (if bL then result_R.`2 else result_R.`1) +
  toint (hash (sk_R.`1, x0L))) by smt.
have -> : e_L =
  toY (toint (if bL then result_L.`2 else result_L.`1) +
  toint (proj (pk_L.`1, x0L, w0L))) by smt.
have -> : sk_R.`1 = sk_L.`1 by smt.
have : result_R = result_L by smt.
have -> : hash(sk_L.`1, x0L) =
  proj (pk_L.`1, x0L, w0L) by smt. smt.
have -> : sk_R.`2 = sk_L.`2 by smt. smt. trivial.
move => ????????.
have -> : result_R0 = result_L0 by smt. trivial.
qed.
```

Simple logic statements can be verified by EASYCRYPT's built-in ambient logic, or

they can be sent to external *SMT solvers* (carried out by the proof tactic `smt`). SMT solvers are external programs, that can take simple lemmas or logic statements as input. After working on these statements, the SMT solvers either return that the statement was true, or that they were not able to solve it. In the latter case, this can mean that the statement was false, but it can also mean that it was true, but too complicated for the SMT solver to solve.

SMT solvers can be very useful, but they are also unpredictable and somewhat error-prone. We will provide a few examples to illustrate that SMT solvers should be used with care and that they may not work even if the statement we want to prove is more or less trivial.

A fellow student, also working with EASYCRYPT, recently found a bug in one of the SMT solvers used by EASYCRYPT. This student was working on cyclic groups, using the built-in theory `CyclicGroup.ec` developed by the EASYCRYPT team. He then found that for a cyclic group of large prime order q, one of the SMT solvers accepted the following lemma.

```
lemma X : 1%r/q%r = 1%r.
```

It turned out there was a bug in an SMT solver named *Alt-Ergo*, which had to be removed from the list of available SMT solvers. The bug was later fixed, but we believe this shows that SMT solvers should be used with care. We should be very certain that the statement we send is in fact true, as this shows that there may exist bugs allowing us to prove lemmas that are in fact false. In fact, it may be wise to try avoiding SMT solvers altogether, at least as far as possible. Experience has shown that avoiding SMT solvers in full, may not be possible, or at least very difficult. As can be seen in the proof listed above, we do use SMT solvers quite often, as we have not found a way to avoid them in this case.

In another case, we were also working with cyclic groups, and wanted to prove the following relation.

```
m{hr} * gen0 ^ w0 ^ k0 * gen1 ^ w0 ^ k10 /
  (gen0 ^ w0 ^ k0 * gen1 ^ w0 ^ k10) = m{hr} * g1.
```

Here, `g1` is the identity element of the group. The SMT solvers used by EASYCRYPT accepts the assumption

```
gen0 ^ w0 ^ k0 * gen1 ^ w0 ^ k10 /
  (gen0 ^ w0 ^ k0 * gen1 ^ w0 ^ k10) = g1
```

as correct, but for some reason, they are not able to prove the preceding conclusion. This shows that the SMT solvers may be unpredictable and that they may have trouble proving even as simple statements as this. In this case, it was possible to prove the conclusion without the use of SMT solvers, but it required a rather complicated and tedious transformation.

2.2 Hoare Logic and Probability Assignments

The logic and proofs in EASYCRYPT are largely based on *Hoare logic* [6]. The perhaps most important notion in Hoare logic is that of the *Hoare triple*. A Hoare triple consists of a precondition P, a postcondition R and and a program Q. The pre- and postconditions

are logic statements. The hoare triple $P\{Q\}R$ informally means that if the precondition is true before the execution of the program, the postcondition will be true when the program terminates.

In EASYCRYPT, we make use of three types of Hoare logic: a regular, possibilistic Hoare logic (HL), a probabilistic Hoare logic (pHL) and a probabilistic relational Hoare logic (pRHL) [9].

Regular Hoare logic is a regular Hoare triple, checking whether or not the postcondition is true after the program terminates. Probabilistic Hoare logic analyzes the probability that the postcondition is true after the program terminates, and probabilistic relational Hoare logic relates two programs checking whether or not they are equivalent given appropriate pre- and postconditions.

As a few examples of how to use the three logics in EASYCRYPT, we consider throwing a pair of dice. We first look at the throw of a six sided die and use HL to prove that the output will be in $\{1, ..., 6\}$. The theories we need to require and import are `AllCore, DInterval` and `Distr`.

```
module Dice1 = {
  proc throw() : bool = {
    var x:int;
    x <$ [1 .. 6];
    return (x \in [1 .. 6]);
  }
}.

lemma l1 : hoare[Dice1.throw : true ==> res].
```

We set up the HL statement by specifying which module and procedure we look at, followed by the precondition and postcondition. Here, we do not have any particular precondition which we need for the statement to be true, so we simply write `true` as the precondition. Writing `res` as postcondition is equivalent to writing `res = true`, i.e. saying that the value returned by the procedure in question is `true`.

To consider the concrete probability that the throw of a die results in some value, say 6, we use pHL. The notation `%r` following the integers in the example below, means that in this case, we evaluate 1 and 6 as real numbers rather than integers. The reason is that there is no division operator for integers.

```
module Dice2 = {
  proc throw() : bool = {
    var x:int;
    x <$ [1 .. 6];
    return (x=6);
  }
}.

lemma l2 :
  phoare[Dice2.throw : true ==> res] = (1%r/6%r).
```

Neither here do we need any particular precondition. As an example of a case where we do need to specify a precondition other than just `true`, we model the throw of one die with

n sides and one die with m sides. We then look at the probability of throwing n and m respectively, and prove that these probabilities are the same provided that $n = m$.

```
op n : { int | 0 < n } as gt0_n.
op m : { int | 0 < m } as gt0_m.

module Dice3 = {
  proc throw() : bool = {
    var x:int;
    x <$ [1 .. n];
    return (x=n);
  }
}.

module Dice4 = {
  proc throw() : bool = {
    var x:int;
    x <$ [1 .. m];
    return (x=m);
  }
}.

lemma l3 :
 equiv[Dice3.throw ~ Dice4.throw : n = m ==> ={res}].
```

We can also state lemmas using probability expressions that may seem a bit more familiar than Hoare logic statements. For example, `lemma l2` above can be written using a probability statement like this:

```
lemma l2_pr &m :
  Pr[Dice2.throw() @ &m : res] = 1%r/6%r.
```

Note that here we have to specify some memory for which to work in (`&m`). This is done automatically when using Hoare logic statements. Also, in many cases, starting up with a probability statement will require you to transform the goal to a Hoare logic statement when proving it. Thus, for most practical purposes it is better to use pHL or pRHL when working with lemmas involving probability. One exception, though, is if we want to prove a concrete cryptographic reduction, say of the sort

```
`|Pr[CPAattack.main() @ &m : res] - 1%r/2%r| <=
`|Pr[DDH1.main() @ &m : res] -
 Pr[DDH0.main() @ &m : res]|.
```

Here, it is best to start out using Hoare logic, and then transform the Hoare logic statements to regular probability statements. We will do exactly that later in the thesis, when proving the security of the generic Cramer-Shoup scheme based on hash proof systems.

Two tactics we will use extensively in the thesis are **`byphoare`** and **`byequiv`**. The first is (among other things) used to prove a probability statement equivalent to an already proven pHL statement. The second is used to prove a probability statement connecting two

equivalent procedures. As an example, we can use the pHL based lemma, `lemma l2` above to prove `lemma l2_pr`. The proof goes as follows:

```
lemma l2_pr &m :
  Pr[Dice2.throw() @ &m : res] = 1%r/6%r
  by byphoare l2.
```

The `byequiv` tactic is used in exactly the same way, when we want to prove the equality of two probability expressions.

Note that when the proof is as short as for `lemma l2_pr`, we can simply use the `by` keyword and omit `qed`.

2.3 Run Time

Usually when working with the security of any cryptographic construction, we take into consideration how much time an adversary is allowed to spend in attacking the construction.[3] A common technique for proving the security of a cryptographic construction is to make a *reduction* from breaking the cryptographic construction to solving some other problem, say $\mathcal{P}$. This means that if an adversary is able to solve $\mathcal{P}$, he can use this solution to break the cryptographic construction.

As a concrete example of why the run time is important, we consider the *discrete logarithm problem* as described in [7]. The discrete logarithm problem is as follows: given a generator g for a cyclic group G as well as an element $x = g^a \in G$, determine the value of a. The discrete logarithm problem is considered to be hard in certain groups.

Now assume we are working with a cryptographic construction that the adversary is able to break, if he is able to compute discrete logarithms. It is evident that any adversary can solve the discrete logarithm problem by simply trying every possible value of a and see which value fits. However, for large groups, this will on average take a long time. Thus, we modify the security assumption by saying that our adversary is able to break the cryptosystem if he is able to compute discrete logarithms in a *reasonable amount of time*. Restated, we can say that for all adversaries $\mathcal{A}$ and for all small positive real numbers ϵ, there exists a time t such that if the run time of $\mathcal{A}$ is less than t, the advantage[4] of $\mathcal{A}$ in breaking the cryptographic construction is less than ϵ.

However, EASYCRYPT is not designed to reason about time complexity. Thus, as far as EASYCRYPT is concerned, there will always exist an adversary able to compute discrete logarithms. Therefore, assuming that the advantage of $\mathcal{A}$ is less than some number ϵ will be equivalent to assuming something false in EASYCRYPT. This means that in EASYCRYPT, we need to work with problems involving run time differently than we do on paper. Instead of working with ϵ, we will simply work with the explicit probability that the adversary is able to solve certain problems. Concrete examples of this will be given in Chapter 4.

[3] Adversaries and attacks will be discussed further in the next chapter.

[4] The advantage is taken to be the probability of some event and will be discussed further in the next chapter

Chapter 3

Theory

In this chapter we introduce the cryptographic and mathematical theory used throughout the thesis. We start by describing what a public key encryption scheme is, as well as two notions of security for such a scheme. We then describe how we can deal with the complexity of the security of encryption schemes (and other cryptographic constructions) using sequences of games. Lastly, we look at theory about subset membership problems, before we define projective, universal and smooth hashing as well as hash proof systems. Along with each definition we provide a description of how the definitions we make can be implemented in EASYCRYPT. We make no claims that our implementations are neither the only nor the best ways to make the implementations.

3.1 Public Key Cryptography

In this section we will look at the definition of a public key cryptosystem, as well as what it means for such a system to be secure. We will explain how to prove the security by looking at two notions of security, namely indistinguishability under chosen plaintext and chosen ciphertext attacks (resp. IND-CPA and IND-CCA attacks).

3.1.1 Definition of a Public Key Encryption System

A public key encryption system consists of three algorithms: a key generation algorithm, an encryption algorithm and a decryption algorithm [8].

The key generation algorithm `KeyGen` takes no input and returns a pair of keys, (pk, sk). Here, pk is the public encryption key, and sk is the decryption key, which should be kept secret. The encryption algorithm `Encrypt` takes the public key pk along with a plaintext m as input and returns a ciphertext c. The decryption algorithm `Decrypt` takes the secret key sk and a ciphertext c as input and returns a plaintext m.

In addition to these three algorithms, we also have a *correctness requirement*. We

require that for all plaintexts m and all key pairs (pk, sk), we have

$$\texttt{Decrypt}(sk, \texttt{Encrypt}(pk, m)) = m.$$

In other words, if we encrypt a message under some public key, decrypting the resulting ciphertext under the corresponding secret key should result in the original message.

We can implement and prove such a correctness property in EASYCRYPT. For efficient reuse of code, we can first define an abstract module type and then a general correctness module.

```
type pkey, skey, plaintext, ciphertext.

module type Scheme = {
  proc keygen()                          : pkey * skey
  proc encrypt(pk:pkey, m:plaintext)   : ciphertext
  proc decrypt(sk:skey, c:ciphertext) : plaintext
}.

module Correctness(S:Scheme) = {
  proc main(m:plaintext) : bool = {
    var pk, sk, c, m';
    (pk,sk) <- S.keygen();
       c    <- S.encrypt(pk,m);
       m'   <- S.decrypt(sk,c);
    return (m' = m);
  }
}.
```

Now we have a module `Correctness` which can take as input any module with type `Scheme`. This module has a procedure which takes a plaintext as input, generates a pair of keys, encrypts the plaintext and decrypts the resulting ciphertext, using the algorithms of the module given as input. Finally it returns `true` if the decryption matches the original plaintext and `false` otherwise.

For any implemented cryptosystem of type `Scheme` with name say `Testsystem`, we can now prove its correctness by using a Hoare statement.

```
lemma testcorrect :
  hoare[Correctness(Testsystem).main : true ==> res].
```

Recall that this means that the postcondition is true after executing the procedure `Correctness(Testsystem).main`.

3.1.2 Two Types of Attacks

Here, we will define two different attacks against a public key cryptosystem, namely a *chosen plaintext attack* (CPA) and a *chosen ciphertext attack* (CCA). Both attacks will be defined under the security notion of *indistinguishability* (IND). This notion of security considers how difficult it is for an adversary to decide which of two plaintexts was encrypted, given a ciphertext.

A chosen plaintext attack can be defined as an attack game between a challenger and an adversary in the following way. The challenger starts by computing (pk, sk) using the cryptosystem's key generation algorithm and sends the public key to the adversary. The adversary then chooses two plaintexts m_0 and m_1 from the message space, and sends these to the challenger. The challenger samples a random bit b (from $\{0, 1\}$) and encrypts m_b. The resulting *target ciphertext* $c^* = \mathcal{E}(pk, m_b)$ is sent to the adversary, who outputs a bit b' according to which plaintext he believes was encrypted. We define the adversary's *advantage* to be $|\Pr[b = b'] - 1/2|$. The cryptosystem is secure if this advantage is negligible, i.e. the adversary cannot do much better than guessing.

To implement this in EASYCRYPT, we first define a module type which we call `CPAadversary`.

```
module type CPAadversary = {
  proc choose(pk:pkey)     : plaintext * plaintext
  proc guess(c:ciphertext) : bool
}.
```

Here we have defined an abstract adversary. The only thing we know is that the adversary is allowed to choose two plaintexts possibly depending on the public key, and to guess which of the plaintexts was encrypted. Exactly how the adversary does any of this is unknown to us.

For the possibility of reusing the code, we define a general module modeling a CPA attack.

