Modelling Oblivious Transfer in EasyCrypt

Mikkel Langtangen Furuberg

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

Norwegian University of Science and Technology
Department of Mathematical Sciences

Preface

This master's thesis is written during the spring semester of 2018, as the final part of the master's programme in Natural Science with Teaching Education at the Norwegian University of Science and Technology (NTNU, Trondheim). The idea of the project was brought up by my supervisor as part of doing research of machine-checking proofs in cryptography, and succeeds the work in EasyCrypt last semester.

Mikkel Langtangen Furuberg
Trondheim, June 1, 2018

Acknowledgments

A special thanks to my supervisor, Kristian Gjøsteen, for insightful and constructive feedback on my work. Our weekly meetings have been crucial for the progress of this master's thesis.

To my fellow students at Matteland, thank you for all the lunch breaks, quizzes and table tennis matches. The positive environment has been a huge motivation for showing up early in the office and staying until late.

I would also like to express my appreciation to all my friends, both in Trondheim and Oslo, for making me think of other things than mathematics, there are actually some other entertaining topics to discuss! Especially thanks to Marit for proofreading and making figures for the thesis, and also for encouraging me throughout the process.

Finally, thanks to my family for being there for me. To my mom for discussions and for proofreading the thesis, and to my dad for making me interested in mathematics and computer science. Really wish that you could read the thesis and give me guidance.

Abstract

We describe how to construct an oblivious transfer protocol which security is based on subset membership problems and smooth projective hash functions. A specific protocol based on the two-message oblivious transfer protocols of Kalai (2005) and the encryption schemes presented by Cramer and Shoup (2002) is presented.

In addition, the protocol is modelled in EASYCRYPT and we use the program to prove the security, which is based on the Decisional Diffie-Hellman assumption. However, we encountered challenges using the program, which is under development. The stability of EASYCRYPT was a problem. Axioms that were important for our implementation were not available, and the necessary information on syntax needed to define new modules and lemmas was inadequate. We had to rewrite existent- and compose many new lemmas. We also found a critical error in EASYCRYPT, which was given high priority by the team behind EASYCRYPT, and has now been corrected. However, our proofs had to surround the problem and became more complicated. We also concluded that EASYCRYPT is not developed primarily for doing algebra, resulting in unnecessary complicated codes for the proof of our protocol.

Sammendrag

Vi beskriver hvordan man konstruerer en oblivious transfer-protokoll, hvor sikkerheten er basert på undergruppeproblemer og glatte projektive hashfunksjoner. Vi presenterer en 1-ut-av-2 oblivious transfer-protokoll basert på Kalai (2005) sin protokoll, og Cramer og Shoup (2002) sitt krypteringssystem.

I tillegg modellerer vi protokollen i programmet EASYCRYPT, som vi bruker til å bevise sikkerheten. Sikkerheten er basert på Decicional Diffie-Hellman-antagelsen. Under modelleringen i EASYCRYPT møtte vi flere utfordringer. Blant annet var stabiliteten til programmet et problem. I tillegg er ikke rammeverket fullstendig, noe som kompliserte implementasjonen og vi måtte skrive mange nye, korte lemmaer for å forenkle bevisene. Vi fant en kritisk feil i EASYCRYPT som ble høyt prioritert av utviklerne av programmet, og har nå blitt rettet opp i. Likevel førte feilen til at bevisene ble mer kompliserte, som et resultat av at vi måtte unngå visse områder av EASYCRYPT. Vi konkluderte også med at programmet ikke er laget primært for å bruke algebra, noe som førte til mye unødvendig kode og tidsbruk.

Table of Contents

Chapter 1

Introduction

1.1 Background

Cryptography plays a key role in the security of modern communication. As a result, it is increasingly important to design new cryptographic systems which yield high security guarantees. For new and complex cryptographic systems, security proofs may be complicated. To cope with the increased vulnerability to errors in manual cryptographic proofs, computer aided proofs have become more central in cryptography. Programs designed for cryptographic security proofs can guarantee that the reasoning of a proof is applied correctly. EASYCRYPT is a program designed for verifying cryptographic proofs.

In cryptography, an *oblivious transfer* protocol is a type of protocol in which a sender transfers one of potentially many pieces of information to a receiver, but remains oblivious as to what piece (if any) has been transferred. An example of how oblivious transfer strengthen private information retrieval form a database is seen in the case where a receiver gets exactly one database element from a database, without the sender/database getting to know which element was queried, and without the receiver knowing anything about the other elements that were not retrieved.

Further work has revealed oblivious transfer to be a fundamental and important problem in cryptography. It is considered one of the critical problems in the field, because of the importance of the applications that can be built based on it. One of the benefits of oblivious transfer is that the security of a protocol can be proved by different mathematical assumptions.

This thesis introduces a cryptographic transfer protocol based on *1-out-of-2* oblivious transfer. We first adapt a general proof for oblivious transfer, presented by Kalai [5], to our protocol. We then model the cryptographic basis for the proof and the protocol as modules in EASYCRYPT. Then, we machine-check the security by proving it in EASYCRYPT.

1.2 Outline of the Thesis

We will in this thesis look at how a specific protocol can be modelled in EasyCrypt.

Chapter 2 contains a short introduction to EasyCrypt. Since we implement a protocol where the proof is based on cyclic group theory, we will focus on how group theory can be expressed in EasyCrypt. We then explain how EasyCrypt may be set up to define and verify proofs.

In Chapter 3, we discuss some security notions that are important for oblivious transfer and the proof of our protocol. We define the Decisional Diffie-Hellman assumption that can be used as a basis to prove the security of many cryptographic protocols. Moreover, we describe the concepts *subset membership problems* and *smooth projective hash functions*, introduced by Cramer and Shoup [2]. Finally, we present a way to model hard subset membership problems and smooth projective hash functions in EasyCrypt.

In Chapter 4, oblivious transfer is discussed, and an example of an oblivious transfer protocol based on the RSA cryptosystem is included. We also present another general example of a protocol based on the article by Kalai [5].

Chapter 5 includes a presentation of an oblivious transfer protocol based on the protocol introduced in [5] and an encryption scheme presented in [2]. The chapter also includes the implementation of the protocol in EasyCrypt.

In Chapter 6, we discuss the security of the oblivious protocol presented in Chapter 5, which allows us to use the theory of Chapter 3. A proof of the security in EasyCrypt is presented, and the implementation is discussed.

Finally, in Chapter 7, we discuss the use of EasyCrypt, with respect to both cryptography in general, and the presented oblivious transfer protocol particularly.

1.3 Cryptography in School Teaching

As this thesis is a part of the master's programme in Natural Science with Teacher Education, we describe the relevance to the teaching profession, and how cryptography can be embedded and used in school.

Cryptography is a branch of mathematics with many everyday applications in our information society. It is easy to explain the importance of cryptography in situations from the daily life of students, such as online banking and communication in social media. Therefore, teaching simple mathematics with applications within cryptography may inspire students to understand the importance of learning mathematics. The most important ideas of the math used in cryptography is not described in the competence aims set out by the Norwegian Directorate for Education and Training for secondary- and high schools in Norway. This means that teaching cryptography in detail on a general basis is not possible. However, for students that need challenges where they may use parts of the standard curriculum combined with new ideas, exercises and assignments, use of cryptography may be inspiring.

Although the mathematics behind cryptography can be advanced, such as group theory, it can be based on calculations with remainders, which is taught in the primary school. Therefore, cryptography can be presented as a project to pupils who needs extra challenges in the mathematics class. Over a period, such pupils should be able to learn some simple cryptographic concepts like Caesar cipher, substitution cipher, Affine cipher and even encryption schemes like RSA. The fact that some ciphertexts can be decryptet without knowing the system makes cryptography an excellent way of using problem solving and research to find the original message. This is also a very practical way to use mathematics and logic to solve problems that can be exciting for the pupils.

Classic cryptography, like Caesar cipher and Affine cipher, which have an easy mathematical solution to decrypt the messages, can also be used in any class as a variation from the classic exercises. To decrypt a message requires investigation and let the pupils explore new ways of using mathematics. These systems can be solved both mathematically, by finding the number of shifts in Caesar cipher or the function in Affine cipher, or by a brute-force attack. In more advanced terms, other attack forms can be introduced. For instance, the pulips can perform known-plaintext attacks, (adaptive) chosen-plaintext attacks or (adaptive) chosen ciphertext attacks.

Another way of doing research with cryptography is to argue for why a form of communication is secure. The easiest way to secure a system is against known ciphertext attacks, which allows the pupils to study how to secure a system against statistical analysis, such as frequency analysis.

The protocol presented in this thesis will probably be too advanced and complicated for if based on the mathematical tools learned in school, but will nevertheless be possible to learn if there some new concepts are introduced. The students will need to learn modulo computation with multiplication, which is not too difficult as it is based on the simple concepts of remainders and powers. This could be an extended project for pupils who reach the competence aims quickly. As mentioned, the field of cryptography can be a project which starts with classical cryptography and ends with the protocol presented in this thesis.

Chapter 2

Introduction to EasyCrypt

We will present a brief introduction to EASYCRYPT, a program used to verify cryptographic proofs. First, we will define some key concepts for this thesis and then explain how EASYCRYPT may be set up to define and verify proofs. The reason to use such a program is to guarantee that the reasoning is applied correctly, and to prevent careless mistakes that may arise in manual proofs. The team behind EASYCRYPT has released a reference manual [1], which is an incomplete overview of the keywords used in the program. Additionally, we have written an introduction to EASYCRYPT, "An Introduction to EasyCrypt and the Security of the ElGamal Cryptosystem" [4], which contains a larger part of how EASYCRYPT works, the logic behind it and a presentation of keywords. This chapter will be based on the papers mentioned above and we will present some keywords and tactics used in this thesis. Note that the introduction given in this chapter is not complete, or necessarily conventional, as it is adapted for this thesis.

2.1 How EasyCrypt Works and General Tactics

EASYCRYPT is a framework for evaluating cryptographic proofs on computers. In this way, the proofs can be verified step-by-step by the computer, which make them more reliable. The program uses *Proof General* to check the proofs. Proof General is a proof assistant, a software tool, which is the base to validate the proofs written in EASYCRYPT. The mathematical formulas have to be expressed in a formal language, and the proofs written in EASYCRYPT have to be stringently formulated in order to make the program useful.

The EASYCRYPT library consists of several files with theories containing axioms, lemmas and operators. These are the framework for the program. Examples are files with codes defining how integers, real numbers, functions, distributions and group theory works. Every operation, distribution etc. are defined, and the properties are either proved, based on previous proved properties, or stated as axioms.

If one states a lemma, it has to be proved, while an axiom not requires any proof. This makes lemmas more safe to work with as the system prevents proving wrong statements, while this is not guaranteed when using axioms. A wrong formulation of an axiom can potentially make everything in EASYCRYPT true, so in our approach we will try to avoid axioms and base the proofs on lemmas.

When declaring a variable or constant in EASYCRYPT, one also has to assign a certain *type* to the variable. EASYCRYPT has some built-in types like `real`, `int` and `bool`, meaning that the declared variable is an integer, a real number or a boolean, respectively. In addition to the already built-in types, one may also declare types oneself, which can be abstract:

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

Or well defined:

```
type X = group * group.
```

where $X = G \times G$, G a group.

Operators are an important feature in EASYCRYPT. Besides well-known operators in the library such as $+$, $\cdot$, $=$, ^and $<$, one can formulate operators restricted to certain types and they are very useful as for example functions. An operator is either formulated with just inputs and outputs:

```
op ( < )  : int -> int -> bool.
```

stating that $<$ takes in two integers and returns a value of true or false. Or an operator can be more specific:

```
op one    : int = 1.
op raise (x y:int) = x^y.
op times (x:X) = x.`1*x.`2
```

The operator `times` takes an element x of type X, X as defined above, and returns the product of the first and second element of x.

Note that a commentary in EASYCRYPT begins with "(*" and ends with "*)", i.e. *`(* Commentary.. *)`*. Moreover, <> means not equal and $1\%r$ is used when considering the integer 1 as a real number. The notation `x.`1` is used to indicate the first element in $x = (x_0, x_1, ...)$, `x.`2` to indicate the second element in x and so on.

2.2 Group Theory in EasyCrypt

In this thesis, some files and types are used more frequently than others. As we are going to implement a protocol based on cyclic group theory, we present some important features related to this. First, two important types are considered: `group` and `t`.