```
module CPA(S:Scheme, A:CPAadversary) = {
  proc main() : bool = {
    var pk, sk, c, m0, m1, b, b';
    (pk, sk)  <- S.keygen();
    (m0, m1)  <- A.choose(pk);
       b      <$ {0,1};
       c      <- S.encrypt(pk, b?m1:m0);
       b'     <- A.guess(c);
     return (b = b');
  }
}.
```

Our CPA module takes as input any module of type `Scheme` and any adversary of type `CPAadversary`, and carries out an attack as described above. For any cryptosystem, let us call it `Testsystem`, and any adversary `A` of correct respective types, we can define the adversary's advantage in EASYCRYPT as

```
`|Pr[CPA(Testsystem,A).main() @ &m : res] - 1%r/2%r|.
```

A more complicated and much stronger attack is the chosen ciphertext attack, which we split up into five phases as in [3].

- *Key generation phase.* The challenger generates (pk, sk) and sends pk to the adversary. In addition, pk is sent to an encryption oracle, and sk is sent to a decryption oracle.

- *Probing phase 1.* In this phase, the adversary interacts with the decryption oracle by sending the oracle a query on which the oracle runs the decryption algorithm and sends the result back to the adversary. This interaction may be adaptive in the sense that one query can be chosen dependently of the previous queries.
- *Target selection phase.* Similar to a chosen plaintext attack. The adversary chooses two plaintexts m_0 and m_1 and sends these to the encryption oracle. The encryption oracle encrypts m_b where b is a random bit and sends the resulting ciphertext c^* to the adversary.
- *Probing phase 2.* This phase is almost the same as probing phase 1, with the difference being that the adversary is not allowed to send the target ciphertext c^* as a query to the decryption oracle. Also note that the total number of queries to the decryption oracle is bounded by some integer Q.
- *Guessing phase.* Finally the adversary outputs a bit b' and wins the game if $b = b'$. The adversary's advantage is defined as in a chosen plaintext attack, i.e. $|\Pr[b = b'] - 1/2|$.

Implementing this attack model in EASYCRYPT is a bit more difficult than implementing the CPA attack. We first need to define an oracle. In this particular case, the oracle should have only one procedure, namely a decryption procedure.

```
module type Oracle = {
  proc decrypt(c:ciphertext) : plaintext
}.
```

We define a CCA adversary almost like the CPA adversary, but in this case we also need to give the adversary access to the oracle.

```
module type CCAadversary(O:Oracle) = {
  proc choose(pk:pkey) : plaintext * plaintext
    {O.decrypt}
  proc guess(c:ciphertext) : bool {O.decrypt}
}.
```

Using bracket notation as above, we give the adversary access to the oracle in both the choose and the guess procedures. This will make the attack *adaptive*. Not giving the adversary access to the oracle in the guess procedure (i.e. replacing {O.decrypt} with { }) would result in an attack often called *CCA1* or *non-adaptive CCA*. This is a similar, but weaker attack, where the adversary is not allowed to choose the decryption queries dependant on each other.

Unlike a CPA module, the CCA module will consist of a few *submodules* in addition to the attack itself. We also need to put a bound on how many queries the adversary is allowed to make to the decryption oracle. We will call this bound `qD`.

```
op qD : int.
axiom qDpos : 0 < qD.

module CCA (S:Scheme, A:CCAadversary) = {
```

```
  var log   : ciphertext list
  var cstar : ciphertext option
  var sk    : skey

  module O = {
    proc decrypt(c:ciphertext) : plaintext option = {
      var m : plaintext option;
      if (size log < qD && Some c <> cstar) {
        log <- c :: log;
        m   <- S.decrypt(sk,c)
      }
      else m <- None;
      return m;
    }
  }

  module A = A(O)

  proc main () : bool = {
    var pk, m0, m1, c, b, b';
      log    <- [];
     cstar   <- None;
    (pk, sk) <- S.keygen();
    (m0, m1) <- A.choose(pk);
       b     <$ {0,1};
       c     <- S.encrypt(pk, b?m1:m0);
     cstar   <- Some c;
       b'    <- A.guess(c);
     return (b = b');
  }
}.
```

We define the variables `log, cstar` and `sk` globally, as they are used in both the `main` procedure and in the decryption oracle. Defining the module `A = A(O)` means we give the adversary access to the oracle during the attack, and each time we make a call to a procedure of the adversary (i.e. `A.choose` or `A.guess`) we must go through the decryption oracle. The notation `c :: log` used in the oracle module means that for each ciphertext the oracle receives, this ciphertext is prepended to the list `log`.

Similarly as in the CPA attack, we can define an adversary's CCA advantage against a cryptosystem named `Testsystem` as follows.

```
Pr[CCA(Testsystem, A).main() @ &m : res] - 1%r/2%r.
```

3.2 Attack Games

The security of a cryptosystem can be defined and proved by using a sequence of *attack games* [8]. An attack game G is played between some *challenger* and an *adversary*. The adversary is the one attacking the cryptosystem. The challenger and the adversary are both

probabilistic algorithms, so an attack game can be modeled as a probability space. Usually, the security of a cryptosystem is linked to some probability, $\Pr[E]$, where E is some event in the probability space. Typically, the goal is to prove that $\Pr[E]$ is negligibly close to some *target probability*, often $0, 1/2$ or $\Pr[E']$ where E' is an event in some attack game where the same adversary plays against a different challenger.

To do this, we define a sequence of games, from G_0 to G_n, where n is some constant natural number and G_0 is the original attack game against the cryptosystem. We define E_0 to be the event E mentioned above. The games should be defined such that for $i \in \{0, ..., n-1\}$, each transition from G_i to G_{i+1} should change as little as possible, to make the analysis as easy as possible. In addition, we want $\Pr[E_i]$ to be negligibly close to $\Pr[E_{i+1}]$ for $i \in \{0, ..., n-1\}$ and we want $\Pr[E_n]$ to be negligibly close to the target probability. This will result in $\Pr[E_0] = \Pr[E]$ being negligibly close to the target probability.

A transition between two games is usually one of three types [8]. The first, and simplest, transition is a *bridging step*. Here, the difference between two games G_i and G_{i+1} is purely conceptual, and we simply move from G_i to G_{i+1} by restating how certain quantities are computed. In a bridging step, these two ways of computing this quantity should be completely equivalent, and hence we get $\Pr[E_i] = \Pr[E_{i+1}]$. The reason for using a bridging step is to prepare for a transition of one of the two next types.

The second type of transition is based on *indistinguishability*. Here, we have two games with a small difference which, if detected by the adversary, will imply an efficient method or algorithm for distinguishing between two distributions (say D_1 and D_2) that are indistinguishable. Let G_i and G_{i+1} be two games. To prove that $|\Pr[E_i] - \Pr[E_{i+1}]|$ is negligible, we argue that there exists a distinguishing algorithm $\mathcal{D}$, that interpolates between the two games. We say that $\mathcal{D}$ wins if it is able to distinguish between the two distributions. When given an element from the distribution D_1, $\mathcal{D}$ wins with probability $\Pr[E_i]$ and when given an element from D_2, $\mathcal{D}$ wins with probability $\Pr[E_{i+1}]$. The assumption that D_1 and D_2 are indistinguishable then implies that $|\Pr[E_i] - \Pr[E_{i+1}]|$ is negligible.

The final transition type is based on *failure events*. In a transition like this, two games G_i and G_{i+1} will proceed identically unless some failure event F occurs. More formally, this can be written as $E_i \wedge \neg F \iff E_{i+1} \wedge \neg F$. This means that the events $E_i \wedge \neg F$ and $E_{i+1} \wedge \neg F$ are the same, and as long as this is true, the following simple but useful lemma holds [8].

Lemma 3.1. *Let A, B and F be events in some probability space such that* $A \wedge \neg F \iff B \wedge \neg F$. *Then* $|\Pr[A] - \Pr[B]| \leq \Pr[F]$.

Proof. First, note that if event A occurs, this will be either in combination with event F occurring or event F not occurring. Thus, $\Pr[A] = \Pr[A \wedge F] + \Pr[A \wedge \neg F]$. This is similar for the event B. Since $A \wedge \neg F \iff B \wedge \neg F$, we have $\Pr[A \wedge \neg F] = \Pr[B \wedge \neg F]$. Also, both $\Pr[A \wedge F]$ and $\Pr[B \wedge F]$ are numbers between 0 and $\Pr[F]$. Thus, we get

$$\begin{aligned} |\Pr[A] - \Pr[B]| &= |\Pr[A \wedge F] + \Pr[A \wedge \neg F] - \Pr[B \wedge F] - \Pr[B \wedge \neg F]| \\ &= |\Pr[A \wedge F] - \Pr[B \wedge F]| \\ &\leq \Pr[F]. \end{aligned}$$

□

Note that for this to be true, A, B and F must be defined on the same underlying probability space.

Using this lemma, we see that proving that $\Pr[F]$ is negligible implies that $\Pr[E_i]$ is negligibly close to $\Pr[E_{i+1}]$.

3.3 SMP and HPS

In this section we describe some basic preliminaries as well as the concepts of subset membership problems and hash proof systems. All definitions in this section come from [3] and [5].

3.3.1 Preliminaries

We first define some basic notation as well as the notion of statistical distance between two random variables.

We will denote by $x \leftarrow \alpha$ the action of assigning the value of α to the variable x. We will denote by $x \xleftarrow{\mathrm{r}} S$ the action of sampling an element according to the distribution on S.

Let X and Y be two random variables that can take values in a finite set S. We define the statistical distance between X and Y to be

$$\mathrm{Dist}(X, Y) = \frac{1}{2} \cdot \sum_{s \in S} |\Pr[X = s] - \Pr[Y = s]|.$$

We call the two variables X and Y ϵ-close if $\mathrm{Dist}(X, Y) \leq \epsilon$.

This notion of statistical distance is used in [3] and is important in the pen-and-paper proofs for the security of the schemes we define in Chapter 4. However, there does not seem to be a good way of defining statistical distance in EASYCRYPT (as far as we have found). Thus, we need to implement definitions involving statistical distance differently than we do on paper. Concrete examples of how we do it will be given in Chapter 4.

3.3.2 Subset Membership Problems

We now define the notion of a *subset membership problem*. A subset membership problem **M** specifies a collection of probability distributions I, consisting of so called *instance descriptions*, denoted Λ or $\Lambda[X, L, W, R]$. Let $[I]$ denote the set of instance descriptions that can be sampled from I with positive probability. An instance description $\Lambda = \Lambda[X, L, W, R]$ specifies three finite sets X, L and W, where $L \subset X$, as well as a binary relation $R \subset X \times W$. For any $x \in X, w \in W$ such that $(x, w) \in R$, w will be called a *witness* for x.

In addition, a subset membership problem **M** specifies a couple of algorithms:

- *Instance sampling algorithm*. This is a probabilistic algorithm which samples an instance description Λ from I. It is required that the output distribution of this algorithm is ι-close to I. $\iota \geq 0$ is some real number called the *approximation error* of the algorithm.

- *Subset sampling algorithm.* This is a probabilistic algorithm which, on input $\Lambda \in [I]$, samples a random $x \in L$ together with a witness w for x. It is required that the algorithm always outputs an element in L, and that the output distribution and the uniform distribution on L are ι'-close. $\iota' \geq 0$ is the approximation error of this algorithm.

We also need to define the notion of a *hard subset membership problem*. Informally, this means that given a random element $x \in X$, it is hard to decide whether this element lies in L or in $X \setminus L$. We define this more formally in Definition 3.1 below.

Definition 3.1. Let $\mathbf{M}$ be a subset membership problem. We first sample the following:

$$\Lambda \xleftarrow{r} I, x \xleftarrow{r} L, x' \xleftarrow{r} X \setminus L.$$

We then define the random variables $U(\mathbf{M}) = (\Lambda, x)$ and $V(\mathbf{M}) = (\Lambda, x')$. $\mathbf{M}$ is called *hard* if $U(\mathbf{M})$ and $V(\mathbf{M})$ are hard to distinguish.

The first thing we need to do when implementing this in EASYCRYPT, is to define the sets X, L and W. To accomplish this, we define operators and use the type constructor `fset`. As we are going to sample elements from these sets, we also need to define distributions of the elements in the sets. Note that we do not explicitly implement the instance sampling algorithm, but rather work with fixed sets X, L and W.

```
type X.
op xs : { X fset | is_lossless (duniform (elems xs))
  /\ is_full (duniform (elems xs))
  /\ is_uniform (duniform (elems xs)) }
  as xs_lfu.
```

The `elems` operator transforms the set into a list, and the `duniform` operator transforms this list into a distribution. The reason we take this detour instead of defining X as a list or distribution in the first place, is that we have to define L as a subset of X, and subsets seem to be more developed than sub-lists or sub-distributions. Also, there does not seem to be a way to transform a finite set to a distribution directly, so we go via a list.

The predicate `is_lossless` makes sure that sampling from this distribution always terminates, the predicate `is_full` means that every single element of type `X` is in the distribution, and the predicate `is_uniform` makes sure that sampling from this distribution is done uniformly.

Defining the set L is done almost equally to X. However, as L is a proper subset of X, we need the elements in L to be of type `X` as well. Therefore, we cannot define L to be full, as this in practice would mean $L = X$. Instead, we use the subset predicate `<`.

```
op ls : { X fset | ls < xs
  /\ is_lossless (duniform (elems ls))
  /\ is_uniform (duniform (elems ls))} as ls_luf.
```

Experiential, it is good practice to perform certain checks that everything works the way we want it to work. For example, we can list a few lemmas to check whether a proper subset in EASYCRYPT behaves the way we would expect it to. A few such lemmas are listed below.

```
lemma sub1 : ls < xs by [].
lemma sub2 : forall(x:X), xs `\` ls <> fset0 by [].
lemma sub3 : forall(x:X), x \in ls
  <=> !x \in xs `\` ls by [].
lemma sub4 : ls `&` (xs `\` ls) = fset0 by [].
```

For clarification: the operator `` `\` `` denotes set difference, `fset0` denotes the empty set, `<>` denotes inequality and `` `&` `` denotes set intersection.

When implementing the subset sampling algorithm of our subset membership problem, we need a way to sample a witness for any x sampled from L. The fact that there exists a witness for the membership of x in L, can be viewed as the existence of a function $wit : W \to L$ such that for all $x \in L$, there exists $w \in W$ such that $wit(w) = x$. In other words, we want the function to be surjective on L. In EASYCRYPT, we can implement this as follows.

```
type W.
op wit : W -> X.
axiom witsur : forall(x:X), x \in ls =>
  exists(w:W), wit w = x.
```

Further, when sampling a witness w for a given x, we must be sure that w is in fact a witness. For this purpose we define an operator that takes as input an x and for any $w \in W$ returns true if $wit(w) = x$ and false otherwise.

```
op iswit x : fun (w:W) => wit w = x.
```

This implementation of witnesses will serve as our implementation of the binary relation R mentioned above.

We are now ready to implement the subset sampling algorithm, which samples an x from L along with a witness w. Later in this thesis, we will provide a security proof where we have to sample an element from $X \setminus L$, so we will wrap this inside the same module as the procedure that samples an element from L.

```
module Sampling = {
  proc fromL() : X * W = {
    var x, w;
    x <$ duniform (elems ls);
    w <$ duniform (elems (filter (iswit x) ws));
    return (x,w);
  }

  proc fromXnotL() : X = {
    var x;
    x <$ duniform (elems (xs `\` ls));
    return x;
  }
}.
```

In the sampling procedure `Sampling.fromL()`, we use the `filter` operator from the `List.ec` theory to make a list consisting of all the elements in `ws` that are witnesses of x.

Again we can list some lemmas to check whether the sampling algorithms behave as we expect them to. One example of such a lemma follows here.

```
lemma sampletest :
  hoare[Sampling.fromL : true ==> res.`1 \in ls].
```

The notation `res.`1` means the first element of the result of running `Sampling.fromL` (in this case the result will be the tuple `(x,w)`, so `res.`1` is in this case `x`).

3.3.3 Projective Hashing

We now move on to define the concept of projective hashing.

Let X and Y be two finite sets, $X, Y \neq \varnothing$. Let $\mathcal{H} = \{H_k\}_{k \in K}$ be a set of hash functions indexed by K such that each H_k is a function from X into Y. Let L be a non-empty proper subset of X and let S be some finite non-empty set. Let $\alpha : K \to S$ be a function and let $\mathbf{H} = (\mathcal{H}, K, X, Y, L, S, \alpha)$.

Definition 3.2. Let $\mathbf{H}$ be defined as above. $\mathbf{H}$ is called a *projective hash family* for (X, L) if the action of H_k on L is determined by $\alpha(k)$ for all $k \in K$.

In other words, given $s = \alpha(k)$ and $x \in L$, we can easily calculate the hash value of x, $H_k(x)$, even though we know nothing about the hash key k.

When implementing the projective property in EASYCRYPT, we also include a witness for x in the projection, as we need that in a later definition. We start by defining the types we have not yet defined, as well as the function α and abstract operators `hash` and `proj`.

```
type K, S, Y.

op alpha : K -> S.
op hash  : (K*X) -> Y.
op proj  : (S*X*W) -> Y.
```

To define the projective property, we use an axiom.

```
axiom projective : forall (x:X, k:K, w:W),
  x \in ls => w \in (filter (iswit x) ws) =>
  proj (alpha k, x, w) = hash (k,x).
```

We now move on to define what it means for a projective hash family to be universal.

Definition 3.3. Let $\mathbf{H}$ be a projective hash family as in definition 3.2. Let $\epsilon \geq 0$ be a real number. $\mathbf{H}$ is *ϵ-universal* if

$$\Pr[H_k(x) = y \wedge \alpha(k) = s] \leq \epsilon \cdot \Pr[\alpha(k) = s],$$

for all $s \in S, x \in X \setminus L, y \in Y$ and $k \xleftarrow{r} K$. Further, $\mathbf{H}$ is said to be *ϵ-2-universal* if

$$\Pr[H_k(x) = y \wedge H_k(x') = y' \wedge \alpha(k) = s] \leq \epsilon \cdot \Pr[H_k(x') = y' \wedge \alpha(k) = s],$$

for all $s \in S, x' \in X, x \in X \setminus L \cup \{x'\}, y, y' \in Y$ and where $k \xleftarrow{r} K$.

In other words, we will say that $\mathbf{H}$ is ϵ-universal if given some $x \in X \setminus L$, we can guess the hash value of x with probability at most ϵ. Similarly, a projective hash family $\mathbf{H}$ is ϵ-2-universal if the value of $H_k(x)$, where $x \in X \setminus L$, can be guessed with probability at most ϵ even if we know the hash value of some other element x' in $X \setminus L$.

We now define what it means for a projective hash family to be *smooth*. Let $\mathbf{H}$ be a projective hash family as defined in Definition 3.2, and define two random variables $U(\mathbf{H})$ and $V(\mathbf{H})$ in the following way. We first sample

$$k \xleftarrow{\text{r}} K,\ x \xleftarrow{\text{r}} X \setminus L,\ y' \xleftarrow{\text{r}} Y.$$

We then set $U(\mathbf{H}) = (x, s, y')$ and $V(\mathbf{H}) = (x, s, y)$ where $s = \alpha(k)$ and $y = H_k(x)$.

Definition 3.4. For a real number $\epsilon \geq 0$, we say that a projective hash family $\mathbf{H}$ is ϵ-smooth if $U(\mathbf{H})$ and $V(\mathbf{H})$ are ϵ-close.

In other words, for $x \in X \setminus L$, it is hard to distinguish the actual hash value of x from some random element of Y. Informally, we can say that we know nothing, or at least very little, of the hash value of x when x is in $X \setminus L$.

As mentioned, EASYCRYPT is not designed to reason about run time, and we have not found a proper way of working with statistical distance. Thus, the universal and smooth properties need to be implemented differently that described above. This will be further discussed in Chapter 4.

3.3.4 Hash Proof Systems

In this section, we first define the notion of a *hash proof system*, before extending it to the notion of a *universal hash proof system*.

Let $\mathbf{M}$ be a subset membership problem as defined in §3.3.2. For each instance $\Lambda = \Lambda[X, L, W, R]$ of $\mathbf{M}$, a hash proof system $\mathbf{P}$ associates Λ with a projective hash family $\mathbf{H} = (H, K, X, L, Y, S, \alpha)$ for (X, L). Furthermore, a hash proof system $\mathbf{P}$ also provides some algorithms:

- A probabilistic algorithm that, on input $\Lambda \in I$, outputs some k, uniformly distributed over K.
- A deterministic algorithm that takes as input $\Lambda \in I$ and $k \in K$ and outputs $s \in S$ such that $\alpha(k) = s$.
- The private evaluation algorithm: a deterministic algorithm that, on input $\Lambda \in I$, $k \in K$ and $x \in X$, outputs some $y \in Y$ such that $H_k(x) = y$.
- The public evaluation algorithm: a deterministic algorithm which takes as input $\Lambda \in I$, $s \in S$ such that $\alpha(k) = s$ for some $k \in K$ and $x \in L$ with a witness $w \in W$ and outputs $y \in Y$ such that $H_k(x) = y$.

Before we can implement the hash proof system itself, we must equip our secret key type `K` with a full, uniform, lossless distribution.

```
op dK = { K distr | is_lossless dK
  /\ is_uniform dK /\ is_full dK } as dK_luf.
```

We are now ready to implement the hash proof system. We will implement all four algorithms described above.

```
module HPS = {
  proc kg() : K = {
    var k;
    k <$ dK;
    return k;
  }

  proc seval(k:K) : S = {
    var s;
    s <- alpha k;
    return s;
  }

  proc priveval(k:K, x:X) : Y = {
    var y;
    y <- hash(k,x);
    return y;
  }

  proc pubeval(s:S, x:X, w:W) : Y = {
    var y;
    y <- proj(s,x,w);
    return y;
  }
}.
```

As mentioned earlier, in this implementation we let our subset membership instance Λ be fixed, and thus we do not feed it as input to the algorithms in the hash proof system.

We now define what it means for a hash proof system **P** to be universal.

Definition 3.5. Let ϵ be a positive real number. Let **M** be a subset membership problem as defined in §3.3.2 and let **P** be a hash proof system for **M**. We will call **P** *ϵ-universal (-2-universal, -smooth)*, if there exists a negligible real number $\delta \geq 0$ such that for all instances $\Lambda = \Lambda[X, L, W, R]$, the projective hash family **H** that **P** associates with Λ is δ-close to an ϵ-universal (-2-universal, -smooth) projective hash family $\mathbf{H}^* = (\mathcal{H}^*, K^*, X, L, Y, S, \alpha^*)$. The projective hash family $\mathbf{H}^*$ is called the *idealization* of **H**.

Cramer and Shoup [3] make this definition as this is necessary to take into consideration when implementing the scheme "in real life". When we work with the scheme in EASYCRYPT, however, we will make the assumption that the hash families we work with are *ideal*. In other words, we assume that δ as defined in Definition 3.5 is zero. The reason is that EASYCRYPT, as mentioned, seems to be inadequate when it comes to working with quantities such as statistical distances.

Chapter 4

The Generic Cramer-Shoup Scheme

In this chapter we discuss the generic Cramer-Shoup scheme based on hash proof systems. We also sketch a proof for the IND-CCA security of this scheme. The definition of the scheme as well as the proof sketch will follow [3] with a couple of exceptions. EASYCRYPT is not designed to reason about complexity or statistical distances. Thus, we will assume that the approximation errors ι and ι' of the subset membership problem are both zero. We will also assume that the projective hash families we use are ideal (cf. definition 3.5). The main simplification, namely removing the extended hash proof system, will make the scheme IND-CPA secure rather than IND-CCA secure. Removing the extended hash proof system will not make the scheme itself much less complicated, but the verification of the security proof in EASYCRYPT will be a whole lot easier. By experience it is wise to start out as easy as possible when verifying a proof in EASYCRYPT, and build up to the full proof step by step.

4.1 The Original Scheme

In this section we describe how to construct an IND-CCA secure encryption scheme based on hash proof systems and subset membership problems.

Let $\mathbf{M}$ be a subset membership problem as defined in §3.3.2. We also need an ϵ-smooth hash proof system $\mathbf{P}$ and a $\hat{\epsilon}$-2-universal hash proof system $\hat{\mathbf{P}}$ for the subset membership problem $\mathbf{M}$.

Let $\Lambda = \Lambda[X, L, W, R] \in [I]$ be a fixed instance description of $\mathbf{M}$. Let $\mathbf{H} = (\mathcal{H}, K, X, Y, L, S, \alpha)$ be the projective hash family that $\mathbf{P}$ associates with Λ, and let $\hat{\mathbf{H}} = (\hat{\mathcal{H}}, \hat{K}, X \times Y, \hat{Y}, L \times Y, \hat{S}, \hat{\alpha})$ be the projective hash family $\hat{\mathbf{P}}$ associates with Λ. For the construction of the scheme, Y will be the message space.

Key generation. To generate a pair of public and secret keys, we do the following.

- Sample $k \xleftarrow{r} K, \hat{k} \xleftarrow{r} \hat{K}$.

- Compute $s \leftarrow \alpha(k), \hat{s} \leftarrow \hat{\alpha}(\hat{k})$.

Set $pk = (s, \hat{s})$ and $sk = (k, \hat{k})$.

Encryption. To encrypt a plaintext $m \in Y$, we do the following.

- Sample $x \xleftarrow{r} L$ along with a witness $w \in W$ for x, using the subset sampling algorithm for **M**.
- Compute $y \leftarrow H_k(x)$, using the public evaluation algorithm of **P** on input (s, x, w).
- Compute $e \leftarrow m + y$.
- Compute $\hat{y} \leftarrow \hat{H}_{\hat{k}}(x, e)$, using the public evaluation algorithm of $\hat{\mathbf{P}}$ on input $(\hat{s}, x, e, w)$.

The ciphertext is $c = (x, e, \hat{y})$.

Decryption. To decrypt a ciphertext $c = (x, e, \hat{y}) \in X \times Y \times \hat{Y}$, we do the following.

- Compute $\hat{y}' \leftarrow \hat{H}_{\hat{k}}(x, e)$, using the private evaluation algorithm of $\hat{\mathbf{P}}$ on input $(\hat{k}, x, e)$.
- If $\hat{y} \neq \hat{y}'$:
  - Output "reject" and stop the process.
- Else:
  - Compute $y' \leftarrow H_k(x)$, using the private evaluation algorithm of $\hat{P}$ on input (k, x).
  - Compute $m' \leftarrow e - y'$ and output the plaintext m'.

Correctness. Recall that for the correctness of the scheme to hold, we demand that $m = m'$. It is not difficult to see that this demand will hold if $\hat{y} = \hat{y}'$ and $y = y'$. Both of these equalities will hold, since the hash families **H** and $\hat{\mathbf{H}}$ are both projective, i.e. we can compute hash values for $x \in L$ using a witness for x and the projection key $s = \alpha(k)$. Thus, the correctness demand holds for this scheme.

The IND-CCA security of this scheme will be discussed in §4.3.

4.2 The Simplified Scheme

In this section, we describe the main simplification of the scheme, namely leaving out the extended hash proof system. This will reduce the security of the scheme from IND-CCA security to IND-CPA security. By making this simplification, the scheme itself will not become much less complicated, but the verification of the security proof in EASYCRYPT will be a whole lot easier. Experience shows that it is best to start out easy in EASYCRYPT, and hence, we will verify the IND-CPA security of this scheme before showing how to implement the IND-CCA security proof. We still assume that the approximation errors of the subset membership problem are zero and that the hash family **H** is ideal.

4.2.1 Construction

We now describe how we can construct an IND-CPA secure scheme by leaving out the extended hash proof system.

Let **M** be a subset membership problem as defined in §3.3.2. **M** specifies a distribution I of instance descriptions Λ. Let **P** be an ϵ-smooth hash proof system for **M**. Fix some $\Lambda \in [I]$, and let **H** = $(\mathcal{H}, K, X, L, Y, S, \alpha)$ be the projective hash family that **P** associates with Λ. Let Y be the message space.

Key generation. Generating a pair of keys is done by first sampling $k \xleftarrow{r} K$ and then setting $s \leftarrow \alpha(k)$. The public encryption key pk is s, and the secret decryption key sk is k.

Encryption. To encrypt some message $m \in Y$, we do the following.

- Sample $x \in L$ along with a witness w.
- Compute $y = H_k(x)$ using the public evaluation algorithm of **P** on inputs s, x, w.
- Compute $e \leftarrow m + y$.

The ciphertext is $c = (x, e)$.

Decryption. To decrypt a ciphertext $c = (x, e) \in X \times Y$, we do the following.

- Compute $y' = H_k(x)$ using the private evaluation algorithm of **P** on inputs k, x.
- Compute $m' \leftarrow e - y'$.

Correctness. Recall that for the scheme to be correct, we demand that $m = m'$. It is not difficult to see that as long as $y = y'$, we will have $m = m'$. As the encryption algorithm samples x from the subset L, the projective property of **H** will make sure that it is possible and easy to calculate the hash value of x, given w and s. Thus, we will have that $y = y'$. Hence, the correctness holds for this scheme.

We now describe how we implement the simplified scheme in EASYCRYPT. We will use the implementations of the subset membership problem and the hash proof system described in §3.3.2 and §3.3.4, respectively.

We start by defining the types for the keys, plaintexts and ciphertexts using type aliases.