The type defined as group consists of operations and properties to define a multiplicative group and is found in the file `CyclicGroup.ec`. There are several operators defined for the type group, and the order of the group is q. The operators defined in the `CyclicGroup.ec` file are:

```
op g:group. (* the generator *)
op ( * ): group -> group -> group.
op inv : group -> group.
op ( / ): group -> group -> group.
op ( ^ ): group -> t -> group.
op log  : group -> t.
op g1 = g ^ F.zero. (* identity element *)
```

Note that `g` is specified as ”the generator”, and is the only defined generator for such a group.

The type `t` is used in exponentiation of group elements. For exponentiation, a group element is the base and an element of type `t` is the exponent. We refer to the file `PrimeField.ec` for more details. The type `t` has the following defined operators:

```
 op zero : t. (* zero *)
  op one  : t. (* one *)

  op ( * ): t -> t -> t.
    (* multiplication modulo q *)
  op ( + ): t -> t -> t.
    (* addition modulo q *)
  op [ - ]: t -> t.
    (* the additive inverse *)
  op inv: t -> t.
    (* the multiplicative inverse *)

  op (-) : t -> t -> t.
    (* subtraction modulo q *)
  op (/) : t -> t -> t.
    (* division modulo q for y <> 0 *)
  op (^) : t -> int -> t.
    (* exponentiation *)
```

In addition to the operators, a handful of lemmas and axioms are written. The frequently used axioms and lemmas from the files `CyclicGroup.ec` and `PrimeField.ec`, without proof, are found in the Appendix A.

As there are many elements of for example zeros, depending on what kind of types we look at, we have to import files to make use of the operators, axioms and lemmas. For example, `G` is used as the prefix for the cyclic group file and `F` for the prime field file. This means that `G.g1` is the way to use the generator `g1` from `CyclicGroup.ec`, while `F.zero` is the way to use the zero element from `PrimeField.ec`.

There is also a distribution of elements of type `t`. This means that it is possible to draw an element of type `t` at random, and the value of `t` is between 0 and $q-1$. This is called `Fistr.dt`, so if we want to sample an element m of type `t` at random, it looks like this:

```
m <$ FDistr.dt;
```

2.3 Formulating Proofs

When proving a statement in EASYCRYPT, lemma is the notation to be used. If a lemma is proved, then it can be used later and is added to the library of EASYCRYPT. Any lemma must be proved before it can be reused to prove other lemmas. Thus, a lemma along with its proof takes the form

```
lemma name : condition1 => ··· =>
what you want to prove.
proof.
  tactic_1
  .
  .
  .
  tactic_n
qed.
```

=> is used as "and" for every condition until the last =>, then it used as implication.

There are two ways of setting up a lemma. The first one is by formulating the statement precisely in the lemma. This is used when proving identities and small theories. Examples of this, excluding the actual proof, are:

```
lemma exists_m : forall a, exists m, a = G.g ^ m.

lemma exp_g : G.g ^ F.one = G.g by smt.

lemma inv_not_one (x y:t) : x<>y =>
y<>F.zero => (x * (inv y)) <> F.one.

lemma aeq1_in_G (a : group) :
log(a) = F.zero <=> a = G.g1.
```

The other way to set up a proof is by creating one or more *modules*. A module consists of *procedures* and this makes way to simulate a case. This is used in proofs when considering probability and setting up a sequence of games. The example of sampling an m of type t have to be done in a module, and will look like this.

```
module samplex = {
    proc main() = {
        var m;
        m <$ FDistr.dt;
        return m;
    }
}.
```

For more theory of modules and procedures, we refer to [4] and [1].

We can for example formulate a proof where we want to prove that the probability of a random m of type t to equal an other value, say n, is $\frac{1}{q}$ (&m denotes that we are working in a memory m, which we have to do when referring to a module. This is not the m sampled from samplex):

```
lemma m_to_equal_n &m : forall(n:t),
Pr[M1.main() @ &m : res = n] = 1%r/F.q%r.
```

Or we can state that if we sample two values, i.e. run the module `samplex` twice, the probability of returning n is equal.

```
lemma samplex_twice &m : forall(n:t),
Pr[M1.main() @ &m : res = n]
= Pr[M1.main() @ &m : res = n].
```

This is of course trivial, but just an example on how to compare the probability of two events.

2.4 Verifying Proofs

EasyCrypt can solve simple problems using axioms and lemmas from the library. This is done with *SMT solvers* which are a collection of external provers. SMT is an abbreviation for *Satisfiability Modulo Theories*, and such solvers combine background theory to solve first-order logic problems. The tactic in proofs is to break down the goal to smaller problems and mainly use the keyword **smt** to solve the subgoals. The type and complexity of the problems SMT solvers can solve differ from situation to situation. When a lemma is proved, EasyCrypt adds the lemma to the library and it can be used by the SMT solvers to prove additional lemmas.

To make use of the SMT solvers, we have to break down the problems into smaller problems where there are lemmas the solvers can use. One way to do this is by the keyword **have**, which allows one to divide the problem into subparts, which may be solved independently in order to reach the goal.

2.4.1 Examples

Using previous written lemmas is a well-used tactic when proving larger lemmas. In order to prove the lemma `samplex_twice` from the previous section, we can use **have** together with the tactic **rewrite**, and say that both sides are equal to $\frac{1}{q}$ as proved in the lemma `m_to_equal_n`. We then end up with $\frac{1}{q} = \frac{1}{q}$, which can be solved by **smt**, and this completes the proof. In EasyCrypt the beginning of the proof looks like this:

```
lemma samplex_twice &m : forall(n:t),
Pr[M1.main() @ &m : res = n]
= Pr[M1.main() @ &m : res = n].
proof.
move => ?. have : Pr[M1.main() @ &m : res = n]
= 1%r/F.q%r. rewrite (m_to_equal_n &m n).
```

move is a tactic to move the conditions away from the actual proof, this time saying that we have an n, and this makes n a variable. The state of the proof after **have** is that we break down the statement to show that `Pr[M1.main() @ &m : res = n]` is $\frac{1}{q}$. This can be showed by using the lemma `m_to_equal_n` together with the memory. Further, we can use **smt** to close that goal. Both sides can be rewritten to $\frac{1}{q}$ which proves the lemma correct. The rest of the proof will look like this:

```
move => ?. rewrite H. smt.
qed.
```

Where H is the proven statement using **have**.

We also show how `m_to_equal_n` is proved in EASYCRYPT. This includes an procedure that we have to run.

```
lemma m_to_equal_n &m : forall(n:t),
Pr[M1.main() @ &m : res = n] = 1%r/F.q%r.
proof.
progress. byphoare => //. proc. rnd. auto.
progress. rewrite FDistr.dt1E. smt.
qed.
```

byphoare is a keyword used in probability, to get into the module, while **proc** is used to get into the procedure. **rnd** is used to do the sampling of m. **auto** and **progress** are collections of other keywords and we refer the reader to [4] for more information. `FDistr.dt1E` is an axiom stating that there are q possibilities when sampling an element of type `t`.

Chapter 3

Background and Definitions

In this chapter, we will present some important concepts and notions on *subset membership problems* and *projective hash functions* which will be the basis for the work described in Chapters 4, 5 and 6. The concepts from this chapter will be discussed further, together with oblivious transfer, in a concrete protocol in Chapter 5. The approach in this chapter will concentrate on the concepts needed for the forthcoming protocol.

In the first section we will look at some notions used in security proofs. Then we will move on to introduce the Decisional Diffie-Hellman problem, Subset Membership Problems and Smooth Projective Hash Functions, which all will be used as the basis for the security analysis in this thesis.

We end the chapter with modelling some of the definitions in EASYCRYPT, and look at how it can be used to encrypt messages.

3.1 Attack Games and Advantage

First, we present some notion on *attack games* and *advantage*, which are used to consider security in cryptography. This is a way to simulate an attack from an adversary and consider the probability of the adversary to succeed.

After a protocol for secure transmission of messages have been suggested, the security of the protocol may be defined and proved by using a sequence of *attack games* [7]. Such a game is played between some *challenger* and an *adversary*. The adversary is the one carrying out the attack.

Both the challenger and the adversary are probabilistic, so it is possible to model the game as a probability space. Typically, the security of a cryptosystem is connected to the probability of some event E, denoted Pr$[E]$.

The goal is often to prove that Pr$[E]$ is *negligibly close* to some target probability, often 0, $1/2$ or the probability Pr$[E']$, where E' is some event in another attack game where the same adversary plays against a different challenger.

To carry out the proof, one defines a sequence of games, from Game 0 to Game n. Game 0 is the original attack game for the system we want to prove the security of and

E_0 is the event E mentioned above. For $i \in \{1, ..., n\}$, the event E_i in game i should be such that Pr$[E_i]$ is negligibly close to Pr$[E_{i+1}]$ for $i \in \{1, ..., n-1\}$. Furthermore, Pr$[E_n]$ should be negligibly close to the target probability. Thus, Pr$[E]$ will also be negligibly close to the target probability. The goal with such games is to combine them and use them to reduce an idea to one or more well known cryptographic assumptions.

When we look at cryptographic systems, we are interested how good a potential adversary will make it in an attack. To determine this we use the term *advantage*. The advantage is a measure of how good an adversary will do in distinguishing two values. We know that in a distinguishing problem one can always guess, which obviously gives the probability $\frac{1}{2}$. We therefore define the advantage of an adversary, A, against a concept, C, as

$$Adv_C(A) = |Succ_C(A) - \frac{1}{2}|,$$

where $Succ_C(A)$ is the probability of A to succeed, which means that A guesses correctly.

3.2 The Decisional Diffie-Hellman Assumption

One of the assumptions we are going to look at is the the Decisional Diffie-Hellman assumption (DDH). The DDH assumption states that it is hard to distinguish tuples of the form

$$(g, g^a, g^w, g^{aw'})$$

from DDH tuples of the form

$$(g, g^a, g^w, g^{aw})$$

where g is a generator for a group G of order q, and $a, w, w' \overset{r}{\leftarrow} \mathbb{Z}_q$, $w \neq w'$.[1] Furthermore, we define the DDH advantage of an distinguishing algorithm D, which takes as input quadruples of group elements and output 1 if the quadruple is a DDH tuple, to be

$$Adv_{DDH}(D) =$$
$$|Pr[a, w \overset{r}{\leftarrow} \mathbb{Z}_q : D(g, g^a, g^w, g^{aw}) = 1] - Pr[a, w, w' \overset{r}{\leftarrow} \mathbb{Z}_q : D(g, g^a, g^w, g^{aw'}) = 1]|$$

The DDH assumption is the assumption that the DDH advantage is negligible for any efficient algorithm D. We also set up a game with a challenger and an adversary $\mathcal{A}$, which is found in Figure 3.1.

3.3 Subset Membership Problems

One of the important concepts we will look at is the subset membership problem. We will use this in Chapter 6 when we discuss the security of oblivious transfer. The subset membership problem states that it should be difficult to distinguish between two elements, one from a given subset and one which is from the relative complement of the subset with respect to the original set.

[1]We denote by $x \overset{r}{\leftarrow} X$ the action of uniformly choosing an element from the set X.

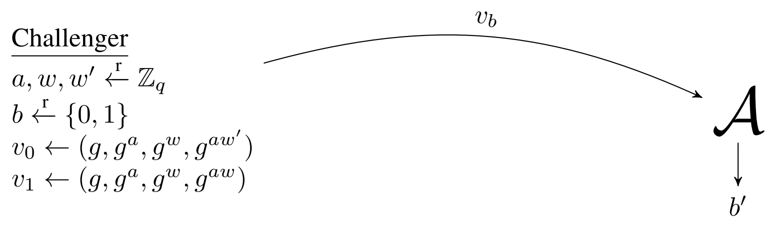

Challenger
$a, w, w' \overset{r}{\leftarrow} \mathbb{Z}_q$
$b \overset{r}{\leftarrow} \{0,1\}$
$v_0 \leftarrow (g, g^a, g^w, g^{aw'})$
$v_1 \leftarrow (g, g^a, g^w, g^{aw})$
v_b
$\mathcal{A}$
b'

Figure 3.1: Decisional Diffie-Hellman game.

A subset membership problem $\mathbf{M}$ specifies a collection $\{I_n\}_{n\in\mathbb{N}}$ of distributions. For every value of $n \in \mathbb{N}$, I_n is a probability distribution of *instance descriptions* Λ. Each instance description Λ specifies the following.

- Two finite non-empty sets, $X, W \subseteq \{0,1\}^{poly(n)}$
- A relation $R \subset X \times W$,
- A non-empty subset $L \subset X$, where $L = \{x : \exists w \; s.t. \; (x, w) \in R\}$

For every $x \in X$ and $w \in W$, if $(x, w) \in R$, then we say that w is a *witness* for x. We require that the relation R is an NP-relation, i.e. it is not possible to find the corresponding witness to a given element in X in polynomial time. The role of a witness will become apparent later in the thesis as we introduce smooth projective hash functions. The notation $\Lambda[X, W, L, R]$ indicates that the instance description Λ specifies X, W, L and R as above.

A subset membership problem is to distinguish two values, $x_0 \overset{r}{\leftarrow} L$ and $x_1 \overset{r}{\leftarrow} X\backslash L$, for a specified Λ with security parameter n.

Loosely speaking, this means that it is hard to distinguish between random elements from L and random elements from $X\backslash L$.

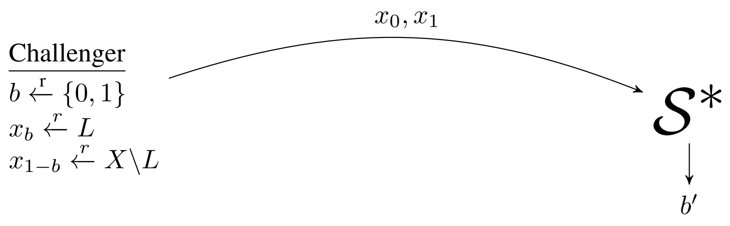
3.3.1 Security Game of Subset Membership Problems

The security of a subset membership problem is based on that, for an adversary $\mathcal{S}^*$, $\mathcal{S}^*$ should not be able to distinguish between elements in L and elements in $X\backslash L$. We set up an attack game for this situation, where $\mathcal{S}^*$ gets two elements and guesses which is in L.

Note that this game differs from the game presented in [7] and Section 3.2, where the adversary only gets one element. If we make an algorithm $\mathcal{D}$ which chooses one of the elements x_0, x_1 in our game, and provides this to the adversary, we see that the games are identical.

Based on this, we define the subset membership problem-advantage to be

$$Adv_{SMP}(\mathcal{S}^*) = |Pr[b = b'] - \frac{1}{2}|.$$



Challenger
$b \xleftarrow{r} \{0,1\}$
$x_b \xleftarrow{r} L$
$x_{1-b} \xleftarrow{r} X\backslash L$
x_0, x_1
$\mathcal{S}^*$
b'

Figure 3.2: Subset membership problem game.

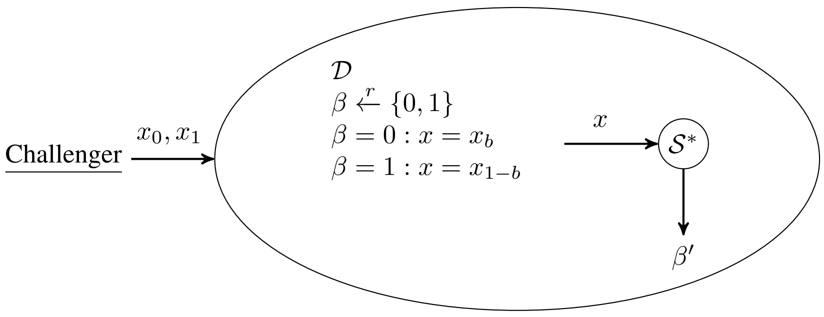


Figure 3.3: Relation of the games.

3.3.2 Subset Membership Algorithms

Kalai [5] lists four algorithms needed to verify a usable subset membership problem $\mathbf{M} = \{I_n\}_{n\in\mathbb{N}}$ to be used in oblivious transfer. If $\mathbf{M}$ satisfies these four, it is said to be *verifiably samplable*. If a subset membership problem $\mathbf{M}$ is verifiably samplable, then it is easy to to sample uniformly from both L and $X\backslash L$, and it is easy to verify that for two elements x_0 and x_1, either $x_0 \in X\backslash L$ or $x_1 \in X\backslash L$. It will become clear in Chapter 6 why the last property is important. If the four following algorithms exist, then $\mathbf{M}$ is verifiably samplable:

1. A probabilistic polynomial-time algorithm that samples $\Lambda = [X, W, L, R]$ on input 1^n, according to I_n.

2. A probabilistic polynomial-time algorithm that on input an instance description $\Lambda = [X, W, L, R] \in \mathbf{M}$, outputs an element $x \in L$ and its witness $w \in W$, such that the distribution of x is statistically close to uniform on L.

3. A probabilistic polynomial-time algorithm that on input an instance description $\Lambda = [X, W, L, R]$ and an element $x_0 \in L$, outputs an element x_1 such that if $x_0 \xleftarrow{r} L$ then the distribution of x_1 is statistically close to uniform on $X\backslash L$, and if $x_0 \xleftarrow{r} X$ then the distribution of x_1 is statistically close to uniform on X.

4. A probabilistic polynomial-time algorithm that on input an instance description $\Lambda = [X, W, L, R]$ and two elements $x_0, x_1 \in X$, checks that there exists a bit b such that $x_b \in X \backslash L$. This property should hold for maliciously chosen Λ.
   - The algorithm outputs 0 for every Λ and x_0, x_1 if $x_0 \notin X \backslash L$ and $x_1 \notin X \backslash L$.
   - The algorithm outputs 1 for every Λ and x_0, x_1 if there exists a bit b such that $x_b \in X \backslash L$.

3.4 Smooth Projective Hash Functions

Smooth projective hash functions will be used as the basis for the main development of the security of the protocol described in Chapter 5, and are, together with the subset membership problem, the core concept of the work with security in this thesis.

The notion smooth projective hash functions was presented by Cramer and Shoup in 2002 [2]. It is based on a set or *family* of hash functions, and is built up on different concepts, which will be discussed in this section. The idea is further discussed in [5], and this will be used as the basis for the concept smooth projective hash functions in this thesis.

3.4.1 Hash Families

General hash functions are widely used in computer science to map data of any length to data of a fixed length. Such functions are also used in cryptography, calling them *cryptographic hash functions*, and the properties of these functions are specialized for cryptographic use. These properties include that the function is a one-way function, deterministic, it is infeasible to find to messages that give the same output, and that a small change in the input gives an output that seems uncorrelated to the first input. First we define a *hash family*.

Definition 3.4.1 (Hash family)**.** Let $\mathcal{H} = \{H_{hk}\}_{hk \in K}$ be a collection of hash functions, where K is the key space. For every $hk \in K$, H_{hk} is a hash function from X into G, where G is a finite non-empty set. We call $\mathbf{F} = (\mathcal{H}, K, X, G)$ a hash family, where every H_{hk} is a hash function.

3.4.2 Projective Hash Families

We expand the definition of a hash family to what is called a *projective hash family*. Let $\mathbf{F} = (\mathcal{H}, K, X, G)$ be a hash family as described above. Let L be a non-empty, proper subset of X, and let S be a finite, non-empty set. We also define a function $\alpha : K \to S$, called the *projection key*.

Definition 3.4.2 (Projective hash family)**.** $\mathbf{H} = (\mathcal{H}, K, S, \alpha, G)$ is called a projective hash family for (X, L) if for every $x \in L$, $hk \in K$, the projection key $s = \alpha(hk)$ uniquely determines $H_{hk}(x)$.

This projection key $s = \alpha(hk)$ only guarantees to evaluate $H_{hk}(x)$ for $x \in L$, and does not guarantee anything for $x \in X \backslash L$. If this is the case for every $\Lambda \in \mathbf{M}$, we say that $\mathbf{H}$ is a projective hash family for a subset membership problem $\mathbf{M}$.

3.4.3 Smooth Projective Hash Families

We now extend the definition to *smooth* projective hash families. The smooth condition is that given a random projection key $s = \alpha(hk)$ and a element $x \in X \backslash L$, the value of the hash function $H_{hk}(x)$ is statistically indistinguishable from random. This means that the projection key reveals (almost) nothing about $H_{hk}(x)$ when $x \in X \backslash L$, but still uniquely determines $H_{hk}(x)$ when $x \in L$.

Definition 3.4.3 (Smooth projective hash family)**.** A projective hash family $(\mathcal{H}, K, S, \alpha, G)$ for a subset membership problem $\mathbf{M}$ is smooth if for every, even maliciously chosen, instance description $\Lambda[X, W, L, R]$ and every $x \in X \backslash L$, the random variables $(\alpha(hk), H_{hk}(x))$ and $(\alpha(hk), \psi)$ are statistically indistinguishable, where $hk \xleftarrow{r} K$ and $\psi \xleftarrow{r} G$.

We say that H_{hk} is a *smooth projective hash function* if $H_{hk} \in \mathcal{H}$, where $\mathcal{H}$ is a part of a smooth projective hash family and $hk \in K$.

3.4.4 Security Game of Smooth Projective Hash Functions

The security of a smooth projective hash function is based on that for an adversary $\mathcal{R}^*$, $\mathcal{R}^*$ should not be able to distinguish between $(\alpha(hk), H_{hk}(x))$ and $(\alpha(hk), \psi)$, for $x \in X \backslash L$, $hk \xleftarrow{r} K$ and $\psi \xleftarrow{r} G$. We set up an attack game for the situation.

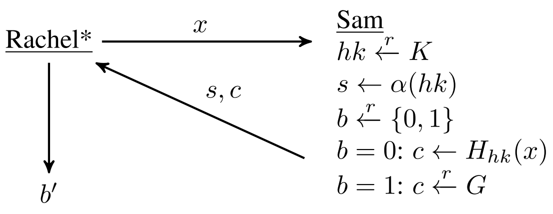

Rachel*
x
s, c
b'
Sam
$hk \xleftarrow{r} K$
$s \leftarrow \alpha(hk)$
$b \xleftarrow{r} \{0,1\}$
$b = 0$: $c \leftarrow H_{hk}(x)$
$b = 1$: $c \xleftarrow{r} G$

Figure 3.4: Smooth projective hash function game with $x \in X \backslash L$.

We define the advantage of $\mathcal{R}^*$ against a smooth projective hash funtion as

$$Adv_{SPHF}(\mathcal{R}^*) = |Pr[b = b'] - \frac{1}{2}|.$$

3.5 Example and Modelling in EasyCrypt

The first thing we will implement in EasyCrypt is a general definition of subset membership problems, smooth projective hash functions and an encryption scheme. We will present some proofs for the properties of subset membership problems and prove the correctness of the encryption scheme. The objective with this is to get used to the notation in EasyCrypt and the logic behind such systems. The code from this section is found in `SMP-SPHF-intro.ec`.

3.5.1 Sets and Functions

The notation and concrete system to be used is this:

- Let L be a proper subset of X.
- Let K be the key space, and $hk \xleftarrow{r} K$
- Let $H_{hk} : X \to G$ be a hash function from X to G.
- Let W be the set of witnesses, and $w \xleftarrow{r} W$.
- Let $s \leftarrow \alpha(hk)$ be the value of the projective key $\alpha(hk)$.
- Let $\pi(w) = x$ be the function to get the element $x \in X$ from the witness w.
- Let $\rho(w, s) = \gamma, \gamma \in G$.

Note that $\rho(w, s) = H_{hk}(\pi(w))$, and that $\rho(w, s)$ is the function which lets one evaluate the hash function by knowing the witness w and the value of the projection key $s = \alpha(hk)$.

3.5.2 Encryption

Encryption Phase 1
At first the receiver decides w and x and sends x to the sender.