```
type pkey = S.
type skey = K.
type plaintext = Y.
type ciphertext = X * Y.
```

In the encryption and decryption algorithms, we need to add and subtract elements of type `Y`. EASYCRYPT however, does not have any built-in addition and subtraction operators for user defined abstract types, so we need to implement operators taking elements from type `Y` to type `int` and back again. We also list a couple of axioms defining how these operators should behave.

```
op toint : Y -> int.
op toY   : int -> Y.
axiom y1 : forall (y:Y),   toY (toint y) = y.
axiom y2 : forall (y:int), toint (toY y) = y.
```

We now define a module called `Genscheme`, which is of type `Scheme` described in §3.1.1. This module will model the cryptosystem described above. In this module, we will use the algorithms of the subset membership problem defined in §3.3.2 and the hash proof system defined in §3.3.4.

```
module Genscheme : Scheme = {
  proc keygen() : pkey * skey = {
    var k, s, pk, sk;
    k  <- HPS.kg();
    s  <- HPS.seval(k);
    pk <- s; sk <- k;
    return (pk, sk);
  }

  proc encrypt(pk,m) : ciphertext = {
    var x, w, e, y, c;
    (x,w) <- Sampling.fromL();
      y   <- HPS.pubeval(pk, x, w);
      e   <- toY (toint m + toint y);
      c   <- (x, e);
    return c;
  }

  proc decrypt(sk,c) : plaintext = {
    var y, m';
    y  <- HPS.priveval(sk, c.`1);
    m' <- toY (toint c.`2 - toint y);
    return m';
  }
}.
```

Proving the correctness of this scheme is straightforward, using the module and lemma described in §3.1.1.

4.2.2 Security of the Simplified Scheme

The encryption scheme described in §4.2.1 is secure against a chosen ciphertext attack under the assumption that **M** is a hard subset membership problem. We state this more formally in the following theorem.

Theorem 4.1. *Let $\mathcal{A}$ be an adversary carrying out an IND-CPA attack against the scheme described in §4.2.1. Then there exists an adversary $\mathcal{B}$ against the subset membership problem **M** such that*

$$\mathit{AdvCPA}(\mathcal{A}) \leq \mathit{AdvSMP}(\mathcal{B}) + \epsilon,$$

and the run time of $\mathcal{B}$ is the same as the run time of $\mathcal{A}$.

In Theorem 4.1, ϵ is the statistical distance between the distribution $(x, s, H_k(x))$ and the distribution (x, s, y') where $x \in X \setminus L$, $s = \alpha(k)$ and y' is chosen randomly from Y.

We now sketch the proof of Theorem 4.1. Let **H** be a fixed projective hash family as described in §4.2.1. We will structure the proof using a sequence of attack games. We let G_0 be the original attack game against our scheme, as defined in §3.1.2. Recall that the challenger samples a random bit $b \in \{0, 1\}$, encrypts m_b and the adversary outputs a bit b', i.e. guessing which message he believes was encrypted. Throughout the proof sketch, we let E_i denote the event that $b = b'$ (i.e. the adversary wins) in game G_i. Thus, the adversary's advantage in the original attack, G_0, is

$$\text{AdvCPA}(\mathcal{A}) = |\Pr[E_0] - 1/2|.$$

We now describe a simulator playing the CPA game against our adversary. The simulator takes as input some instance description Λ and some $x^* \in X$. The simulator uses the key generation algorithm described in the scheme. When receiving two messages m_0 and m_1 from the adversary, the simulator samples a random bit b, computes $y^* = H_k(x^*)$ and $e^* = m_b + y^*$. The simulator sends $c^* = (x^*, e^*)$ to the adversary, who outputs a bit b'.

In G_1, the simulator is given (Λ, x^*) with $x^* \in L$. The transition from G_0 to G_1 is just a bridging step, so in G_1, the simulator perfectly simulates the original attack. In G_2, the simulator is given (Λ, x^*) with $x^* \in X \setminus L$. Distinguishing between G_1 and G_2 is essentially the same as deciding whether x is in L or $X \setminus L$, i.e. solving the underlying subset membership problem. Thus, we define the advantage in solving the subset membership problem as

$$\text{AdvSMP}(\mathcal{B}) = |\Pr[E_2] - \Pr[E_1]|.$$

In G_3, we modify the simulator. Now, instead of computing $y^* = H_k(x^*)$, the simulator sets $y^* = y'$, where y' is chosen uniformly at random from Y. Because of the smoothness property of **H**, we have

$$|\Pr[E_3] - \Pr[E_2]| \leq \epsilon.$$

Also, in G_3, the adversary's output b' is completely independent of the hidden bit b. Thus, we have

$$\Pr[E_3] = 1/2.$$

Combining the above relations, we see that

$$\text{AdvCPA}(\mathcal{A}) \leq \text{AdvSMP}(\mathcal{B}) + \epsilon,$$

which is the relation we want to verify in EASYCRYPT.

As mentioned earlier, we cannot use EASYCRYPT to reason about complexity or statistical distances, meaning we cannot include the inequality

$$|\Pr[E_3] - \Pr[E_2]| \leq \epsilon$$

in EASYCRYPT. Instead, we will define a smoothness adversary whose task is to distinguish between tuples of the form $(x, s, H_k(x))$ and tuples of the form (x, s, y) where y is chosen at random from Y. We then replace ϵ by this adversary's distinguishing advantage.

Thus, the relation we want to prove in EASYCRYPT is the following.

```
`|Pr[CPA(Genscheme,A).main() @ &m : res]-1%r/2%r| <=
  `|Pr[SMP1(SMPadv(A)).main() @ &m : res] -
    Pr[SMP0(SMPadv(A)).main() @ &m : res]| +
  `|Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] -
    Pr[Smooth0(SmoothAdv(A)).main() @ &m : res]|.
```

The left hand side of the inequality denotes the adversary's advantage in the original CPA attack (which we call G_0 in the proof sketch above). The first absolute difference on the right hand side denotes the distinguishing advantage in the subset membership problem, and the second absolute difference denotes the smoothness distinguishing advantage.

By defining the advantage in solving the subset membership problem the way we do in EASYCRYPT, the adversary $\mathcal{B}$ trivially gets the same run time as $\mathcal{A}$. The reason for this is that we simply use the adversary $\mathcal{A}$ to construct $\mathcal{B}$, meaning the adversaries attacking the encryption scheme and solving the subset membership problem will essentially be the same. The exact construction of the SMP adversary will be discussed below. The same goes for the smoothness advantage.

We now discuss how we structure and implement the proof in EASYCRYPT. Most of the modules used to model the games in this proof will be defined inside a *section*. The exception is the general module used to define a CPA attack which we have defined in §3.1.2. This way, we can define an adversary in the beginning of the section and use this throughout the entire proof. Note that every lemma we list needs to be proved in EASYCRYPT. As mentioned, these proofs will be omitted in the text. The general security proof layout will be as follows:

```
section Security.

declare module A : CPAadversary{CPA}.
axiom Ag_ll : islossless A.guess.
axiom Ac_ll : islossless A.choose.

(* Games, lemmas and proofs *)
```

```
...

end section Security.
```

We now have an adversary called `A` of type `CPAadversary` as defined in §3.1.2. The axioms `Ag_ll` and `Ac_ll` make sure that the `guess` and `choose` procedures of the adversary terminate.

We now define modules modeling the simulator described in the pen-and-paper proof above. Note that modules (and lemmas) defined inside a section are defined as "local".

```
local module Game1(A:CPAadversary) = {
  proc main() : bool = {
    var x, w, pk, sk, m0, m1, b, b', y, e, c;
     (x, w)   <- Sampling.fromL;
    (pk, sk)  <- Genscheme.keygen();
    (m0, m1)  <- A.choose(pk);
       b      <$ {0,1};
       y      <- HPS.priveval(sk, x);
       e      <- toY (toint (b?m1:m0) + toint y);
       c      <- (x,e);
       b'     <- A.guess(c);
     return (b = b');
  }
}.

local module Game2(A:CPAadversary) = {
  proc main() : bool = {
    var x, pk, sk, m0, m1, b, b', y, e, c;
       x      <- Sampling.fromXnotL;
    (pk, sk)  <- Genscheme.keygen();
    (m0, m1)  <- A.choose(pk);
       b      <$ {0,1};
       y      <- HPS.priveval(sk, x);
       e      <- toY (toint (b?m1:m0) + toint y);
       c      <- (x,e);
       b'     <- A.guess(c);
     return (b = b');
  }
}.
```

As in the pen-and-paper proof, the difference between the two modules `Game1` and `Game2`, is that x is sampled from L and $X \setminus L$, respectively.

As we omit the approximation errors of the sampling algorithms of the subset membership problem, the game `Game1` simulates the CPA attack perfectly. Thus, the first thing we prove is the following equivalence.

```
local equiv CPA_Game1 :
  CPA(Genscheme, A).main ~ Game1(A).main :
  ={glob A} ==> ={res}.
```

The notation `={glob A}` means that the global variables of the adversary (if any) is the same in both attacks, and thus makes sure the adversary is the same.

After proving this equivalence, it is straightforward to transform the statement to a regular probability expression:

```
local lemma CPA_Game1_pr &m :
  Pr[CPA(Genscheme,A).main() @ &m : res] =
  Pr[Game1(A).main() @ &m : res]
  by byequiv(CPA_Game1).
```

The next relation we prove is that distinguishing between the modules `Game1` and `Game2` is equivalent to solving the underlying SMP, i.e. deciding whether $x \in L$ or $x \in X \setminus L$. For this purpose, we define an SMP adversary guessing that x is either in L or in $X \setminus L$. Furthermore, we make a reduction from a CPA attack to an SMP attack. To make the SMP adversary easier to work with in EASYCRYPT, we split it into two modules.

```
module type SMPadversary = {
  proc guess(x:X) : bool
}.

module SMP1(A:SMPadversary) = {
  proc main() : bool = {
    var x, w, b;
    (x,w) <- Sampling.fromL(); b   <- A.guess(x);
    return b;
  }
}.

module SMP0(A:SMPadversary) = {
  proc main() : bool = {
    var x, b;
    x <- Sampling.fromXnotL(); b <- A.guess(x);
    return b;
  }
}.
```

For the reduction, we define a module with a procedure taking an `x` as input and then use this `x` in the attack.

```
module SMPadv(A:CPAadversary) = {
  proc guess(x:X) : bool = {
    var b, b', m0, m1, pk, sk, y;
    (pk, sk) <- Genscheme.keygen();
    (m0, m1) <- A.choose(pk);
    y  <- hash(sk,x);
    b  <$ {0,1};
    b' <- A.guess(x, toY (toint (b?m1:m0) + toint y));
    return (b = b');
  }
}.
```

We now want to prove that

```
`|Pr[Game1(A).main() @ &m : res] -
  Pr[Game2(A).main() @ &m : res]| =
`|Pr[SMP1(SMPadv(A)).main() @ &m : res] -
  Pr[SMP0(SMPadv(A)).main() @ &m : res]|.
```

We do this by proving the following equivalences.

```
local equiv Game1_SMP1 :
  Game1(A).main ~ SMP1(SMPadv(A)).main :
  ={glob A} ==> ={res}.

local equiv Game2_SMP0 :
  Game2(A).main ~ SMP0(SMPadv(A)).main :
  ={glob A} ==> ={res}.
```

By transforming these equivalences to probability expressions, the above equality follows trivially.

We now define the module `Game3`. Here, instead of computing the value `y` as `hash(k,x)`, we sample `y` uniformly at random from `ys`. The set `ys` is defined similarly to the set `xs` and contains every element of type `Y`.

```
local module Game3(A:CPAadversary) = {
  proc main() : bool = {
    var x, pk, sk, m0, m1, b, b', y, e, c;
       x      <- Sampling.fromXnotL;
    (pk, sk)  <- Genscheme.keygen();
    (m0, m1)  <- A.choose(pk);
       b      <$ {0,1};
       y      <$ duniform (elems ys);
       e      <- toY (toint (b?m1:m0) + toint y);
       c      <- (x,e);
       b'     <- A.guess(c);
     return (b = b');
  }
}.
```

The transition between `Game2` and `Game3` is based on indistinguishability. The idea is to prove that an adversary's ability to distinguish between these games is bounded by his ability to distinguish between tuples of the form $(x, s, H_k(x))$ and tuples of the form (x, s, y) where y is chosen randomly and $x \in X \setminus L$. To do this, we give the CPA adversary access to a smoothness adversary and make a reduction from a CPA attack to a smoothness attack. We define a smoothness adversary using a module type with a guessing procedure. This adversary has the ability to guess either `(x, s, hash(k,x))` or `(x, s, y)`. We split this guessing into two modules to make it easier to work with when we prove the equivalences with `Game2` and `Game3`.

```
module type SmoothAdversary = {
  proc guess(x:X, s:S, y:Y) : bool
}.

module Smooth1(A:SmoothAdversary) = {
  proc main() : bool = {
    var b, k, s, x;
    x <- Sampling.fromXnotL();
    k <- HPS.kg();
    s <- HPS.seval(k);
    b <- A.guess(x, s, hash(k,x));
    return b;
  }
}.

module Smooth0(A:SmoothAdversary) = {
  proc main() : bool = {
    var b, k, s, x, y;
    x <- Sampling.fromXnotL();
    k <- HPS.kg();
    s <- HPS.seval(k);
    y <- Sampling.fromY();
    b <- A.guess(x, s, y);
    return b;
  }
}.
```

For the reduction, we define a module with a procedure taking `(x, s, y)` as input, and uses these values in the `choose` and `guess` procedures of the attack.

```
module SmoothAdv(A:CPAadversary) = {
  proc guess(x:S, s:S, y:Y) : bool = {
    var m0, m1, b, b';
    (m0, m1) <- A.choose(s);
    b  <$ {0,1};
    b' <- A.guess(x, toY (toint (b?m0:m1) + toint y));
    return (b = b');
  }
}.
```

To prove that distinguishing between `Game2` and `Game3` is equivalent to distinguishing between `(x, s, hash(k,x))` and `(x, s, y)`, we prove the following two equivalences.