- $w \xleftarrow{r} W$
- $x \leftarrow \pi(w)$

Key Generation and Encryption Phase 2
The encryption algorithm, outputting a ciphertext c with input a message M and an $x \in L$, works as follow:

- $hk \xleftarrow{r} K$
- $s \leftarrow \alpha(hk)$
- $a \leftarrow H_{hk}(x)$
- $c \leftarrow (s, a \cdot M)$

Decryption
At last, the receiver uses the decryption algorithm, with input a ciphertext $c = (s, v)$ and gives a message M, which works like this:

- $t \leftarrow \rho(w, s)$
- $M \leftarrow v/t$

3.5.3 Modelling in EasyCrypt

We then model it in EASYCRYPT. First, we have to define the sets:

```
type hkey.
type X.
type G = group.
type W.
type L = X.
type S.
```

One problem is how to define subgroups in EASYCRYPT. This requires both axioms and lemmas. This is how we defined that L is a proper subset of X.

```
axiom Lsubset (s1 : L fset) (s2 : X fset) : s1 < s2.
```

This was used to prove properties with elements from L and $X\backslash L$. The first one stating that $x \in X\backslash L$ is equivalent with $x \in X \wedge x \notin L$.

```
lemma mem1 (xs:X fset, ls:L fset) : forall (x:X),
x \in (xs `\` ls) <=> (x \in xs) /\ !(x \in ls).
proof. smt. qed.
```

Then, $x \in L$ is equivalent with $x \notin X\backslash L$.

```
lemma mem3 (xs:X fset, ls:L fset) : forall (x:X),
x \in ls <=> !(x \in xs `\` ls) by smt.
```

And the last one stating that for an element $x \in X$, either $x \in L$ or $x \in X\backslash L$.

```
lemma mem4 (xs:X fset, ls:L fset) : forall (x:X),
x \in ls \/ x \in (xs `\` ls) by smt.
```

Furthermore, we set up a module to sample elements from X, L, $X\backslash L$ and W at random.

```
module Sampling = {
  proc fromX(xs : X fset) : X = {
    var x;
    x <$ MUniform.duniform (elems xs);
    return x;
  }
  proc fromL(ls : X fset) : L = {
    var x;
    x <$ MUniform.duniform (elems ls);
    return x;
  }
  proc fromXnotL(xs:X fset, ls:L fset) : X = {
    var x;
    x <$ MUniform.duniform (elems (xs `\` ls));
    return x;
  }
  proc fromW(ws:W fset) : W = {
    var w;
    w <$ MUniform.duniform (elems (ws));
    return w;
  }
}.
```

To make sure that these procedures output the elements they are supposed to, we stated lemmas that the elements in fact are in X, L, $X \backslash L$ and W. These lemmas can be found in the Appendix B, and are named `test1, test2, test3, test4`.

Moreover we define the functions as operators. These operators, together with axioms which states the properties of the functions, are used to make the encryption scheme.

```
module PHF : Scheme = {
  proc keygen() : hkey = {
    var hk;
    hk <$ dhkey;
    return hk;
  }
  proc witness() : W * X = {
    var w,x,ws;
    w <- Sampling.fromW(ws);
    x <- fpi w;
    return (w,x);
  }
  proc encrypt(hk:hkey, M:group, x:X) : S * G = {
    var a, s, c;
    s <- falpha hk;
    a <- fk hk x;
    c <- (s, a*M);
    return c;
  }
  proc decrypt(s:S, v:G, w:W) : group option = {
    var t, m;
    t <- frho w s;
    m <- v / t;
    return Some m;
  }
}.
```

The proof for the correctness of the scheme, can be found in the Appendix B.

As mentioned, this implementation was made to experiment with notation and logic in EasyCrypt. Specially the subgroup property was seen as an challenge and important feature for the forthcoming protocol.

Chapter 4

Oblivious Transfer

In this chapter we will define oblivious transfer and present some examples. The security part of an oblivious transfer protocol will be discussed in Chapter 6.

Oblivious transfer was introduced by Rabin in 1981 [6]. He presented a scheme where a sender sends a message to a receiver with a probability of 50%. The sender will be oblivious whether or not the receiver received the message.

An oblivious transfer protocol is a type of a cryptographic protocol which differ from a normal key exchange or encryption scheme. The security is based on that the persons interacting which each other not acquire more information than intended. This is the case for both the sender of the message(s), and the receiver.

4.1 Definition

The idea of oblivious transfer has been developed since Rabin presented his design, and is now used in for example secure multiparty computations. An oblivious transfer protocol is a cryptographic protocol between a sender that has a set of messages, and a receiver who gets to know one or more of the messages. The sender does not know which of the messages that have been obtained by the receiver, and the receiver does not get to know anything about the messages not transferred. This is called k-out-of-n oblivious transfer, where k is the number of messages sent, and n is the number of messages held by the sender. The most common versions of oblivious transfer is of the type 1-out-of-n oblivious transfer, and specially 1-out-of-2 oblivious transfer, which this thesis will focus on. 1-out-of-2 oblivious transfer has been proved to be equal to Rabin's definition [3].

Oblivious transfer is a general definition of a way to communicate securely and the security can be based on several different assumptions. In this thesis we will look at a system which rely on both the Decisional Diffie-Hellman assumption and smooth projective hash functions. Kalai [5] describes that oblivious transfer is ”considered to be the main bottleneck with respect to the amount of computation required by secure multiparty protocols”. This together with the fact that the security can be based on many different assumptions makes oblivious transfer a relevant research field in cryptography.

4.2 1-out-of-2 oblivious transfer

The 1-out-of-2 oblivious transfer is the case where the sender, Sam, has two messages and transfers one of them to the receiver, Rachel, and Sam does not know which of the two messages Rachel has received.

This means that Sam starts with two messages, M_0 and M_1, while Rachel chooses a bit, $b \in \{0, 1\}$, at random. Sam sends M_b to Rachel, but does not want to reveal anything about M_{1-b}. Rachel wants to learn M_b, but does not want Sam to know which of the two messages M_0 and M_1 she receives.

We define the input and the output for an 1-out-of-2 oblivious transfer protocol between a receiver and a sender.

- **Input**
  - Receiver: a bit, $b \in \{0, 1\}$.
  - Sender: two messages, M_0, M_1.
- **Output**
  - Receiver: M_b.
  - Sender: nothing.

As we are going to focus on 1-out-of-2 oblivious transfer, the reader should from now think of the term *oblivious transfer* as 1-out-of-2 oblivious transfer.

4.3 Example Using RSA Encryption

An oblivious transfer protocol using RSA encryption is maybe the easiest example of an oblivious transfer protocol. The sender, Sam, has two messages M_0, M_1 and wants to send one of them to the receiver, Rachel. The protocol works like this:

1. Sam generates an RSA key pair including an $N = pq$ where p and q are large primes, the public key e and the secret key d, which is the inverse of e modulo N. N and e are public values. He then generates two random numbers $y_0, y_1 \overset{r}{\leftarrow} \mathbb{Z}_N$ and sends them to Rachel together with N and e.

2. Rachel picks a bit b at random and a random element r in $\mathbb{Z}_N$. She then computes $v = (y_b + r^e)\ mod\ N$ and sends v to Sam.

3. Sam computes two values $r_0 = (v - y_0)^d$ and $r_1 = (v - y_1)^d$, where $r_b = r$. He then computes $c_0 = M_0 + r_0$ and $c_1 = M_1 + r_1$, and sends both values to Rachel. M_0 and M_1 are the two original messages.

4. Rachel obtains the message $M_b = c_b - r$.

$$
\begin{array}{lcl}
\underline{\text{Rachel}} & & \underline{\text{Sam}} \\
 & & \text{Generate } N = pq \text{ and } e, d \\
b \overset{r}{\leftarrow} \{0,1\} & \xleftarrow{\quad N, e, y_0, y_1 \quad} & y_0, y_1 \overset{r}{\leftarrow} \mathbb{Z}_n \\
r \overset{r}{\leftarrow} \mathbb{Z}_n & & \\
v = (y_b + r^e) \mod N & \xrightarrow{\quad v \quad} & r_0 = (v - y_0)^d \\
 & & r_1 = (v - y_1)^d \\
 & & c_0 = M_0 + r_0 \\
M_b = c_b - r & \xleftarrow{\quad c_0, c_1 \quad} & c_1 = M_1 + r_1
\end{array}
$$

If this protocol is secure, then both the security of the receiver and the sender is preserved. This means that

1. Sam cannot tell which of his elements r_0 and r_1 that equals Rachel's r, and is therefore not able to learn Rachel's choice b.

2. Rachel cannot obtain the other message, M_{1-b}. The value of $c_{1-b} - r$ will give a random value and Rachel cannot learn both messages without knowing r_{1-b}.

4.4 The Basic Idea of an Oblivious Transfer Protocol

In our presented protocol in Chapter 5 we will use some different concepts than the RSA encryption scheme. The protocol will be based on properties presented in [5] and Chapter 3, and we will give a quick overview of how a general version of this oblivious transfer protocol works. The complete protocol will be presented later.

1. Rachel chooses a set X and a subset L of X and computes a bit $b \overset{r}{\leftarrow} \{0,1\}$. Then she samples an element $x_b \overset{r}{\leftarrow} L$ with a witness w_b, and an element $x_{1-b} \overset{r}{\leftarrow} X \backslash L$. She sends the set X and the elements x_0 and x_1 to Sam.

2. Sam generates two hash keys at random $hk_0, hk_1 \overset{r}{\leftarrow} K$, where K is the key space. He computes $c_0 = M_0 \cdot H_{hk_0}(x_0)$ and $c_1 = M_1 \cdot H_{hk_1}(x_1)$, where $H_{hk_0}, H_{hk_1} \in \mathcal{H}$ and $\mathcal{H}$ is an collection of hash functions. The clue now is that Sam also computes two projection keys, $\alpha(hk_0)$ and $\alpha(hk_1)$, which can be used to compute the values $H_{hk_0}(x_0), H_{hk_1}(x_1)$ without knowing the hash functions if one have the witness w_b. Sam sends c_0, c_1, $\alpha(hk_0)$ and $\alpha(hk_1)$ to Rachel.

3. Rachel can now obtain the value of $H_{hk_b}(x_b)$ by using w_b and $\alpha(hk_b)$, and then compute $M_b = c_b / H_{hk_b}(x_b)$.

$$
\begin{array}{lcl}
\underline{\text{Rachel}} & & \underline{\text{Sam}} \\
\text{Choose } X, L & & \\
b \overset{r}{\leftarrow} \{0,1\} & & \\
x_b \overset{r}{\leftarrow} L & & \\
w_b \text{ a witness for } x_b & & \\
x_{1-b} \overset{r}{\leftarrow} X \backslash L & \xrightarrow{\quad X, x_0, x_1 \quad} & hk_0, hk_1 \overset{r}{\leftarrow} K \\
 & & c_0 = M_0 \cdot H_{hk_0}(x_0) \\
H_{hk_b}(x_b) = \alpha(hk)^w & \xleftarrow{c_0, c_1, \alpha(hk_0), \alpha(hk_1)} & c_1 = M_1 \cdot H_{hk_1}(x_1) \\
M_b = c_b / H_{hk_b}(x_b) & &
\end{array}
$$

For a protocol like this to be secure, we need to make sure that the security of both the receiver and the sender is maintained. This means that

1. Sam cannot distinguish the elements x_0 and x_1 and thereby tell which message Rachel receives. This will be preserved by the *subset membership problem*.
2. Rachel cannot get any information about M_{1-b}, and if she tries to obtain this message, she will get a random value. This will be preserved by the concept *smooth projective hash functions*.

Note that this protocol only needs two interactions between the receiver and the sender, while the RSA protocol needs three. The RSA protocol also needs the sender to start the communication. This means that the receiver cannot request a message without the sender transfer his details first, which is not an optimal way of transferring messages.

Chapter 5

Kalai Oblivious Transfer Protocol

We are going to present some cryptographic ideas and a protocol to be implemented in EASYCRYPT, based on the properties presented in the previous chapters. We are also going to present a proof of the correctness of an encryption scheme corresponding to the protocol. Correctness is a verification for the scheme to give the right output, given a permitted input and a successful execution.

The oblivious transfer protocol is based on the protocol presented by Kalai in [5]. The idea of this protocol is the same as the one given by Kalai, but differ in the sets used and the security notion, in which we will use some notions presented in [2].

5.1 Definitions

First, we describe the sets used in Chapter 3 and use this to show encryption and decryption of messages, as in Section 3.5.

5.1.1 Sets and Functions

- Let G be a cyclic group of order q.
- Let $W = \mathbb{Z}_q$ be the set of witnesses.
- Let $X = G \times G$ be the group with elements on the form $(g_0^{m_0}, g_1^{m_1})$, $g_0, g_1 \in G$, $m_0, m_1 \in \{0, ..., q-1\}$.
- Let L be the subset of a group X, where every element in L is on the form (g_0^w, g_1^w).
- Let $K = \mathbb{Z}_q \times \mathbb{Z}_q$ be the key space, where a key $hk \in K$ is on the form (hk_0, hk_1).
- Let $\pi(w) = (g_0^w, g_1^w)$ be the function that takes an element in W and (g_0, g_1), and creates an element $x \in L$.
- Let $H_{hk} : G \times G \to G$ be the hash function, where $H_{hk}(x) = x_0^{hk_0} \cdot x_1^{hk_1}$.

- Let $s = \alpha(hk) = g_0^{hk_0} \cdot g_1^{hk_1}$ be the projection key.
- Let $\rho(s, w) = s^w$ be the function which lets one evaluate $H_{hk}(x)$.