```
local equiv Game2_Smooth1 :
  Game2(A).main ~ Smooth1(SmoothAdv(A)).main :
  ={glob A} ==> ={res}.

local equiv Game3_Smooth0 :
  Exp2G3(A).main ~ Smooth0(SmoothAdv(A)).main :
  ={glob A} ==> ={res}.
```

These equivalences can be transformed to probability expressions in the usual way, using the **byequiv** tactic.

Finally, we need to prove that when playing against `Game3`, the adversary wins with probability $1/2$. To do this we first define a module we call `Game3indep`.

```
local module Game3indep(A:CPAadversary) = {
  proc main() : bool = {
    var x, pk, sk, m0, m1, b, b', y, e, c;
       x      <- Sampling.fromXnotL;
    (pk, sk)  <- Genscheme.keygen();
    (m0, m1)  <- A.choose(pk);
       y      <$ duniform (elems ys);
       e      <- toY (toint y);
       c      <- (x,e);
       b'     <- A.guess(c);
       b      <$ {0,1};
     return (b = b');
  }
}.
```

In this game, it is easy to see that the adversary's output `b'` is completely independent of the bit `b`, and even that the ciphertext is independent of both `m0` and `m1`. Thus, it is straightforward to prove the following lemma.

```
local lemma Game3indep_half :
  phoare[Game3indep(A).main : true ==> res] =
  (1%r/2%r).
```

Now, the only thing left is to prove the equivalence between `Game3` and `Game3indep`.

```
local equiv :
  Game3(A).main ~ Game3indep(A).main :
  ={glob A} ==> ={res}.
```

After transforming all the above equivalences to a probability statements, we have the following probabilities.

```
(* 1 *)
Pr[CPA(Genscheme, A).main() @ &m : res] =
Pr[Game1(A).main() @ &m : res].

(* 2 *)
Pr[Game1(A).main() @ &m : res] =
Pr[SMP1(SMPadv(A)).main() @ &m : res].

(* 3 *)
Pr[Game2(A).main() @ &m : res] =
Pr[SMP0(SMPadv(A)).main() @ &m : res].
```

```
(* 4 *)
Pr[Game2(A).main() @ &m : res] =
Pr[Smooth1(SmoothAdv(A)).main() @ &m : res].

(* 5 *)
Pr[Game3(A).main() @ &m : res] =
Pr[Smooth0(SmoothAdv(A)).main() @ &m : res].

(* 6 *)
Pr[Game3(A).main() @ &m : res] =
Pr[Game3indep(A).main() @ &m : res].

(* 7 *)
Pr[Game3indep(A).main() @ &m : res] = 1%r/2%r.
```

Using the above relations, as well as the triangle inequality for absolute differences, we can prove our final lemma.

```
local lemma secure &m :
`|Pr[CPA(Genscheme,A).main() @ &m : res]-1%r/2%r| <=
  `|Pr[SMP1(SMPadv(A)).main() @ &m : res] -
    Pr[SMP0(SMPadv(A)).main() @ &m : res]| +
  `|Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] -
    Pr[Smooth0(SmoothAdv(A)).main() @ &m : res]|.
```

This proves, as discussed in the beginning of the section, that the adversary's advantage in winning in a CPA attack against our scheme, is bounded by the advantage in distinguishing between elements from X and elements from $X \setminus L$, as well as the advantage in distinguishing hash values of x from random elements of Y when $x \notin L$.

4.3 Security of the Original Scheme

In this section, we discuss the security of the encryption scheme defined in §4.1. This scheme is secure against an IND-CCA attack under the assumption that **M** is a hard subset membership problem. We formalize this in the following theorem.

Theorem 4.2. *Let $\mathcal{A}$ be an adversary carrying out an IND-CCA attack against the scheme described in §4.1. Then there exists an adversary $\mathcal{B}$ against the subset membership problem **M** such that*

$$AdvCCA(\mathcal{A}) \leq AdvSMP(\mathcal{B}) + \epsilon + Q \cdot \hat{\epsilon},$$

and the run time of $\mathcal{B}$ is the same as the run time of $\mathcal{A}$.

In Theorem 4.2, ϵ comes from the ϵ-smooth property of **H**, Q is the maximal allowed number of queries to the decryption oracle and $\hat{\epsilon}$ comes from the $\hat{\epsilon}$-2-universal property of $\hat{\mathbf{H}}$.

We now sketch the proof of Theorem 4.2. The full proof can be found in [3].

Let **H** and $\hat{\mathbf{H}}$ be two fixed projective hash families as described in §4.1.

We structure this proof sketch as a sequence of games, similar to what we did for the simplified scheme in §4.2.2. We let G_0 be the original CCA attack against our scheme, as defined in §3.1.2. Again, we will let E_i denote the event that the adversary wins (i.e. that $b = b'$) in G_i. Thus, the adversary's advantage in the original attack is

$$\text{AdvCCA}(\mathcal{A}) = |\Pr[E_0] - 1/2|.$$

We now describe a simulator playing the CCA game against an adversary. The simulator takes as input some instance description Λ along with some $x^* \in X$. During the interaction between the simulator and the adversary, the key generation phase and both probing phases run as described in §3.1.2, with the simulator running the usual decryption algorithm using sk.

In the target selection phase, the simulator receives m_0 and m_1 from the adversary and samples a random bit b. The simulator then computes $y^* = H_k(x^*)$ using the private evaluation algorithm of $\mathbf{P}$, where x^* is the input given to the simulator. Then, it computes $e^* = m_b + y^*$, before computing $\hat{y}^* = \hat{H}_{\hat{k}}(x^*, e^*)$ using the private evaluation algorithm for $\hat{\mathbf{P}}$. The simulator then sends the target ciphertext $c^* = (x^*, e^*, \hat{y}^*)$ to the adversary.

In the guessing phase, the adversary outputs a bit b'.

We let G_1 be the game where the simulator is given (Λ, x^*) with $x^* \in L$. This can be considered as a bridging step from G_0 to G_1. In other words, the simulator in this case perfectly simulates the original attack.

Let G_2 be the game where the simulator is given (Λ, x^*) with $x \in X \setminus L$. Distinguishing between G_1 and G_2 is again essentially the same as solving the subset membership problem (deciding whether $x \in L$ or $x \in X \setminus L$). We define the advantage in solving the subset membership problem as

$$\text{AdvSMP}(\mathcal{B}) = |\Pr[E_2] - \Pr[E_1]|.$$

We now make a transition from G_2 to a game G_3, based on a failure event. In G_3, we modify the decryption oracle. Now, in addition to rejecting a ciphertext $c = (x, e, \hat{y})$ if $\hat{H}_{\hat{k}}(x, e) \neq \hat{y}$, the oracle also rejects a ciphertext if $x \notin L$. Let F be the event that the decryption oracle rejects a ciphertext where $x \notin L$, but $\hat{H}_{\hat{k}}(x, e) = \hat{y}$. We can prove that

$$|\Pr[E_3] - \Pr[E_2]| \leq \Pr[F],$$

and that

$$\Pr[F] \leq Q \cdot \hat{\epsilon},$$

where Q is the maximal number of queries that the adversary is allowed to make to the decryption oracle.

We now make a transition from G_3 to a game G_4, based on indistinguishability. In G_4, we modify the encryption oracle. Instead of computing $y^* = H_k(x^*)$, the encryption oracle now sets $y^* = y'$, where y' is sampled at random from Y. Using the fact that the hash family $\mathbf{H}$ is ϵ-smooth, we can prove that

$$|\Pr[E_4] - \Pr[E_3]| \leq \epsilon.$$

In G_4, the adversary's output b' is independent of the hidden bit b, so

$$\Pr[E_4] = 1/2.$$

Combining the relations above, we can see that

$$\text{AdvCCA} \leq \text{AdvSMP}(\mathcal{B}) + \epsilon + Q \cdot \hat{\epsilon},$$

which is the relation we want to prove in EASYCRYPT. As with the simplified scheme, we will define a smoothness adversary and replace ϵ by this adversary's advantage in distinguishing between correct hash values and random elements in Y. We also replace the term $Q \cdot \hat{\epsilon}$ by the explicit probability that the adversary has been able to guess a hash value of some $x \notin L$. Thus, the relation we want to prove in EASYCRYPT is

```
`|Pr[CCA(Genscheme, A).main() @ &m : res]-1%r/2%r| <=
`|Pr[SMP1(SMPadv(A)).main() @ &m : res] -
  Pr[SMP0(SMPadv(A)).main() @ &m : res]| +
`|Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] -
  Pr[Smooth0(SmoothAdv(A)).main() @ &m : res]| +
`|Pr[Game3(A).main() @ &m :
  exists c, c \in Game3.log => !c.`1 \in ls /\
  c.`3 = hash_(Game3.sk.`2, c.`1, c.`2)]|.
```

As seen above, we make sure that the run time of $\mathcal{B}$ is the same as the run time of $\mathcal{A}$ by constructing $\mathcal{B}$ from $\mathcal{A}$, as we did in §4.2.2.

Before we discuss how to implement the security proof in EASYCRYPT, we discuss how to implement the encryption scheme. We will not discuss every detail of this implementation but rather focus on the main differences between the implementation of this scheme and the simplified scheme. In addition to the decryption oracle, there are two main differences. The first is that we need an extended hash proof system in addition to the hash proof system defined in §3.3.4. The second is that we need to define a failure event where the adversary guesses the hash value of some $x \notin L$. Both of these additions will be discussed in detail below.

We first implement the extended hash proof system which associates the projective hash family $\hat{\mathbf{H}} = (\hat{\mathcal{H}}, \hat{K}, X \times Y, L \times Y, \hat{Y}, \hat{S}, \hat{\alpha})$ with our subset membership problem. The differences between this hash proof system and the previous one, is that the keys are sampled from a (possibly) different distribution, the hashing and projection algorithms take an $e \in Y$ as input in addition to x and the output distribution of the hash functions is (possibly) different. We first define additional types:

```
type K_, S_, Y_.
```

We must equip the type `K_` with a full, uniform, lossless distribution, which we will call `dK_`. This distribution is defined exactly as the distribution `dK` in §3.3.4. We also need additional `alpha`, `hash` and `proj` operators to use in the algorithms of the extended hash proof system.

```
op alpha_ : K_ -> S_.
op hash_  : (K_ * X * Y) -> Y_.
op proj_  : (S_ * X * Y * W) -> Y_.
```

We define the projective property for the extended hash proof system as we did in §3.3.3.

Using the new types, the distribution `dK_` and the above operators, we can define our extended hash proof system in EASYCRYPT.

```
module HPS_Ext = {
  proc kg() : K_ = {
    var k;
    k <$ dK_;
    return k;
  }

  proc seval(k:K_) : S_ = {
    var s;
    s <- alpha_ k;
    return s;
  }

  proc priveval(k:K, x:X, e:Y) : Y_ = {
    var y;
    y <- hash_ (k, x, e);
    return y;
  }

  proc pubeval(s:S, x:X, e:Y, w:W) : Y_ = {
    var y;
    y <- proj_ (s, x, e, w);
    return y;
  }
}.
```

When implementing the encryption scheme itself, we must redefine the types of the keys, plaintexts and ciphertexts.

```
type pkey = S * S_. type skey = K * K_.
type plaintext = Y. type ciphertext = X * Y * Y_.
```

To implement the scheme itself, we again define a module we call `Genscheme`, which is of type `Scheme`.

```
module Genscheme : Scheme = {
  proc keygen() : pkey * skey = {
    var k, k_, s, s_, pk, sk;
    k <- HPS.kg(); k_ <- HPS_Ext.kg();
    s <- HPS.seval(k); s_ <- HPS_Ext.seval(k_);
    pk <- (s, s_); sk <- (k, k_);
    return (pk, sk);
  }

  proc encrypt(pk:pkey, m:plaintext) : ciphertext = {
    var x, w, y, e, y_, c;
    (x, w) <- Sampling.fromL();
```

```
      y    <- HPS.pubeval(pk.`1, x, w);
      e    <- toY (toint m + toint y);
      y_   <- HPS_Ext.pubeval(pk.`2, x, e, w);
      c    <- (x, e, y_);
    return c;
  }

  proc decrypt(sk:skey,c:ciphertext) :
    plaintext option = {
    var y, y_', m;
    y_' <- HPS_Ext.priveval(sk.`2, c.`1, c.`2);
    if (c.`3 = y_') {
      y <- HPS.priveval(sk.`1, c.`1);
      m <- Some (toY (toint c.`2 - toint y));
    } else {
      m <- None;
    }
    return m;
  }
}.
```

As we see, the main difference between this scheme and the simplified scheme is that here, we check whether or not a given ciphertext is valid. This is done by checking if the third element of the ciphertext (`c.`3 = y_`) equals the hash value of the first two elements of the ciphertext (`c.`1 = x` and `c.`2 = e`) under the secret hash key `k_`. Proving the correctness for this scheme is done exactly as for the CPA secure scheme, using the correctness module defined in §3.1.1.

We now discuss how to implement the security proof for this encryption scheme in EASYCRYPT. As with the simplified scheme, we place all the games and transitions inside a section. We start the section by declaring an adversary of type `CCAadversary`, which is not allowed to work in the memory space of the module `CCA`.

```
declare module A : CCAadversary{CCA}.
```

We need to make sure that the adversary's procedures `A.choose` and `A.guess` terminate, i.e. we must state axioms saying that these procedures are lossless. In this case, we must make an addition to these axioms compared to the case of the simplified scheme. Both adversary procedures have access to the decryption oracle, so we will say both `A.choose` and `A.guess` are lossless, only if the decryption oracle is lossless, i.e. terminates.

```
axiom Ag_ll : forall (O <: CCAoracle{A}),
  islossless O.decrypt => islossless A(O).guess.
axiom Ac_ll : forall (O <: CCAoracle{A}),
  islossless O.decrypt => islossless A(O).choose.
```

The notation `forall (O <: CCAoracle{A})` means for all oracles `O` of the module type `CCAoracle` not working in the memory space of `A`. In other words, we make sure that the oracle does not have access to the global variables of `A` and does not know how the adversary really works.

We start the proof by defining modules modeling the two experiments as we did in §4.2.2, with the exception that here, we must also add the decryption oracle.

```
local module Game1(A:CCAadversary) = {
  var log   : ciphertext list
  var cstar : ciphertext option
  var sk    : skey

  module O = {
    proc decrypt(c:ciphertext) : plaintext option = {
     var m : plaintext option;
     var y, y_';
     if (size log < qD && Some c <> cstar) {
      log <- c :: log;
      y_' <- HPS_Ext.priveval(sk.`2, c.`1, c.`2);
      if (y_' = c.`3) {
        y <- HPS.priveval(sk.`1, c.`1);
        m <- Some (toY (toint c.`2 - toint y));
      } else m <- None;
     } else m <- None;
     return m;
    }
  }

  module A = A(O)

  proc main() : bool = {
    var xstar, w, m0, m1, b, b';
    var ystar, y_', estar, c, pk;
    log <- [];
    cstar <- None;
    (pk, sk) <- Genscheme.keygen();
    (xstar, w) <- Sampling.fromL();
    (m0, m1) <- A.choose(pk);
    b <$ {0,1};
    ystar <- HPS.priveval(sk, c.`1);
    y_' <- HPS_Ext.priveval(sk.`2, xstar, estar);
    c <- (xstar, estar, y_');
    cstar <- Some c;
    b' <- A.guess(c);
    return (b = b');
  }
}.
```

To make sure that the adversary procedures `A.choose` and `A.guess` terminate, we can prove that the decryption oracle is lossless by proving the following lemma.

```
local lemma Game1_O_ll :
islossless Game1(A).O.decrypt.
```

As we still omit the approximation errors of the subset membership problem, this experiment is equivalent to the original CCA attack. We state this in the following lemma.

```
local equiv CCA_Game1 :
  CCA(Genscheme, A).main ~ Game1(A).main :
  ={glob A} ==> ={res}.
```

The module in which x is sampled from $X \setminus L$ instead of L is defined exactly as `Game1`, with the exception that we replace `(xstar, w) <- Sampling.fromL();` with `xstar <- Sampling.fromXnotL();` in the main procedure. This module will be referred to as `Game2`.

We can, similarly to the security proof for the simplified scheme, prove that the advantage in distinguishing between `Game1` and `Game2` is equivalent to solving the subset membership problem. We do this by defining SMP modules and make a reduction from a CCA attack to an SMP attack, similar to what we did in §4.2.2. We then prove that

```
Pr[Game1(A).main() @ &m : res] =
Pr[SMP1(SMPadv(A)).main() @ &m : res]
```

and

```
Pr[Game2(A).main() @ &m : res] =
Pr[SMP0(SMPadv(A)).main() @ &m : res].
```

The main difference between the proof of the simplified scheme and this scheme, is that we need a module where we modify the decryption oracle. In this proof, we will call this module `Game3`. In G_3 in the pen-and-paper proof sketch, we make a change to the decryption oracle, telling it to reject a ciphertext $(x, e, \hat{y}')$ if $x \notin L$, in addition to when $\hat{y}' \neq H_k(x, e)$. We will only describe the change in the decryption oracle, as well as the global variables, as the rest of the module will be (almost) equal to `Game2`. The definition of the decryption oracle and the global variables in `Game3` is as follows.

```
module Game3(A:CCAadversary) = {
var log   : ciphertext list
var cstar : ciphertext option
var sk    : skey
var bad   : bool

module O = {
 proc decrypt(c:ciphertext) : plaintext option = {
  var m : plaintext option;
  var y, y_';
  if (size log < qD && Some c <> cstar) {
   log <- c :: log;
   y_' <- HPS_Ext.priveval(sk.`2, c.`1, c.`2);
   if (y_' = c.`3) {
    y <- HPS.priveval(sk.`1, c.`1);
    m <- Some (toY (toint c.`2 - toint y));
    if (!c.`1 \in ls
     /\ c.`3 = HPS_Ext.priveval(sk.`2, c.`1, c.`2))
```

```
        {bad <- true;}
    } else m <- None;
   } else m <- None;
   return m;
  }
}

··· (* Rest of the module, similar to Game2 *)

}.
```

In the module `Game3`, we modify the decryption oracle in a different way than in the pen-and-paper proof sketch. Here, we do not reject a ciphertext $(x, e, \hat{y}')$, if $x \notin L$. Instead, we accept any ciphertext as long as $\hat{y} = H_k(x, e)$, but define a boolean we call `bad` and set this to be true if the first component of the ciphertext is not in L. The rest of this module equals `Game2`, with the exception that we define `bad` to be false in the beginning of the main procedure.

Recall that in the pen-and-paper proof sketch, we proved that the absolute difference between the adversary's advantage in G_2 and G_3 is bounded by $Q \cdot \hat{\epsilon}$. As we have discussed, EASYCRYPT is not designed to reason about complexity. Thus, we cannot prove that any probability is less than this bound in EASYCRYPT. Instead we implicitly include the $\hat{\epsilon}$-2-universal property in the security proof by working directly with the probability that the adversary has been able to guess the hash value of some $x \notin L$.

To achieve this, we need to break the statement down into several smaller lemmas and proceed step by step. We start by proving that if the variable `bad` remains false, `Game2` and `Game3` are equivalent. We first prove that if `bad` remains false, the decryption oracles are equivalent, by proving the following lemma.

```
local equiv Game2_Game3_decrypt_failure :
  Game2(A).O.decrypt ~ Game3(A).O.decrypt :
  !Game3.bad{1} /\ Game2.log{1} = Game3.log{2}
  /\ Game2.cstar{1} = Game3.cstar{2}
  /\ Game2.sk{1} = Game3.sk{2}
  /\ Some c{1} = Some c{2} ==>
  !Game3.bad{1} => Game2.log{1} = Game3.log{2}
  /\ Game2.cstar{1} = Game3.cstar{2}
  /\ Game2.sk{1} = Game3.sk{2} /\ ={res}.
```

Recall that the notation `={res}` means that the result is the same after executing both procedures. In this case it means that for any ciphertext given to the decryption oracle, the resulting plaintext should be the same in the two games. To make sure this is true, we need to assume that the global variables of the two modules and the ciphertexts given to the decryption oracles are the same, as we have done in the above equivalence. After this equivalence is proved, we can prove that the main procedures also are equivalent, if the `bad` variable remains false.

```
local equiv Game2_Game3_main_bad :
  Game2(A).main ~ Game3(A).main :
  ={glob A} ==> !Game3.bad{2} => ={res}.
```

We are now ready to prove that

```
`|Pr[Game2(A).main() @ &m : res] -
  Pr[Game3(A).main() @ &m : res]|
```

is bounded above by the probability of the bad event being set to true. We cannot, however, prove this directly. To make full use of EASYCRYPT's built-in proof system, we first need to prove the following lemma.

```
local lemma Game2_Game3_main_bad_pr &m :
  Pr[Game2(A).main() @ &m : res] <=
  Pr[Game3(A).main() @ &m : res] +
  Pr[Game3(A).main() @ &m : Game3.bad].
```

The reason that we need this intermediate step is that we want to transform the equivalence `Game2_Game3_main_bad` to a probability expression using the **`byequiv`** tactic. For this tactic to work, we cannot have two different procedures on one side of the inequality sign. After proving this, however, it is straightforward to prove that the difference of the adversary winning the two games is less than our equal to the bad event being set to `true`.

```
local lemma Game2_Game3_main_bad_pr2 &m :
  Pr[Game2(A).main() @ &m : res] -
  Pr[Game3(A).main() @ &m : res] <=
  Pr[Game3(A).main() @ &m : Game3.bad].
```

Now, we need to prove that the *absolute* difference between the adversary winning in `Game2` and `Game3` is bounded by the probability that the bad event is set to `true`.

```
`|Pr[Game2(A).main() @ &m : res] -
  Pr[Game3(A).main() @ &m : res]| <=
`|Pr[Game3(A).main() @ &m : Game3.bad]|.
```

For a pen-and-paper proof of this relation, we can apply Lemma 3.1. However, we are not able to prove this by following the exact pen-and-paper proof for Lemma 3.1. Instead, the proof of this will be largely based on a proof of a similar relation, found in a file called `bad_abs.ec` on the EASYCRYPT GitHub page.[1] By proceeding as in this file, we also need to prove that the adversary winning and the adversary not winning are complementary events in both `Game2` and `Game3`. In EASYCRYPT, we state this as follows:

```
Pr[Game2(A).main() @ &m : res] +
  Pr[Game2(A).main() @ &m : !res] = 1%r.

Pr[Game3(A).main() @ &m : res] +
  Pr[Game3(A).main() @ &m : !res] = 1%r.
```

The next step is to put a bound on the bad event itself. We want to prove that the probability that the bad event is set to true is bounded by the probability that the adversary has guessed a hash value for some $x \notin L$.

[1] `https://github.com/EasyCrypt/easycrypt/blob/1.0/examples/cramer-shoup/bad_abs.ec`

We start by proving that the existence of a ciphertext `c = (x,e,y_')` in the ciphertext log, where

```
!c.`1 \in ls /\ c.`3 = hash_(sk.`2, c.`1, c.`2)
```

implies that the bad event will be set to true. Recall that the notation `c.`1` means the first element of the tuple `c` and so on. This is clearly true because of the way we have defined the decryption oracle in the module `Game3`. We define this in EASYCRYPT is as follows:

```
local equiv bad_bound :
  Game3(A).main ~ Game3(A).main :
  ={glob A} ==> exists c, c \in Game3.log{2} =>
  !c.`1 \in ls /\
  c.`3 = hash_(Game3.sk.`2{2}, c.`1, c.`2) =>
  Game3.bad{2}.
```

Using the **byequiv** proof tactic, the above equivalence can be transformed to a probability relation.

```
local lemma bad_bound_pr &m :
  Pr[Game3(A).main() @ &m : Game3.bad] <=
  Pr[Game3(A).main() @ &m : exists c,
    c \in Game3.log => !c.`1 \in ls /\
    c.`3 = hash_(Game3.sk.`2, c.`1, c.`2)].
```

The above inequality as well as the fact that

```
`|Pr[Game2(A).main() @ &m : res] -
  Pr[Game3(A).main() @ &m : res]| <=
`|Pr[Game3(A).main() @ &m : Game3.bad]|
```

clearly implies that the difference between the adversary winning in `Game2` and `Game3` is bounded by the probability that the adversary has been able to guess a hash value for some $x \notin L$. Or in other words,

```
`|Pr[Game2(A).main() @ &m : res] -
  Pr[Game3(A).main() @ &m : res]| <=
  Pr[Game3(A).main() @ &m : exists c,
    c \in Game3.log => !c.`1 \in ls /\
    c.`3 = hash_(Game3.sk.`2, c.`1, c.`2)].
```

For the next part of the proof, we define a module we will call `Game4`. This module will be almost equal to `Game3` with the difference that we sample `ystar` at random from the set `ys` instead of computing it as the hash value of `xstar`. We also need to define a smoothness adversary who is given `(x, s, y)` as input, and whose task is to determine if `y = hash(k,x)` or if `y` is chosen at random from the set `ys`, and make a reduction from a CCA attack to a smoothness attack. The reduction is done as follows:

```
local module SmoothAdv(A:CCAadv) = {
  var log   : ciphertext list
  var cstar : ciphertext option
  var bad   : bool
  var sk    : skey
  var k     : K
  var k_    : K_

   module O = {
    proc decrypt(c:ciphertext) : plaintext option = {
      var m : plaintext option;
      var y_', y;
      if (size log < qD && (Some c <> cstar)) {
        log <- c :: log;
        y_' <- HPS_Ext.priveval(k_, c.`1, c.`2);
        if (y_' = c.`3) {
          y <- HPS.priveval(k, c.`1);
          m <- Some (toY (toint c.`2 - toint y));
          if (!(c.`1 \in ls)
            /\ c.`3 = hash_(k_, c.`1, c.`2))
            {bad <- true;}
        } else m <- None;
      }
      else m <- None;
      return m;
    }
  }

  module A = A(O)


  proc guess(x:X, s:S, y:Y) : bool = {
    var m0, m1, b, b', s_;
      log <- []; cstar <- None; bad <- false;
      k_ <- HPS_Ext.kg(); s_ <- HPS_Ext.seval(k_);
       sk   <- (k, k_);
    (m0,m1) <- A.choose(s,s_);
       b    <$ {0,1};
     cstar  <- Some (x,
       toY (toint (b?m1:m0) + toint y),
       hash_(k_, x,
       toY (toint (b?m1:m0) + toint y)));
       b'   <- A.guess(x,
       toY (toint (b?m1:m0) + toint y),
       hash_(k_, x,
       toY (toint (b?m1:m0) + toint y)));
    return (b = b');
  }
}.
```

The smoothness adversary itself, as well as the modules where the adversary returns a guess that `y = hash(k,x)` or that `y` is chosen at random, is defined as follows:

```
module type SmoothAdversary = {
  proc guess(x:X, s:S, y:Y) : bool
}.

local module Smooth1(A:SmoothAdversary) = {
  proc main() : bool = {
    var b, x;
    x <- Sampling.fromXnotL();
    SmoothAdv.k <- HPS.kg();
    b <- A.guess(x, alpha SmoothAdv.k,
       hash(SmoothAdv.k,x));
    return b;
  }
}.

local module Smooth0(A:SmoothAdversary) = {
  proc main() : bool = {
    var b, x, y;
    x <- Sampling.fromXnotL();
    SmoothAdv.k <- HPS.kg();
    y <- Sampling.fromY();
    b <- A.guess(x, alpha SmoothAdv.k, y);
    return b;
  }
}.
```

We now need to prove that

```
`|Pr[Game3(A).main() @ &m : res] -
  Pr[Game4(A).main() @ &m : res]| =
`|Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] -
  Pr[Smooth0(SmoothAdv(A)).main() @ &m : res]|.
```

We do this by proving the following equivalences.

```
local equiv Game3_Smooth1 :
  Game3(A).main ~ Smooth1(SmoothAdv(A)).main :
  ={glob A} ==> ={res}.

local equiv Game4_Smooth0 :
  Game4(A).main ~ Smooth0(SmoothAdv(A)).main :
  ={glob A} ==> ={res}.
```

Note that in the guessing module, we use the global variable `SmoothAdv.k`, first defined in the reduction above, when sampling the secret hash key. The reason is that we need to give this key to the decryption oracle in the `SmoothAdv` module, as this is necessary for EASYCRYPT to accept the proof.

Finally, we need to prove that the adversary wins with probability $1/2$ in `Game4`. We do this by defining a module we call `Game4indep` where the adversary's output is completely independent of the hidden bit `b` (similar to the module `Game3indep` in §4.2.2). To prove that the adversary wins with probability $1/2$, we prove the following.

```
local equiv Game4_Game4indep :
  Game4(A).main ~ Game4indep(A).main :
  ={glob A} ==> ={res}.

local lemma Game4indephalf &m :
  Pr[Game4indep(A).main() @ &m : res] = 1%r/2%r.
```

Using the **byequiv** tactic to transform all equivalences to probability expressions, we now have the following relations.