It is easy to see that $\rho(s, w) = s^w = (g_0^{hk_0} \cdot g_1^{hk_1})^w = (g_0^w)^{hk_0} \cdot (g_1^w)^{hk_1} = H_{hk}(\pi(w))$ which makes this a projective hash familiy.

$$\begin{tikzcd}
\text{L} \arrow[r, "H_{hk}(x)"] & \text{G} \\
\text{W} \arrow[u, "\pi(w)"] \arrow[ur, "\rho(s,w)"'] &
\end{tikzcd}$$

Figure 5.1: Relation of the hash function and projection key for $x \in L$.

5.1.2 Encryption

Encryption Phase 1
The encryption of a message, M, is done as in Section 3.5, starting with the receiver:

- $w \overset{r}{\leftarrow} W$
- $x = (g_0^w, g_1^w) \leftarrow \pi(w)$

x is sent to the sender.

Key Generation and Encryption Phase 2
The sender encrypts the message M:

- $hk \overset{r}{\leftarrow} K$
- $s \leftarrow g_0^{hk_0} \cdot g_1^{hk_1}$
- $a \leftarrow x_0^{hk_0} \cdot x_1^{hk_1}$
- $c \leftarrow (s, M \cdot a)$

$c = (s, v)$ is sent to the receiver.

Decryption
The receiver can then decrypt the message:

- $t \leftarrow s^w$
- $M \leftarrow v/t$

5.2 The Oblivious Transfer Protocol

Now, we are going to use the scheme to set up an oblivious transfer protocol. To do this, we use the functions and elements described in the previous section. An oblivious transfer between a receiver, Rachel, and a sender, Sam, is set up like this:

- Rachel and Sam agrees on two elements $g_0, g_1 \in G$.
- Rachel draws randomly two witnesses $w, w' \overset{r}{\leftarrow} W$, $w \neq w'$, and a bit $b \overset{r}{\leftarrow} \{0,1\}$. Then she makes two elements $x_b = (g_0^w, g_1^w) \in L$ and $x_{1-b} = (g_0^w, g_1^{w'}) \in X \backslash L$. x_0, x_1 is sent to Sam.
- Sam makes a pair of hash keys $hk = (hk_0, hk_1), hk \overset{r}{\leftarrow} K$. He then hashes the values from Rachel: $H_{hk}(x_0) = x_{00}^{hk_0} \cdot x_{01}^{hk_1}$ and $H_{hk}(x_1) = x_{10}^{hk_0} \cdot x_{11}^{hk_1}$. These values are used in the encryption of his two messages $M_0, M_1 \in G$: $c_0 = M_0 \cdot H_{hk}(x_0)$ and $c_1 = M_1 \cdot H_{hk}(x_1)$. Furthermore, he calculate $\alpha(hk) = g_0^{hk_0} \cdot g_1^{hk_0}$. Then $c_0, c_1, \alpha(hk)$ are sent to Rachel.
- Rachel uses $\alpha(hk)$ and w to compute the message M_b: $M_b = c_b/\alpha(hk)^w = c_b/(g_0^{hk_0} \cdot g_1^{hk_1})^w = c_b/((g_0^w)^{hk_0} \cdot (g_1^w)^{hk_1}) = c_b/H_{hk}(x_b)$.

This makes Rachel learn one of Sam's messages. Because of the subset membership problem, Sam cannot distinguish x_0 and x_1, and is therefore not able to tell if Rachel will learn M_0 or M_1. The smoothness requirement of the projective hash function maintains the security of Sam, as Rachel will not get any information about M_{1-b}. The security of the protocol will be discussed in the forthcoming chapter.

Rachel		Sam
$w, w' \overset{r}{\leftarrow} W$	x_0, x_1	$hk_0, hk_1 \overset{r}{\leftarrow} K$
$b \overset{r}{\leftarrow} \{0,1\}$	$\longrightarrow$	$hk = (hk_0, hk_1)$
$x_b \leftarrow (g_0^w, g_1^w)$		$H_{hk}(x_0) = x_{00}^{hk_0} \cdot x_{01}^{hk_1}$
$x_{1-b} \leftarrow (g_0^w, g_1^{w'})$		$H_{hk}(x_1) = x_{10}^{hk_0} \cdot x_{11}^{hk_1}$
		$s = \alpha(hk) = g_0^{hk_0} \cdot g_1^{hk_1}$
	s, c_0, c_1	$c_0 = M_0 \cdot H_{hk}(x_0)$
$M_b = c_b/\alpha(hk)^w$	$\longleftarrow$	$c_1 = M_1 \cdot H_{hk}(x_1)$

5.3 Verifying Algorithm

In Section 3.3.2 we presented a set of conditions for a subset membership problem to be verifiably samplable. We look at these four algorithms in order of the Kalai oblivious transfer protocol to make sure that the setup of the protocol is possible and secure.

1. Given G and q, the instance description, Λ, consists of descriptions of $g_0, g_1 \in G$ and $W = \mathbb{Z}_q$.
2. An algorithm to sample a witness $w \overset{r}{\leftarrow} W$ obviously exists. This produces an element $x_0 = (g_0^w, g_1^w) \in L$ which is statistically close to uniform on L if $w \overset{r}{\leftarrow} W$.

3. The same algorithm can produce a second witness $w' \overset{r}{\leftarrow} W, w' \neq w$. This, together with the witness w, produces an element $x_1 = (g_0^w, g_1^{w'}) \in X \backslash L$ which is statistically close to uniform on $X \backslash L$ if $w, w' \overset{r}{\leftarrow} W$. If the previous algorithm sampling w does not work and $x_0 \notin L$, then x_1 will be random in X.

4. To verify that there exists a bit b such that $x_b \in X \backslash L$, the only thing to do is to compare the two entries of each two tuple. An algorithm that takes two elements (x_{00}, x_{01}), (x_{10}, x_{11}) and checks that $x_{00} = x_{10}$ and $x_{01} \neq x_{11}$ exists.

We conclude that the subset membership problem is verifiably samplable.

5.4 Modelling in EasyCrypt

We will now look at the modelling of the protocol in EASYCRYPT, together with a proof of correctness. The implementation of the security will be discussed in Chapter 6. The code is found in the file `KalaiProtocol.ec`.

As EASYCRYPT has only defined one operator in the `CyclicGroup.ec` file, we use this, called $G.g$, as one of the generators for G. The other one is defined as g^m for an m of type `t`. This means that the elements (g_0, g_1) is set as $(G.g, G.g^m)$ in EASYCRYPT, where $G.g$ is the generator found in `CyclicGroup.ec`.

First, we have to define the sets to be used. They are defined straightforward, with both the hash key and the witnesses defined as two tuples. Recall that L is not implemented, and will not be later either. This is because the work with subgroups are difficult and there is no need to specify for EASYCRYPT how L is defined.

```
type hkey = F.t * F.t.
type X = group * group.
type G = group.
type W = F.t * F.t.
type S = group.
```

The functions are defined as operators:

```
op fkop (x:X) (hk:hkey) = (x.`1^hk.`1 * x.`2^hk.`2).
op generatorop (m:t) = (G.g, G.g^m).
op alphaop (x:X) (hk:hkey)
        = (x.`1^hk.`1 * x.`2^hk.`2).
op rhoop (s:S) (w:W) = s^w.`1.
op piop (x:X) (w:W) = (x.`1^w.`1, x.`2^w.`1).
op notpiop (x:X) (w:W) = (x.`1^w.`1, x.`2^w.`2).
```

The `generatorop` is an operator that sets the element (g, g^m). Moreover, all the operators have ending "op" for "operator", and `fkop` is the hash function H_{hk}. The operator `notpiop` is an operator that generates an element $(g^w, g^{mw'}) \in X \backslash L$. To cope with the operators in the proofs, there are lemmas to rewrite every operator. These lemmas can be found in the Appendix C.1. We have also defined the functions as a module, which can be found in the file `KalaiProtocol.ec`, but they are not important as the operators are rather used in the upcoming proofs.

We have constructed a module called `Initialize` which sets and draws some elements, such as the witnesses and hash keys. In addition, it also sets elements in L and $X\setminus L$. This is the same as some of the operators, and the `Initialize` module is only used for the correctness of the encyrption scheme, and can be found in the Appendix C.2. The first thing we implement is an encryption scheme, which looks like this:

```
module type Scheme = {
  proc encrypt(hk:hkey,x:X,g:X,m:group) : S * G
  proc decrypt(s:S,g:G,w:W) : group option
}.

module OT1 : Scheme = {
  proc encrypt(hk:hkey,x:X,g:X,m:group) : S * G = {
    var s,c,a;
    a <- fkop x hk;
    s <- alphaop g hk;
    c <- (s, a*m);
    return c;
  }
  proc decrypt(s:S, g:G, w:W) : group option= {
    var t, m;
    t <- rhoop s w;
    m <- g / t;
    return Some m;
  }
}.
```

To prove the correctness, we had to make a module which takes all the elements and a message, and encypts and decrypts the message:

```
module Correctness1 = {
  proc main(M:G) = {
    var g,w,x0,x1,M',hk,c1,c2;
    g <- Initialize.setg();
    w <- Initialize.drawW();
    x0 <- Initialize.setXinL(g,w);
    x1 <- Initialize.setXnotL(g,w);

    hk <- Initialize.keygen();

    (c1,c2) <- OT1.encrypt(hk,x0,g,M);
    M' <- OT1.decrypt(c1,c2,w);

    return (M' = Some M);
  }
}.
```

Then we prove that the probability of the module `Correctness1` is 1. The proof is found in the Appendix C.3

```
lemma OT_correct1 &m m :
    Pr[Correctness1.main(m) @ &m : res] = 1%r.
```

Then we set up the oblivious transfer protocol. First we have a module to set the elements $g_0, g_1 \in G$, draw the witnesses and the hash keys. It also contains the encryption and decryption algorithms, which are the same as in the previous module. We make sure that the receiver also picks a bit b at random. This bit determines the order of the elements to be sent, i.e. if the receiver sends x_0, x_1 or x_1, x_0.

```
module ObliviousTransfer = {
  proc generator(m:t) : X = {
    return generatorop m; }
  proc witness() : W = {
    var w1,w2;
    w1 <$ FDistr.dt;
    w2 <$ FDistr.dt;
    return (w1,w2);
  }
  proc setX(x:X, w:W) : bool * (X * X) = {
    var x0, x1, b;
    x0 <- piop x w;
    x1 <- notpiop x w;
    b <$ {0,1};
    return (b, b ? (x0, x1) : (x1, x0));
  }

  proc encrypt(hk:hkey,x:X,g:X,M:G) : S * G = {
    var s,c,a;
    a <- fkop x hk;
    s <- alphaop g hk;
    c <- (s, a*M);
    return c;
  }
  proc decrypt(s:S, c:G, w:W) : group option = {
    var t, M';
    t <- rhoop s w;
    M <- c / t;
    return Some M';
  }
}.
```

Then we have a module for the correctness of the protocol, with detailed commentaries. The input is the two messages which is in the possession of the sender. This module describes how the protocol works and in which order, and compares that the message transferred is the same as the one received. The two messages which is in possession of the sender is called M_0 and M_1, while the message obtained by the receiver is called M'.

```
module OT_Correctness = {
  proc main(M0 M1:G) = {
    var m,g,w,x,x1,x2,b,hk,a1,a2,s,M';
```

```
      (* They first agree on the element (g0,g1) *)
      m <$ FDistr.dt;
      g <- ObliviousTransfer.generator(m);

      (* The receiver picks two witnesses *)
      w <- ObliviousTransfer.witness();

      (* And makes two elements,
          one in L and one in X\L, in addition
          to draw a bit b *)
      (b,x) <- ObliviousTransfer.setX(g,w);
      (x1,x2) <- x;

      (* The sender encrypts the messages with both
          elements with a random hash key, s will
          give the same value in both cases *)
      hk <- Initialize.keygen();
      (s,a1) <-
      ObliviousTransfer.encrypt(hk, x1, g, M0);
      (s,a2) <-
      ObliviousTransfer.encrypt(hk, x2, g, M1);

      (* The receiver decrypts the chosen message (b)*)
      M' <-
      ObliviousTransfer.decrypt(s,b ? a1 : a2,w);

      return
      (M' = (b ? Some M0 : Some M1));
    }
  }.
```

The proof of the correctness assures that the probaility of the `OT_Correctness` module is 1, which means that the decrypted message is truly one of the sender's messages. The proof is found in the Appendix C.4, and the input are two messages M_0, M_1

```
lemma ot_correctness &m M0 M1 :
    Pr[OT_Correctness.main(M0,M1)
    @ &m : res] = 1%r.
```

This makes sure that the protocol works, and that the receiver gets the chosen message. The proof does not secure that the receiver only can obtain one message, and that the sender can not distinguish the two elements. These problems will be taken care of in the security chapter.

Chapter 6

Security of Oblivious Transfer and the Kalai Protocol in EasyCrypt

In this chapter, we will describe the general proof for oblivious transfer and then adapt this proof for the protocol described in Chapter 5 in particular, and implement this in EASYCRYPT. The proofs are based on the reasoning described in the articles [2] and [5], where the security of smooth projective hash functions and oblivious transfer are defined. Moreover, as a part of the security of the encryption scheme, it should satisfy the correctness property. This property is proved in EASYCRYPT in the previous chapter.

6.1 Security of Oblivious Transfer

The security of oblivious transfer includes protection of both the receiver and the sender. We will split the definition and look at the security of the receiver and the sender separately. In this section we describe what we mean by security, both for the receiver and the sender.

6.1.1 Security of the Receiver

The security of the receiver is based on indistinguishability. The receiver will see the message M_b, $b \in \{0, 1\}$, and the sender should not be able to learn the value of b. This means that if the receiver picks two random elements, x_b and x_{1-b}, the sender's view in the case when the receiver tries to obtain M_b is indistinguishable from the case when the receiver tries to obtain M_{1-b}. To make this secure, the receiver has to pick the bit b at random to make sure that the sender cannot learn anything about the bit based on previous transfers. In addition x_b and x_{1-b} should be picked from sets which make them difficult to distinguish.

We look at the advantage of a maliciously sender, $\mathcal{S}^*$. For a receiver, with input b, interacting with a maliciously sender, outputting b', the advantage of the sender is

$$Adv_{OT}(\mathcal{S}^*) = |Pr[b = b'] - \frac{1}{2}|$$

$$\begin{tikzcd} {} \arrow[dr, "b"] & & \\ & \mathcal{R} \arrow[rr, leftrightarrow] & & \mathcal{S}^* \arrow[d] \\ & & & b' \end{tikzcd}$$

6.1.2 Security of the Sender

The security of the sender is based on comparison with a trusted third party. The sender wants the receiver to acquire information of one of the two messages, but the receiver should not learn anything about the other message. To obtain this we compare it to an ideal implementation which includes a trusted third party. This trusted third party will have access to both messages of the sender. When the receiver inquires the message M_b, she asks the trusted third party. The trusted third party will give her information only about M_b, and will not tell the value of b to the sender.

In a real implementation of the protocol, without a trusted third party, we require that the receiver does not learn more than in the ideal implementation. This means that the output of the receiver with and without the ideal implementation should be indistinguishable. Moreover, in a secure implementation the receiver will not be able to tell if she interacts with a sender with the messages (M_0, M_1) or one message and a random value, (M_b, ψ), $\psi \overset{r}{\leftarrow} G$.

We look at the advantage of a maliciously receiver, $\mathcal{R}^*$. We want to make this a distinguishing problem for an adversary. To do this, we let the adversary know three messages that can be sent. The challenger will send two of them, and let the adversary guess which two of the original three messages she receives. Let us say that the adversary sends three messages M_{00}, M_{01}, M_1, to the challenger. We now describe an algorithm, called **Sim**:
The challenger draws two bits, $b, \beta \overset{r}{\leftarrow} \{0, 1\}$, and chooses two of the three messages in this way:

- If $\beta = 0$:
  - $M'_0 = M_{0b}$
  - $M'_1 = M_1$
- If $\beta = 1$:
  - $M'_0 = M_1$
  - $M'_1 = M_{0b}$

Then M'_0 and M'_1 are hashed and sent, together with $\alpha(hk)$, to $\mathcal{R}^*$. $\mathcal{R}^*$ can obtain one of the three original messages with $\alpha(hk)$, but will now guess which of the two other messages she has received. We will illustrate this by $\mathcal{R}^*$ guessing a bit b', and she wins if $b = b'$.

$$\begin{tikzcd}
 & \text{sim} \arrow[d, "{M'_0, M'_1, b}"] \\
\mathcal{R}^* \arrow[ur, "{M_{00}, M_{01}, M_1}"] \arrow[r, leftrightarrow] \arrow[d] & \underline{\text{challenger}} \\
b' &
\end{tikzcd}$$

Figure 6.1: The Game with a Maliciously Sender, $\mathcal{R}^*$

As $\mathcal{R}^*$ will obtain the message M_{0b} one half of the times, she will have no problem in guessing b if she wants. While the other half, she will receive M_1 and make a guess if the other message is M_{00} or M_{01}. In addition, we want to avoid the advantage to be negative, and therefore define the advantage as:

$$Adv_{OT}(\mathcal{R}^*) = max(Pr[b = b'] - \frac{3}{4}, 0).$$

6.2 Security of the Kalai Oblivious Transfer Protocol in EasyCrypt

The security of the protocol presented in Chapter 5 is based on the Decisional Diffie-Hellman assumption and the smoothness property of a smooth projective hash function. We will divide the security presentation here as well, and show the implementation in EASYCRYPT.

6.2.1 Security of the Receiver

The security of the receiver is based on the subset membership problem. The hardness of a subset membership problem can be based on different mathematical assumptions. Examples include factoring and the N'th residuosity assumption and the quadratic residuosity assumption. They rely on the fact that it is difficult to decide if an integer $x < N$ is respectively an N'th power or a square modulo N, where $N = nq$, n and q are to unknown, large primes. An other example is the Decisional Diffie-Hellman assumption which the oblivious transfer protocol presented in this thesis is based on.

In the protocol, the subset membership problem is defined by the group X. Moreover, L is the subgroup of X generated by (g, g^m), $g \in G$, $m \in \{0, ..., q-1\}$. The subset membership problem states that it is difficult to distinguish between elements in L and elements in $X \backslash L$. The hardness of the subset membership problem is implied by the Decisional Diffie-Hellman assumption, which states that it is hard to distinguish tuples of the form

$$(g, g^m, g^w, g^{mw'}) \tag{6.1}$$

from tuples of the form

$$(g, g^m, g^w, g^{mw}) \tag{6.2}$$

where g is a generator for G, and $m, w, w' \xleftarrow{r} \mathbb{Z}_q$, $w \neq w'$.

To show that the hardness of the subset membership problem is based on the DDH assumption, which is rather obvious from the definition, we are going to set up a game. The game is between a receiver, Rachel, and a maliciously sender, Sam*. Suppose that we are feeding Sam* with two elements, $x_b \in L$ and $x_{1-b} \in X \backslash L$, and he guesses a bit b'. We set up a game as in Section 3.3.1 for the situation. Recall that x_b is on the form (g^w, g^{mw}), while x_{1-b} is on the form $(g^w, g^{mw'})$.

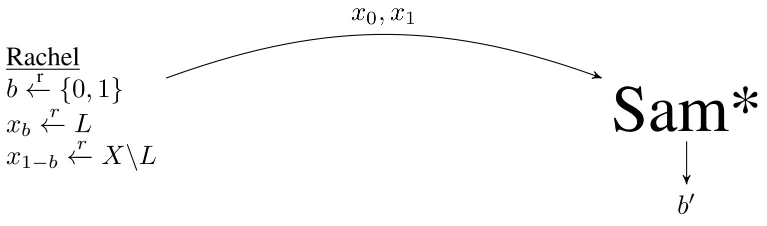

x_0, x_1
Rachel
$b \xleftarrow{r} \{0,1\}$
$x_b \xleftarrow{r} L$
$x_{1-b} \xleftarrow{r} X \backslash L$
Sam*
b'

Figure 6.2: G_0: The Actual Game with an Dishonest Sender, Sam*.

We know from Section 3.3.1 that the advantage of Sam* is

$$Adv_{SMP}(Sam^*) = |Pr[b = b'] - \frac{1}{2}|.$$

Now, we set up another game where Rachel draws a bit at random again, but this time she sets both x_0 and x_1 as random elements from X. The game is found in Figure 6.3.

As x_0, x_1 are randomly chosen from the entire set X, they are not telling anything about b. So the probability for Sam* to guess b correctly is clearly $\frac{1}{2}$. With this, we can make the reduction from our subset membership problem to the DDH assumption. Recall the advantage of an DDH adversary defined in Section 3.2.

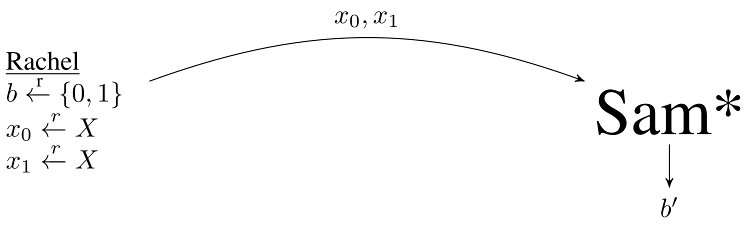


Figure 6.3: G_1: A Game Where x_0, x_1 Are Completely Random.

$$\begin{aligned}
Adv_{SMP}(Sam^*) =& |Pr[b = b'|G_0] - \frac{1}{2}| \\
=& |Pr[b = b'|G_0] - |Pr[b = b'|G_1] \\
=& |Pr[m, w \xleftarrow{r} \mathbb{Z}_q : D(g, g^m, g^w, g^{mw}) = 1] \\
& - Pr[m, w, w' \xleftarrow{r} \mathbb{Z}_q : D(g, g^m, g^w, g^{mw'}) = 1]| \\
=& Adv_{DDH}(D)
\end{aligned}$$

We know that the advantage of DDH is negligible, which means that the advantage of an adversary against our subset membership problem is negligible. This shows that the security of the sender is maintained under the Decisional Diffie-Hellman assumption.

In EASYCRYPT we set up the games G_0 and G_1 as modules and try to reduce it to the DDH assumption. The complete implementation is found in the file `SMP-DDH.ec`. G_0 looks like this:

```
module SMP0 (A:Adversary) = {
  proc main() : bool = {
    var b, m, w1;
    m <$ FDistr.dt;
    w1 <$ FDistr.dt;
    b <- A.guess
    (G.g, G.g ^ m, G.g ^ w1, G.g ^ (m*w1));
    return b;
  }
}.
```

while G_1 looks like this:

```
module SMP1 (A:Adversary) = {
   proc main() : bool = {
     var b, m, w1, w2;

     m <$ FDistr.dt;
     w1 <$ FDistr.dt;
```

```
      w2 <$ FDistr.dt;
      b <- A.guess
      (G.g, G.g ^ m, G.g ^ w1, G.g ^ (m*w2));
      return b;
    }
  }.
```

A file containing the DDH assumption is made by the EASYCRYPT team, but to compare it to G_0 and G_1 we had to make some modifications. The original DDH games can be found in the file `DiffeHellman.ec`, while our DDH games are setup like this:

```
  module DDH0 (A:Adversary) = {
  proc main() : bool = {
    var b, x, y;
    x <$ FDistr.dt;
    y <$ FDistr.dt;
    b <- A.guess
    (G.g, G.g ^ x, G.g ^ y, G.g ^ (x*y));
    return b;
  }
}.

module DDH1 (A:Adversary) = {
  proc main() : bool = {
    var b, x, y, z;

    x <$ FDistr.dt;
    y <$ FDistr.dt;
    z <$ FDistr.dt;
    b <- A.guess
    (G.g, G.g ^ x, G.g ^ y, G.g ^ (x*z));
    return b;
  }
}.
```

At last, we can compare the difference of `SMP0` and `SMP1` to the difference of `DDH0` and `DDH1`. The proof and important modules are found in the Appendix E.

```
   lemma adv_DDH_SMP &m :
     `| Pr[SMP0(A).main()@ &m : res] -
     Pr[SMP1(A).main()@ &m : res] | <=
     `| Pr[DDH0(A).main()@ &m : res] -
     Pr[DDH1(A).main()@ &m : res] |.
```

This proves the security of the receiver, as the subset membership problem based on the DDH assumption, makes sure that the sender cannot distinguish two elements, $x_b \in L$ and $x_{1-b} \in X \backslash L$.

6.2.2 Security of the Sender

We consider the security of the sender in terms of the smoothness requirement of a smooth projective hash function. The variables $(\alpha(hk), H_{hk}(x_{1-b}))$ and $(\alpha(hk), \psi)$ are indistinguishable for a smooth projective hash function, where $hk \xleftarrow{r} K$ and $x_{1-b} \in X \backslash L$. This implies that the variables $(\alpha(hk), H_{hk}(x_{1-b}) \cdot M_{1-b})$ and $(\alpha(hk), \psi)$ are indistinguishable, which again implies that the receiver will not be able to tell if she interacts with a sender with the messages (M_0, M_1) or one message and a random value, (M_b, ψ). Therefore, an oblivious transfer protocol using smooth projective hash functions will be secure in terms of the sender.