```
(* 1 *)
Pr[CCA(Genscheme, A).main() @ &m : res] =
Pr[Game1(A).main() @ &m : res].

(* 2 *)
Pr[Game1(A).main() @ &m : res] =
Pr[SMP1(SMPadv(A)).main() @ &m : res].

(* 3 *)
Pr[Game2(A).main() @ &m : res] =
Pr[SMP0(SMPadv(A)).main() @ &m : res].

(* 4 *)
`|Pr[Game2(A).main() @ &m : res] -
  Pr[Game3(A).main() @ &m : res]| <=
`|Pr[Game3(A).main() @ &m : exists c,
    c \in Game3.log => !c.`1 \in ls /\
    c.`3 = hash_(Game3.sk.`2, c.`1, c.`2)]|.

(* 5 *)
Pr[Game3(A).main() @ &m : res] =
Pr[Smooth1(SmoothAdv(A)).main() @ &m : res].

(* 6 *)
Pr[Game4(A).main() @ &m : res] =
Pr[Smooth0(SmoothAdv(A)).main() @ &m : res].

(* 7 *)
Pr[Game4(A).main() @ &m : res] =
Pr[Game4indep(A).main() @ &m : res].

(* 8 *)
Pr[Game4indep(A).main() @ &m : res] = 1%r/2%r.
```

Using the above relations, we can prove our final lemma.

```
local lemma secure &m :
`|Pr[CCA(Genscheme, A).main() @ &m : res]-1%r/2%r| <=
`|Pr[SMP1(SMPadv(A)).main() @ &m : res] -
  Pr[SMP0(SMPadv(A)).main() @ &m : res]| +
`|Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] -
  Pr[Smooth0(SmoothAdv(A)).main() @ &m : res]| +
`|Pr[Game3(A).main() @ &m :
  exists c, c \in Game3.log => !c.`1 \in ls /\
  c.`3 = hash_(Game3.sk.`2, c.`1, c.`2)]|.
```

This proves, as discussed in the pen-and-paper proof sketch, that the adversary's advantage is bounded by the advantage in deciding whether $x \in L$ or $x \in X \setminus L$, the advantage in deciding whether $y = H_k(x)$ or y is random when given a tuple (x, s, y) where $s = \alpha(k)$ and $x \in X \setminus L$, as well as the probability that the adversary is able to guess a hash value of some $x \notin L$.

Chapter 5

A Concrete Construction of an Encryption Scheme

In this chapter, we construct a concrete encryption scheme based on the simplified generic scheme in §4.2. We will use the Decision Diffie-Hellman (DDH) assumption as a subset membership problem. As the simplified generic scheme, this scheme will be IND-CPA secure. The structure of the security proof for this scheme will to a large extent be the same as for the simplified generic scheme. We will also discuss some difficulties that have arised when verifying the proof in EASYCRYPT. All definitions in this chapter will follow [3], except for a few minor differences. For example, we will use multiplicative notation for our groups rather than additive because the theory for cyclic groups in EASYCRYPT is defined using multiplicative notation.

It is also possible to use the DDH assumption to derive a concrete IND-CCA encryption scheme from the generic scheme described in §4.1. However, the interesting aspects of the DDH based scheme arise when working with smoothness. For this purpose, we do not need an extended hash proof system. In other words, we leave out unnecessary details to emphasize that it is the smoothness property we are interested in.

5.1 Construction

In this section we describe how to construct the concrete scheme based on the the DDH assumption. We also describe how this scheme can be implemented in EASYCRYPT.

We first describe the *DDH assumption*. Let G be a group of prime order q. Let w be an element of $\mathbb{Z}_q$ and let w' be an element of $\mathbb{Z}_q \setminus \{0\}$. The DDH assumption is the assumption that it is hard to distinguish tuples of the form (g_0, g_1, g_0^w, g_1^w) from tuples of the form $(g_0, g_1, g_0^w, g_1^{w+w'})$ where $g_0, g_1 \in G$. Distinguishing two such tuples can serve as a subset membership problem in the following way.

Recall the sets X, W and L from §3.3.2. Let $X = G \times G$, let $W = \mathbb{Z}_q$ and let L be the subgroup of X generated by (g_0, g_1). A witness for $(x_0, x_1) \in L$ will be $w \in \mathbb{Z}_q$

such that $(x_0, x_1) = (g_0^w, g_1^w)$. It is easy to see that we can sample an element from L by first sampling w from $\mathbb{Z}_q$ and then computing $(x_0, x_1) = (g_0^w, g_1^w)$. Similarly, we can sample an element from $X \setminus L$ by first sampling w from $\mathbb{Z}_q$ and w' from $\mathbb{Z}_q \setminus \{0\}$ and then compute $(x_0, x_1) = (g_0^w, g_1^{w+w'})$. It is evident that solving this subset membership problem is equivalent to breaking the DDH assumption.

We now describe our hash proof system. Recalling the sets K, S and Y from §3.3.3, we let $K = \mathbb{Z}_q \times \mathbb{Z}_q$ and $S = Y = G$. For $(k_0, k_1) \in K$, we define our set of hash functions to be $\mathcal{H} = \{H_{k_0,k_1}\}_{k_0,k_1 \in K}$. Each $H_{k_0,k_1} : X \to Y$ is defined by $H_{k_0,k_1}(x_0, x_1) = x_0^{k_0} x_1^{k_1}$. Define $\alpha : K \to S$ by $\alpha(k_0, k_1) = g_0^{k_0} g_1^{k_1}$.

We see that given $s \in S$ and $x \in L$ along with a witness $w \in W$, we can calculate

$$s^w = (g_0^{k_0} g_1^{k_1})^w = g_0^{wk_0} g_1^{wk_1} = x_0^{k_0} x_1^{k_1} = H_{k_0,k_1}(x_0, x_1). \tag{5.1}$$

In other words, the hash family $\mathbf{H} = (\mathcal{H}, K, X, Y, L, S, \alpha)$ with all sets and α as described above, is projective.

Also, the projective hash family $\mathbf{H}$ is 0-smooth (or, equivalently, $1/q$-universal). To see this, we look at exactly how $H_{k_0,k_1}(x_0, x_1)$ is computed. For notational purposes, we will now let $H_{k_0,k_1}(x_0, x_1) = y$.

We first let $(g_0, g_1) = (g_0, g_0^r), r \in \mathbb{Z}_q$ be generators for L. For any $x = (x_0, x_1)$ we have $(x_0, x_1) = (g_0^w, g_1^{w+w'})$. If $w' = 0$ we have $(x_0, x_1) \in L$ and if $w' \neq 0$, we have $(x_0, x_1) \notin L$. For k_0, k_1 sampled at random from $\mathbb{Z}_q$, we compute $s = \alpha(k_0, k_1)$ as

$$s = g_0^{k_0} g_1^{k_1} = g_0^{k_0 + rk_1} = g_0^a,$$

with $a = k_0 + rk_1$. The hash value y is computed as

$$y = x_0^{k_0} x_1^{k_1} = g^{wk_0 + wrk_1 + w'rk_1} = g_0^{aw + bw'} = x_0^a (g_0^{w'})^b,$$

with $a = k_0 + rk_1$ and $b = rk_1$. We now see that instead of sampling k_0 and k_1 and computing $a = k_0 + rk_1$ and $b = rk_1$, we can sample a and b from $\mathbb{Z}_q$ and compute $k_0 = a - b$ and $k_1 = b/r$. By doing this, we see that for $(x_0, x_1) \in L$ (i.e. when $w' = 0$), we will have

$$y = x_0^a (g_0^{w'})^b = x_0^a \cdot 1^b,$$

meaning that y will be determined only by s and w. For $(x_0, x_1) \notin L$ (i.e. when $w' \neq 0$), however, $(g_0^{w'})^b$ will be a random group element since b is chosen completely at random. Thus, the distribution of $(g_0^{w'})^b$ and hence the distribution of y will be uniform in the group G. In other words, the hash family $\mathbf{H}$ is $1/q$- universal, or, equivalently, 0-smooth.

We now describe the encryption scheme. The message space is the group G.

Key Generation. To generate a pair of public and secret keys, we do the following.

- Sample $g_0, g_1 \xleftarrow{r} G$.
- Sample $k_0, k_1 \xleftarrow{r} \mathbb{Z}_q$.
- Compute $s \leftarrow g_0^{k_0} g_1^{k_1}$.

Set $pk = (g_0, g_1, s)$ and $sk = (k_0, k_1)$.

Encryption. To encrypt a message $m \in G$ under $pk = (g_0, g_1, s)$, we do the following.

- Sample $w \xleftarrow{r} \mathbb{Z}_q$.
- Compute $x_0 \leftarrow g_0^w, x_1 \leftarrow g_1^w, y \leftarrow s^w, e \leftarrow m \cdot y$.

The ciphertext is $c = (x_0, x_1, e)$.

Decryption. To decrypt a ciphertext $c = (x_0, x_1, e)$ under $sk = (k_0, k_1)$, we do the following.

- Compute $y' \leftarrow x_0^{k_0} x_1^{k_1}, m' \leftarrow e/y'$.

Correctness. Recall that the correctness property demands that $m = m'$ for a message encrypted and decrypted under corresponding public and secret keys. In the above scheme we see that

$$m' = e/y' = (m \cdot y)/y' = m$$

if $y = y'$. Using the projective property described in Equation 5.1, we see that this will hold.

We now describe how we can implement this scheme in EASYCRYPT. We will not explicitly implement the sampling algorithms and the hash proof systems as modules on their own. Instead, we implement everything directly into the algorithms of the encryption scheme. We first introduce the types we use in this scheme.

```
type X = group * group. type W = t.
type K = t * t. type S = group. type Y = group.
type pkey = group * group * group.
type skey = K.
type plaintext = group.
type ciphertext = group * group * group.
```

Saying that an element is of type `t` in EASYCRYPT means that the element is in $\mathbb{Z}_q$, and an element of type `group` is in the group G.

When implementing the scheme in EASYCRYPT, we will not sample g_0 and g_1 directly from the group as described above. We will use the `CyclicGroup.ec` theory developed by the EASYCRYPT team. This theory includes a generator `g` for the entire group. This generator will be our g_0. For g_1, we will sample an `r` from `FDistr.dt` (i.e. $r \xleftarrow{r} \mathbb{Z}_q$) and compute `g' <- g^r`. Now, `g'` will be our g_1 and we will let L be the subgroup of X generated by `(g, g')`.

We now describe how the algorithms of the scheme can be implemented in EASYCRYPT. The module we define will be of type `Scheme` as described in §3.1.1.

```
module DDHscheme : Scheme = {
  proc keygen() : pkey * skey = {
    var k0, k1, g', s, r, pk, sk;
    r  <$ FDistr. dt;
    k0 <$ FDistr.dt;
    k1 <$ FDistr.dt;
```

```
    g' <- g^r;
    s  <- g^k0 * g'^k1;
    pk <- (g, g', s);
    sk <- (k0, k1);
    return (pk, sk);
  }

  proc encrypt(pk:pkey, m:plaintext) : ciphertext = {
    var x0, x1, y, e, c, w;
    w  <$ FDistr.dt;
    x0 <- pk.`1^w;
    x1 <- pk.`2^w;
    y  <- pk.`3^w;
    e  <- m * y;
    c  <- (x0, x1, e);
    return c;
  }

  proc decrypt(sk:skey, c:ciphertext) : plaintext = {
    var m', y;
    y  <- c.`1^sk.`1 * c.`2^sk.`2;
    m' <- c.`3 / y;
    return m';
  }
}.
```

The correctness of this scheme can be proved using the `Correctness` module described in §3.1.1.

5.2 Security

In this section, we sketch the proof of the following theorem, and describe how to structure the proof in EASYCRYPT.

Theorem 5.1. *Let $\mathcal{A}$ be and adversary carrying out an IND-CPA attack against the encryption scheme described in §5.1. Then there exists a DDH adversary $\mathcal{B}$, such that*

$$\mathit{AdvCPA}(\mathcal{A}) = \mathit{AdvDDH}(\mathcal{B}),$$

and the run time of $\mathcal{B}$ is the same as the run time of $\mathcal{A}$.

We now sketch the proof of Theorem §5.1. We use a sequence of games very similar to the proof of the simplified generic scheme. The details of the games will of course be a bit different, as we now work with concrete group structures.

As in §4.2.2, we structure the proof using a sequence of games. Again, we let G_0 be the original CPA attack against our scheme. We let E_i be the event that the adversary wins (i.e. $b = b'$) in G_i. Thus, the adversary's advantage in winning G_0 is

$$\mathrm{AdvCPA}(\mathcal{A}) = |\Pr[E_0] - 1/2|.$$

We now describe a simulator behaving as in §4.2.2. In the context we work in now, the simulator will take an instance description Λ and some $x = (x_0, x_1) \in X = G \times G$ as input. The simulator computes $pk = s$ and $sk = (k_0, k_1)$ using the key generation algorithm of the encryption scheme. It then receives (m_0, m_1) from the adversary, samples a bit b, encrypts m_b and sends the ciphertext to the adversary, who in turn outputs a bit b'.

In the game G_1, the simulator is given $x \in L$, i.e. $x = (x_0, x_1) = (g_0^w, g_1^w)$. Here, the simulator perfectly simulates the original attack.

In G_2, the simulator is given $x = (x_0, x_1) = (g_0^w, g_1^{w+w'})$ with $w \in \mathbb{Z}_q$ and $w' \in \mathbb{Z}_q \setminus \{0\}$. We see that distinguishing between G_1 and G_2 is the same as breaking the DDH assumption. Thus, we get

$$|\Pr[E_2] - \Pr[E_1]| = \mathrm{AdvDDH}(\mathcal{B}).$$

Here, AdvDDH is the advantage in distinguishing between a DDH tuple (g_0, g_1, g_0^w, g_1^w) and a random tuple $(g_0, g_1, g_0^w, g_1^{w+w'})$.

We now make a transition from G_2 to a game G_3. Here, we modify the simulator such that instead of computing $y = x_0^{k_0} x_1^{k_1}$, the simulator samples y at random from the group. In this game, the adversary's output will be independent of the hidden bit b, so

$$\Pr[E_3] = 1/2.$$

Furthermore, the projective hash family $\mathbf{H}$ we use in this scheme is 0-smooth, or equivalently $1/q$-universal. Thus, we have

$$\Pr[E_3] = \Pr[E_2].$$

Summarizing the above relations, we see that

$$\mathrm{AdvCPA}(\mathcal{A}) = \mathrm{AdvDDH}(\mathcal{B}).$$

In the language of EASYCRYPT, we state the relation we want to prove as follows.

```
`|Pr[CPA(DDHscheme,A).main() @ &m : res]-1%r/2%r| =
`|Pr[DDH1(DDHadv(A)).main() @ &m : res] -
  Pr[DDH0(DDHadv(A)).main() @ &m : res]|.
```

We implement the two games G_1 and G_2 in EASYCRYPT as follows:

```
local module Game1(A:CPAadversary) = {
  proc main() : bool = {
    var w, pk, sk, m0, m1, b, b', y, e, c, x0, x1;
    (pk, sk) <- DDHscheme.keygen();
    w <$ FDistr.dt;
    x0 <- pk.`1^w; x1 <- pk.`2^w;
    (m0, m1) <- A.choose(pk);
    b <$ {0,1};
    y <- x0^sk.`1 * x1^sk.`2;
    e <- (b?m1:m0) * y;
    c <- (x0, x1, e);
    b' <- A.guess(c);
```

```
    return (b = b');
  }
}.

local module Game2(A:CPAadversary) = {
  proc main() : bool = {
    var w, w', pk, sk, m0, m1, b, b', y, e, c, x0, x1;
    (pk, sk) <- DDHscheme.keygen();
    w <$ FDistr.dt; w' <$ FDistr.dt \ (pred1 F.zero);
    x0 <- pk.`1^w; x1 <- pk.`2^(w + w');
    (m0, m1) <- A.choose(pk);
    b <$ {0,1};
    y <- x0^sk.`1 * x1^sk.`2;
    e <- (b?m1:m0) * y;
    c <- (x0, x1, e);
    b' <- A.guess(c);
    return (b = b');
  }
}.
```

As for the corresponding modules in the simplified generic scheme, the difference between `Game1` and `Game2` is that $x = (x_0, x_1) \in L$ and $x = (x_0, x_1) \notin L$, respectively.

We now define DDH modules and a DDH adversary.

```
module type DDHadversary = {
  proc guess(g g' s x0 x1 y : group) : bool
}.

module DDH1(A:DDHadversary) = {
  proc main() : bool = {
    var r, g', x0, x1, b, w, k0, k1, s, y;
    r <$ FDistr.dt; g' <- g^r;
    w <$ FDistr.dt;
    k0 <$ FDistr.dt; k1 <$ FDistr.dt;
    s <- g^k0 * g'^k1;
    x0 <- g^w; x1 <- g'^w;
    y <- x0^k0 * x1^k1;
    b <- A.guess(g, g', s, x0, x1, y);
    return b;
  }
}.

module DDH0(A:DDHadversary) = {
  proc main() : bool = {
    var r, g', x0, x1, b, w, w', k0, k1, s, y;
    r <$ FDistr.dt; g' <- g^r;
    w <$ FDistr.dt; w' <$ FDistr.dt \ (pred1 F.zero);
    k0 <$ FDistr.dt; k1 <$ FDistr.dt;
    s <- g^k0 * g'^k1;
    x0 <- g^w; x1 <- g'^w';
```

```
    y <- x0^k0 * x1^k1;
    b <- A.guess(g, g', s, x0, x1, y);
    return b;
  }
}.
```

We make a reduction from a CPA adversary to a DDH adversary as follows:

```
module DDHadv(A:CPAadversary) = {
  proc guess(g, g', s, x0, x1, y) : bool = {
    var m0, m1, b, b';
    (m0, m1) <- A.choose(g, g', s);
       b     <$ {0,1};
       b'    <- A.guess(x0, x1, (b?m1:m0) * y);
    return (b = b');
  }
}.
```

We can now prove that the DDH problem works as a subset membership problem by proving the following equivalences.

```
local equiv Game1_DDH1 :
  Game1(A).main ~ DDH1(DDHadv(A)).main :
  ={glob A} ==> ={res}.

local equiv Game2_DDH0 :
  Game2(A).main ~ DDH0(DDHadv(A)).main :
  ={glob A} ==> ={res}.
```

From these equivalences, it follows trivially that

```
`|Pr[Game1(A).main() @ &m : res] -
  Pr[Game2(A).main() @ &m : res]| =
`|Pr[DDH1(DDHadv(A)).main() @ &m : res] -
  Pr[DDH0(DDHadv(A)).main() @ &m : res]|.
```

As mentioned above, the projective hash family we use is 0-smooth. The first step in proving this in EASYCRYPT is to define a module which we call `Game3`.

```
local module Game3(A:CPAadversary) = {
  proc main() : bool = {
    var w, w', pk, sk, m0, m1, b, b', y, e, c, x0, x1;
    (pk, sk) <- DDHscheme.keygen();
    w <$ FDistr.dt; w' <$ FDistr.dt \ (pred1 F.zero);
    x0 <- pk.`1^w; x1 <- pk.`2^(w + w');
    (m0, m1) <- A.choose(pk);
    b <$ {0,1};
    y <$ dG;
    e <- (b?m1:m0) * y;
    c <- (x0, x1, e);
    b' <- A.guess(c);
```

```
    return (b = b');
  }
}.
```

Here, `dG` is a distribution of group elements, defined as

```
op dG : { group distr | is_lossless dG /\
  is_full dG /\ is_uniform dG } as dG_luf.
```

The only difference between `Game3` and `Game2`, is that in `Game3`, we sample `y` at random from the group instead of computing `y <- x0^k0*x1^k1`. We are able to prove in EASYCRYPT that the probability that the adversary wins in `Game3` is $1/2$, by using a procedure very similar to what we did in §4.2.2.

The next step is to define smoothness modules guessing that `y` is either computed as `x0^k0*x1^k1` or chosen at random, as well as constructing a smoothness adversary from the CPA adversary (very similar to the smoothness adversary defined in §4.2.2).

```
module type SmoothAdversary = {
  proc guess(x0 x1 g g' s y : group) : bool
}.

module Smooth1(A:SmoothAdversary) = {
  proc main() : bool = {
    var x0, x1, k0, k1, a, g', s, b, w, w', y;
    a <$ FDistr.dt; g' <- g^a;
    w <$ FDistr.dt; w' <$ FDistr.dt \ (pred1 F.zero);
    x0 <- g^w; x1 <- g'^(w + w');
    k0 <$ FDistr.dt; k1 <$ FDistr.dt;
    s <- g^k0*g'^k1;
    y <- x0^k0*x1^k1;
    b <- A.guess(x0, x1, g, g', s, y);
    return b;
  }
}.

module Smooth0(A:SmoothAdversary) = {
  proc main() : bool = {
    var x0, x1, k0, k1, a, g', s, b, w, w', y;
    a <$ FDistr.dt; g' <- g^a;
    w <$ FDistr.dt; w' <$ FDistr.dt \ (pred1 F.zero);
    x0 <- g^w; x1 <- g'^(w + w');
    k0 <$ FDistr.dt; k1 <$ FDistr.dt;
    s <- g^k0*g'^k1;
    y <$ dG;
    b <- A.guess(x0, x1, g, g', s, y);
    return b;
  }
}.
```

The reduction from a CPA attack to a smoothness attack is done as follows:

```
module SmoothAdv(A:CPAadversary) = {
  proc guess(x0, x1, g, g', s, y) = {
    var m0, m1, b, b';
    (m0, m1) <- A.choose(g, g', s);
       b     <$ {0,1};
       b'    <- A.guess(x0, x1, (b?m1:m0) * y);
    return (b = b');
  }
}.
```

As before, we give `x = (x0, x1), pk = (g, g', s)` and `y` as input (where `y` may either be randomly chosen, or the hash value of `(x0, x1)`. These values are then used in the CPA attack.

We are now able to prove that distinguishing between `Game2` and `Game3` is equivalent to distinguishing between the two smoothness modules by proving the following equivalences.

```
local equiv Game2_Smooth1 :
  Game2(A).main ~ Smooth1(SmoothAdv(A)).main :
  ={glob A} ==> ={res}.

local equiv Game3_Smooth0 :
  Game3(A).main ~ Smooth0(SmoothAdv(A)).main :
  ={glob A} ==> ={res}.
```

Transforming the equivalences regarding the DDH assumption and smoothness to probability statements, we are able to prove the following lemma, which is quite similar to the final lemma in the security proof for the IND-CPA secure generic scheme.

```
local lemma secure &m :
`|Pr[CPA(DDHscheme,A).main() @ &m : res]-1%r/2%r| <=
`|Pr[DDH1(DDHadv(A)).main() @ &m : res] -
  Pr[DDH0(DDHadv(A)).main() @ &m : res]| +
`|Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] -
  Pr[Smooth0(SmoothAdv(A)).main() @ &m : res]|.
```

The final step of the proof is to prove the fact that **H** is 0-smooth, or in other words that

```
Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] =
Pr[Smooth0(SmoothAdv(A)).main() @ &m : res].
```

This is a relation we have not been able to fully prove in EASYCRYPT. We have, however, managed to prove one step towards proving the above relation, namely that $x_0^{k_0} x_1^{k_1}$ is distributed uniformly in G when $(x_0, x_1) \notin L$. Formally, the statement we want to prove is

$$\Pr[H_{k_0,k_1}(x_0, x_1) = y \mid s] = 1/q,$$

for all $(x_0, x_1) \notin L$, $y \in G$ and $s = \alpha(k_0, k_1)$.

To state this in EASYCRYPT, we first need the following module.

```
module Ycomp = {
  proc main(s:group, w w' r : t) : Y = {
    var x0, x1, k0, k1, y;
    k1 <$ FDistr.dt;
    x0 <- g^w;
    x1 <- g^r^(w+w');
    k0 <- log s - k1 * r;
    y  <- x0^k0*x1^k1;
    return y;
  }
}.
```

Note that to make sure that `s = alpha(k0, k1)` we do not sample `k0` at random, but instead compute `k0` in a suitable manner.

The probability statement above is stated as

```
lemma Ycomp_pr &m  :
forall(y s : group, r w w' : t),
w <> F.zero /\ w' <> F.zero /\ r <> F.zero =>
Pr[Ycomp.main(s, w, w', r) @ &m : res = y] = 1%r/q%r.
```

Note that we take `w, w'` and `r` as input along with `s`. This seems to be necessary for the proof to pass, as EASYCRYPT otherwise does not understand which values we are reasoning about when we try to reason about `w, w'` and `r`. We must also assume that all these values are different from 0, as we end up with a subgoal in EASYCRYPT where we have to divide by `r * w'`, in which case the product cannot be 0.

However, even though we have proved the fact above, we have not been able to prove that

```
Pr[Smooth1(SmoothAdv(A)).main() @ &m : res] =
Pr[Smooth0(SmoothAdv(A)).main() @ &m : res].
```

The problem seems to arise when comparing the *regular assignment*

```
y <- x0^k0*x1^k1
```

to the *random assignment*

```
 y <$ dG.
```

We have this far not been able to find a proper way of comparing two such assignments. However, even though we have not been able to show that the advantage of the smoothness adversary is zero, we have at least been able to prove the fact that the projective hash family **H** that we use in this scheme, is 0-smooth.

Chapter 6

Concluding Remarks

In this masters thesis we have implemented three encryption schemes in EASYCRYPT, namely the generic encryption scheme proposed in [3], a less secure simplification of this scheme, and a concretization of the simplified scheme, based on the Decision Diffie Hellman assumption. We have also verified the IND-CCA security of the generic scheme and the IND-CPA security of the simplified generic scheme under the assumption that the underlying subset membership problems are hard.

As for the security of the DDH based scheme, there is still a small gap in our security proof. We have been able to prove that the DDH assumption is a suitable subset membership problem and that for any $x \in X$ that does not lie in the special subset $L \subset X$, the hash value of x is uniformly distributed in the group G (which is, in fact all we need as this proves that the projective hash family $\mathbf{H}$ is 0-smooth). We have not, however, been able to use this fact to prove that the advantage of the smoothness adversary we have defined is zero. The issue seems to arise when we try to compare a regular assignment and a random assignment.

As there is no complete user manual, it seems at the moment that the best way of learning to use EASYCRYPT is by studying code examples developed by others. This means that learning to use EASYCRYPT is (at least for us) a rather long process involving a lot of experimenting, patience and trial and error. Here, we must again emphasize that the EASYCRYPT team is very helpful in answering questions. Also, the bug that was found which allowed us to prove that an arbitrary prime number q is equal to 1 (discussed in §2.1.5) shows that EASYCRYPT may be prone to bugs allowing us to prove something that is in fact false. Certainly, bugs may be an issue that is hard to avoid in any computer program. However, bugs allowing us to prove false statements are quite critical in programs whose purpose is to verify proofs. We recommend anyone working with EASYCRYPT and finding a bug, to report this bug to the EASYCRYPT team so that it may be fixed.

Other than this, EASYCRYPT seems to be a reasonable tool for verifying security proofs, and the need for such tools seems to become more and more important, as both cryptographic constructions and their security proofs become more complex.

6.1 Further Work

We believe it would be interesting to continue working on the concrete scheme based on the Decision Diffie-Hellman assumption. Firstly, it would be interesting to continue working on the CPA-security to hopefully figure out how we compare regular assignments to random assignments. Secondly, it would be interesting to try and implement the full scheme based on the DDH assumption (found in [3]) and prove this scheme's IND-CCA security. Even though the most interesting part of this scheme is the 0-smoothness property, it would be a fun exercise to implement the extended hash family used in the IND-CCA secure DDH scheme, as this involves working with for example finite sums in EASYCRYPT.

Bibliography

[1] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, Cham, 2014. Springer International Publishing.

[2] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *Eurocrypt 2006, LNCS*, pages 409–426, 2008.

[3] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption, 2001. cramer@brics.dk, sho@zurich.ibm.com 11668 received 12 Oct 2001, last revised 12 Dec 2001.

[4] Mikkel Langtangen Furuberg and Morten Rotvold Solberg. An introduction to easycrypt and the security of the elgamal cryptosystem, 2017. Project assignment at NTNU, fall 2017. To get access to this article, send an e-mail to kristian.gjosteen@ntnu.no.

[5] Shai Halevi and Yael Tauman Kalai. Smooth projective hashing and two-message oblivious transfer. Cryptology ePrint Archive, Report 2007/118, 2007. `https://eprint.iacr.org/2007/118`.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[7] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer, 1997.

[8] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/2004/332`.

[9] The EasyCrypt Team. Easycrypt reference manual. `https://www.easycrypt.info/documentation/refman.pdf`. [Version 1.x compiled 2018-02-19].