We set up a game like in Section 6.1.2, and we look at the advantage of a maliciously receiver, Rachel*, when Sam uses a smooth projective hash function, H. The procedure is the same as described Section 6.1.2, with **Sim** is defined in the same section.

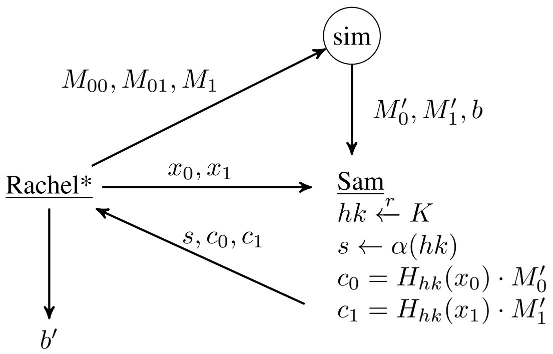


Figure 6.4: The Game with a Maliciously Receiver, Rachel*

We look at the probability of Rachel* to guess the bit b, when c_{1-b} will look randomly.

- Given $x_0 \in L, x_1 \in X \backslash L$:
  $c_0 = H_{hk}(x_0) \cdot M_0'$
  $c_1 \xleftarrow{r} G$
  - $\beta = 0 \implies Pr[b = b' | \beta = 0] \leq 1$
  - $\beta = 1 \implies Pr[b = b' | \beta = 1] = \frac{1}{2}$

$$Pr[b = b'] = Pr[b = b' | \beta = 0] \cdot \frac{1}{2} + Pr[b = b' | \beta = 1] \cdot \frac{1}{2} \leq \frac{1}{4} + \frac{1}{2} = \frac{3}{4}$$

- Given $x_0 \in X \backslash L, x_1 \in L$:
  $c_0 \xleftarrow{r} G$
  $c_1 = H_{hk}(x_1) \cdot M_1'$
  - $\beta = 0 \implies Pr[b = b' | \beta = 0] = \frac{1}{2}$
  - $\beta = 1 \implies Pr[b = b' | \beta = 1] \leq 1$

$$Pr[b = b'] = Pr[b = b'|\beta = 0] \cdot \frac{1}{2} + Pr[b = b'|\beta = 1] \cdot \frac{1}{2} \leq \frac{1}{2} + \frac{1}{4} = \frac{3}{4}$$

As Rachel* can guess wrong on purpose, we have $Pr[b = b'] \leq \frac{3}{4}$.

It now remains to show that the projective hash function actually is smooth. This is the fact that if the receiver picks an element $(x_0, x_1) \in X\backslash L$, she will not be able to determine the hash value from the function $\rho(s, w)$. To see this, we use a second witness, w', and the function

$$\pi'(x, w, w') : W \to X\backslash L \texttt{ where } \pi'(x, w, w') = (x_0^w, x_1^{w'}).$$

The hash function will now give $H_{hk}(x) = (g_0^w)^{hk_0} \cdot (g_1^{w'})^{hk_1}$, $x = (g_0, g_1)$, which cannot be obtained by $\rho(s, w)$, where $s = \alpha(hk) = g_0^{hk_0} \cdot g_1^{hk_1}$ is given.

More formally, the smoothness requirement is:

$$Pr[H_{hk}(x) = g \mid \alpha(hk)] = \frac{1}{q}, hk \xleftarrow{r} K,\ \forall x \in X\backslash L,\ \forall g \in G \tag{6.3}$$

To prove the smoothness requirement in EasyCrypt, we divide the requirement into two smaller lemmas. These two lemmas will help us to prove the smoothness requirement. The first lemma, called Lemma 1, is

$$\forall hk\ \forall t \in G\ \exists hk'\ :\ \alpha(hk) = \alpha(hk') \wedge H_{hk'}(x) = t, x \in X\backslash L. \tag{6.4}$$

While the second lemma, called Lemma 2, is

$$\begin{aligned}&\forall hk\ \forall t \in G\ \forall hk'\ :\\ &\qquad \alpha(hk) = \alpha(hk') \wedge H_{hk}(x) = t = H_{hk'}(x) \implies hk = hk', x \in X\backslash L.\end{aligned} \tag{6.5}$$

To prove these two lemmas in EasyCrypt we had to write a number of other lemmas to be used in the proof, which also makes the proofs of lemma 1 and lemma 2 short. These other lemmas can be found in Appendix D.1 for lemma 1 and in Appendix D.2 and D.3 for lemma 2. The proof of lemma 1 in EasyCrypt looks like this:

```
            (*********** Proof of Lemma 1 ***********)

lemma lemma1 (x nl:X) (hk1:hkey) (t:G) (m:t) (w:W)
: exists hk2, hk1<>hk2 => x = generatorop m =>
nl = notpiop x w => alphaop x hk1 = alphaop x hk2
/\ fkop nl hk2 = t by smt.
```

Here we set x as the element that generates L, namley $(G.g, G.g^m)$ for a m of type t. nl is an element in $X\backslash L$, w is a pair of witnesses and t is an element in the domain G. $hk1$ and $hk2$ are the two pairs of hash keys. Then we formulate the lemma and conditions as in (6.4), and with help from the other lemmas found in Appendix D.1 the SMT provers accept the proof.

The proof of lemma 2 formulated in EasyCrypt looks like this:

```
    (*********** Proof of Lemma 2 ***********)

lemma lemma2 (hk1 hk2 : hkey) (x nl:X) (w:W) (m:t) :
m<>F.zero => x = generatorop m => nl = notpiop x w =>
alphaop x hk1 = alphaop x hk2 =>
fkop nl hk1 = fkop nl hk2  => w.`1<>F.zero =>
w.`2<>F.zero => w.`1<>w.`2 => hk1=hk2 by smt.
```

The variables are stated in the same way as in the proof for lemma 1. In addition we have some extra conditions. To secure that the generating element is not equal to $(G.g, G.g1)$, in which the lemma is not true, we have the condition that $m \neq 0$. We also have the condition that $w \neq (0, 0)$, and that the two witnesses not are equal. These conditions do strictly spoken apply in lemma 1 as well, but are not formulated in EASYCRYPT as they make no difference.

Finally, we prove the smoothness requirement as stated in (6.3). The proof uses a module, as in the proofs for correctness, to implement the variables. As $s = \alpha(hk)$ is given, the possible pairs of hash keys are limited and both values cannot be independently random chosen. To handle this, we only draw one of the hash key values randomly and define the other one so the pair satisfies the given value of s. As we know that $s = g^{hk_0} \cdot (g^m)^{hk_1}$, we set $hk_0 = log_g(s) - hk_1 \cdot m$ and samples hk_1 randomly from K.

- $x_1 \leftarrow (g^w, g^{mw'})$
- $hk_1 \overset{r}{\leftarrow} K$
- $hk_0 \leftarrow log_g(\alpha) - hk_1 \cdot g^{mw'}$

The module, with output $x \in G$, looks like this:

```
module Smoothness = {
  proc main(s:G, w1:t, w2:t, m:t) = {
    var x1,x2,hk0,hk1,x;
    hk1 <$ FDistr.dt;
    x1 <- G.g ^ w1;
    x2 <- G.g ^ m ^ w2;
    hk0 <- log s - hk1*m;
    x <- x1^hk0*x2^hk1;
    return x;
  }
}.
```

We then prove that the probability for the output of the module `Smoothness` to equal an element $n \in G$ is $\frac{1}{q}$. The conditions in the proof is that neither m nor the witnesses w, w' are zero. This only includes a sketch for the proof. The proof is found in Appendix D.5, and for the additional lemmas see Appendix D.4.

```
lemma smoothness &m : forall(n:group), forall(s:G),
forall(w1 w2 m:t), w1<>F.zero => w2<>F.zero =>
m<>F.zero => w1<>w2 =>
```

```
Pr[Smoothness.main(s,w1,w2,m) @ &m : res = n]
= 1%r/F.q%r.
```

This proves the smoothness requirement, and as we have shown, the security of the sender is maintained.

Chapter 7

Concluding Remarks

EasyCrypt do provide new opportunities in cryptographic proofs, and computer aided proofs may be the most secure way to validate protocols. Nevertheless, EasyCrypt has some challenges and critical errors. In this chapter we are going to look at some of the challenges we encountered in EasyCrypt, with the errors, applications and restrictions in the program.

7.1 Critical Error

In our work, we found a critical error while trying to prove smoothness. Our goal in EasyCrypt was to prove that `1 = inv q`, which is obviously not true, however the SMT solvers proved this correct. We explored the problem by separating the statement in a new file in order to test if other false statements could be proved, and to secure that we the SMT provers were not affected by the other lemmas and axioms in the original file. For example, with some rewrite tactics was it easy to prove the false statement `q = 1` from `1 = inv q`:

```
lemma fqeqone : F.q%r = 1%r.
proof.
have : inv F.q%r = 1%r by smt. move => ?.
rewrite -H. smt.
qed.
```

Furthermore, we proved that `q` equaled both two and four, and possibly any other number. The problem was that we could prove `inv q` to be any number, including `q = inv q`.

The problem was reported to the team behind EasyCrypt which made it a high priority and found out that it was a problem with one of the SMT solvers, called Alt-Ergo. To cope with this error we had to remove Alt-Ergo from the files which made a lot of extra work because some of our proofs used Alt-Ergo and had to be proven in an other way, circumventing Alt-Ergo. This problem has now been fixed in the latest version of Alt-Ergo, and should not be a problem in EasyCrypt anymore.

7.2 Stability

The stability of the program was a problem. In some cases, when we turned off the computer, a proof which had been validated did not work when we reopened EASYCRYPT. In addition, we had a problem with finding imported packages in some cases, however restarting the program could sometimes fix this problem. We also experienced trouble with running files on different computers although the versions were corresponding, and some of the files in the EASYCRYPT package was not possible to run or use. This made some trust issues due to restarting the computer and even quit EASYCRYPT.

7.3 Incomplete Selection of Lemmas and Definitions

As the EASYCRYPT team has written all the definitions, lemmas and axioms used in the program there are some challenges using them. The framework does not always provide the needed definitions and axioms. While lemmas could be written by ourselves, axioms and definitions are more risky. An axiom could possibly change the hole state of the program and in the worst case one can introduce an axiom which is wrong, and thereby falsely make all statements true. Therefore, we have chosen to use a minimum of axioms during the proofs and rather tried to prove lemmas. Definitions are also challenging to write as we do not have full information of syntax and how it will influence the state of the program, and it could possibly be a lot to add.

For example, logarithms are difficult to work with in EASYCRYPT as they are only defined with the generator, g, as base. This make calculations with the logarithm of random group elements complicated. Furthermore, inverses are difficult to work with, because in this case, the SMT solvers does not approve simple expressions. For example, to prove that the product of a number and the inverse of another number, both non-negative, is non-zero requires several lines of code:

```
lemma invxy_not_zero (x y:t) : y<>F.zero => x<>F.zero
=> x * inv y <> F.zero.
proof.
move => ? ?. have : x * inv y * y <> F.zero * y.
rewrite mulC.rewrite mulA. rewrite mulC.
rewrite mulA. rewrite mulfVreverse. smt. smt. smt.
qed.
```

7.4 Working with Algebra in EasyCrypt

One of the main problems with the protocol we have implemented, is the algebra. EASYCRYPT is clearly not made for primary doing algebra as the lemmas and axioms in the framework and the SMT solvers are not optimal collaborators. A lot of the mathematical definitions and expressions are defined and proved in a simplified way, only covering special cases. This has resulted in the need for a lot of rewriting and composing of new lemmas from our side. In some lemmas, this algebra has been our main challenge in implementing the proofs.

For example, in EasyCrypt the order of the variables is important as this is the way the lemmas are written. For two integers m, n this can easily be proved: `m*n*1=m*n`. For two group elements a, b, however, it is not that easy to prove `a*b*G.g1=a*b` in EasyCrypt, where `G.g1` is the identity element. This is the proof:

```
lemma idmult (a b:group) : a*b*G.g1=a*b.
proof.
rewrite mulC. rewrite mulA. smt.
qed.
```

As the lemma stating multiplication with the identity element is defined as `G.g1*x=x`, one gets trouble when `G.g1` is not the first element and that `G.g1` is multiplied with two elements, and not one. So the expressions have to be rewritten to prove the statement. With larger expressions this becomes very tedious and requires unnecessary many lines of code, which for example can be seen in Appendix D.3.

7.5 Syntax

The syntax of EasyCrypt is not based on any well-known programming language, which makes it illogical to learn. One of the biggest challenges was to formulate the proofs and know which keywords to use. The reference manual [1] has simple examples and is not finished, so not all tactics are included. The best way to learn the syntax was to look at the examples and theory files and then figure out ourselves how to transfer the syntax to our own proofs. Despite being a problem in the beginning, the framework and syntax becomes more logical as one works with it, but it requires some time to get used to.

7.6 Usage of EasyCrypt in Cryptographic Proofs

The main application of EasyCrypt is setting up games to do reductions and then use this to prove security. We have used EasyCrypt to mainly solve algebra which is inconvenient. The primary application of EasyCrypt is setting up sequences of games, and although the games can be hard to formulate and difficult to prove relations EasyCrypt is a good tool in these cases. The module system is built for games, where one can define adversaries abstractly. This approach is used in Section 6.2.1 when proving the security of the sender in the Kalai oblivious transfer protocol.

When it comes to algebra, the framework actually looks good enough for a lot of cases, but one of the problems is how the SMT solvers use them. Many of the definitions make trouble when applied, such as logarithms and inverses, and adding expressions to each side of the equal sign. The most tedious example of rewriting algebra is found in the Appendix D.3. This makes using algebra a very tedious project, also with simple expressions which are easy to solve by hand.

Nevertheless, using EasyCrypt in these situations prevents the possibility of doing careless mistakes and makes proofs more reliable. So smaller parts with algebra will be acceptable, but we do not recommend using EasyCrypt for a system based on algebra only.

The advantages with computer aided proofs are great and they will make proofs more secure. The team behind EASYCRYPT has developed a good framework, but there are still problems with the program, syntax and the SMT provers that the program uses, and it takes time to learn the program to know. There are other programs with equal application and we recommend to try one of them, although we do not know if they are more usable.

Bibliography

[1] Gilles Barthe, Franois Dupressoir, Benjamin Grgoire, Csar Kunz, Alley Stoughton, and Pierre-Yves Strub. Easycrypt reference manual. https://www.easycrypt.info/documentation/refman.pdf. [Version 1.x compiled 2018-02-19].

[2] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 45–64, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[3] Claude Crépeau. Equivalence between two flavours of oblivious transfers. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 350–354, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[4] Mikkel Furuberg and Morten Solberg. An introduction to easycrypt and the security of the elgamal cryptosystem, 2017.

[5] Yael Tauman Kalai. Smooth projective hashing and two-message oblivious transfer. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 78–95, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[6] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981. https://eprint.iacr.org/2005/187.

[7] 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.

Appendices

Appendix A

Important Axioms and Lemmas in Group Theory

```
(* From CyclicGroup.ec *)

axiom mul_pow g (x y:t):
g ^ x * g ^ y = g ^ (x + y).

axiom pow_pow g (x y:t):
(g ^ x) ^ y = g ^ (x * y).

lemma log_pow (g1:group) x:
log (g1 ^ x) = log g1 * x.

lemma log_mul (g1 g2:group):
log (g1 * g2) = log g1 + log g2.

lemma mulC (x y: group): x * y = y * x.

lemma mulA (x y z: group):
x * (y * z) = x * y * z.

lemma mul1 x: g1 * x = x.

lemma log_g : log g = F.one.

lemma g_neq0 : g1 <> g.

lemma mulN (x:group): x * inv x = g1.


(* From PrimeField.ec *)
```

```
axiom addC (x y:t): x + y = y + x.

axiom addfN (x:t): x + -x = zero.

axiom sub_def (x y:t) : x - y = x + -y.

axiom mulC (x y:t): x * y = y * x.

axiom mulA (x y z:t): x * (y * z) = (x * y) * z.

axiom mulf1 (x:t): x * one = x.

axiom mulfV (x:t): x <> zero => (x * (inv x)) = one.

axiom div_def (x y:t): x / y = x * (inv y).
```

Appendix **B**

Modelling of General Definitions

```
lemma test1 (ls:L fset, xs:X fset) &m :
phoare[Sampling.fromX : true ==> mem xs res] = 1%r.
proof.
proc. auto. move => &hr h. smt.
qed.

lemma test2 (ls:L fset, xs:X fset) &m :
phoare[Sampling.fromL : true ==> mem xs res] = 1%r.
proof.
proc. auto.  smt.
qed.

lemma test3 (ls:L fset, xs:X fset) &m :
phoare[Sampling.fromL : true ==>
!(mem (xs `\` ls) res)] = 1%r.
proof.
proc. auto. smt.
qed.

lemma test4 (ls:L fset, xs:X fset) :
phoare[Sampling.fromXnotL : true ==>
!(mem ls res)] = 1%r.
proof.
proc. auto. smt.
qed.


op dhkey : { hkey distr | is_lossless dhkey /\ is_uniform dhkey } as dhke

axiom dhkeylossless : weight dhkey = 1%r.

op dW : { W distr | is_lossless dW /\ is_uniform dW } as dW_lu.
```

```
axiom dWlossless : weight dW = 1%r.

op hash : hkey -> X -> G.

op fpi : W -> X.

op fk : hkey -> X -> G.

op frho : W -> S -> G.

op falpha : hkey -> S.


axiom prob_pif_eq_rho2 &m (w:W) (s:S) :
  frho w s = (fpi w, fk (fpi w)).     *)

module Correctness = {
  proc main(m:group) : bool = {
    var hk, s, c, m', x, w;
    hk = PHF.keygen();
     (w,x) = PHF.witness();

     (s,c) = PHF.encrypt(hk,m,x);
    m' = PHF.decrypt(s,c,w);

    return (m' = Some m);
  }
}.

lemma PHF_correct &m m :
    Pr[Correctness.main(m) @ &m : res] = 1%r.
  proof.
    byphoare => //. proc. inline*. wp. auto.
    progress. smt. admit. rewrite mulC.
    rewrite (fk_eq_rho v1 (fpi v1) v0 (falpha v0)).
    have : fk v0 (fpi v1) / frho v1 (falpha v0) = g1.
    rewrite (fk_eq_rho v1 (fpi v1) v0 (falpha v0)).
    rewrite divK. smt. move => ?. rewrite -div_def.
    rewrite log_mul. smt.
qed.
```

Appendix C

Initialization of the Kalai Protocol

C.1 Operator Lemmas

```
(* Lemmas used to rewrite the operators
    so algebra can be used *)

lemma generatoroplemma (m:t) : generatorop m =
(G.g, G.g^m) by smt.

lemma pioplemma (x:X) (w:W) : piop x w =
(x.`1^w.`1, x.`2^w.`1) by smt.

lemma fkoplemma (x:X) (hk:hkey) : fkop x hk =
(x.`1^hk.`1 * x.`2^hk.`2) by smt.

lemma alphaoplemma (x:X) (hk:hkey) :
alphaop x hk = (x.`1^hk.`1 * x.`2^hk.`2) by smt.

lemma rhooplemma (s:S) (w:W) : rhoop s w =
s^w.`1 by smt.

lemma notpioplemma (x:X) (w:W) : notpiop x w =
(x.`1^w.`1, x.`2^w.`2) by smt.
```

C.2 Initialization Module

```
(* An initialized module which can be used,
    but has many of the same functions as the
    operators, therefore barely used *)
```

```
module Initialize = {
  proc keygen() : hkey = {
    var hk1,hk2;
    hk1 <$ FDistr.dt;
    hk2 <$ FDistr.dt;
    return (hk1,hk2);
  }
  proc setg() : X = {
    var a:group;
    return (G.g,a);
  }
  proc drawW() : W = {
    var w1,w2;
    var d:t;
    w1 <$ FDistr.dt;
    (* d <> 0;*)
    w2 <- w1+d;
    return (w1,w2);
  }
  proc setXinL(g:X,w:W) : X = {
    var a,b,x1,x2,w1,w2;
    (a,b) = g;
    (w1,w2) = w;
    x1 = a^w1;
    x2 = b^w1;
    return (x1,x2);
  }

  proc setXnotL(g:X,w:W) : X = {
    var a,b,x1,x2,w1,w2;
    (a,b) = g;
    (w1,w2) = w;
    x1 = a^w1;
    x2 = b^w2;
    return (x1,x2);
  }
}.
```

C.3 Correctness of the Encryption Scheme

```
lemma OT_correct1 &m M :
    Pr[Correctness1.main(M) @ &m : res] = 1%r.
proof.
byphoare => //. proc. inline*. auto. progress. smt.
rewrite fkoplemma. rewrite alphaoplemma.
rewrite rhooplemma. rewrite mulC. rewrite mulA.
have : (G.g ^ v ^ v0 * a{hr} ^ v ^ v1) /
```

```
(G.g ^ v0 * a{hr} ^ v1)^v = G.g1.
have <- : (G.g ^ v0 * a{hr}  ^ v1)^v
= (G.g ^ v ^ v0 * a{hr}^v ^ v1). rewrite -pow_mul.
rewrite pow_pow. rewrite pow_pow. rewrite pow_pow.
rewrite pow_pow. smt. rewrite divK. smt. progress.
rewrite -div_def. rewrite log_mul. rewrite log_mul.
have : (log(G.g ^ v ^ v0) + log (a{hr} ^ v ^ v1)) -
log(G.g ^ v ^ v0 * a{hr} ^ v ^ v1) = F.zero.
have <- : (log(G.g ^ v ^ v0) + log (a{hr} ^ v ^ v1)
= log(G.g ^ v ^ v0 * a{hr} ^ v ^ v1)). smt.
rewrite sub_def. rewrite addfN. smt. smt.
qed.
```

C.4 Correctness of the Oblivious Transfer Protocol

```
lemma ot_correctness &m message1 message2 :
    Pr[OT_Correctness.main(message1,message2)
    @ &m : res] = 1%r.
proof.
byphoare => //. proc. inline*. auto. progress. smt.
smt. auto. case (v2). rewrite fkoplemma.
rewrite pioplemma. rewrite generatoroplemma.
rewrite alphaoplemma. rewrite rhooplemma.
have : (G.g, G.g ^ v).`1 = G.g by smt. move => ?.
rewrite H17. have : (G.g, G.g ^ v).`2
= G.g^v by smt. move => ?. rewrite H18.
have : (v0, v1).`1 = v0 by smt. move => ?.
rewrite H19. have : (v3, v4).`1 = v3 by smt.
move => ?. rewrite H20.
have : (v3, v4).`2 = v4 by smt. move => ?.
rewrite H21. have : (G.g ^ v0, G.g ^ v ^ v0).`1
= G.g^v0 by smt. move => ?. rewrite H22.
have : (G.g ^ v0, G.g ^ v ^ v0).`2 = G.g^v^v0 by smt.
move => ?. rewrite H23. rewrite mulC. rewrite pow_pow
pow_pow pow_pow pow_pow. rewrite mul_pow mul_pow.
have : G.g ^ (v3 + v * v4) ^ v0
= G.g^(v3*v0 + v*v4*v0) by smt. move => ?.
rewrite H24. have :
G.g ^ (v0 * v3 + v * (v0 * v4)) /
G.g ^ (v3 * v0 + v * v4 * v0) = g1 by smt. move => ?.
rewrite mulCD. rewrite H25. rewrite mul1. smt.
rewrite generatoroplemma. rewrite pioplemma.
rewrite fkoplemma. rewrite alphaoplemma.
rewrite rhooplemma.
have : (G.g, G.g ^ v).`1 = G.g by smt.
have : (G.g, G.g ^ v).`2 = G.g^v by smt.
have : (v0, v1).`1 = v0 by smt.
have : (v3,v4).`1 = v3 by smt.
have : (v3,v4).`2 = v4 by smt. move => ? ? ? ? ?.
rewrite H17 H18 H19 H20 H21.
have : (G.g ^ v0, G.g ^ v ^ v0).`1 = G.g^v0 by smt.
have : (G.g ^ v0, G.g ^ v ^ v0).`2 = G.g^v^v0 by smt.
move => ? ?. rewrite H22 H23. rewrite mulC.
rewrite pow_pow pow_pow pow_pow pow_pow.
rewrite mul_pow mul_pow.
have : G.g ^ (v3 + v * v4) ^ v0
= G.g ^ (v3*v0 + v*v4*v0) by smt. move => ?.
rewrite H24.
have : G.g ^ (v0 * v3 + v * (v0 * v4)) /
G.g ^ (v3 * v0 + v * v4 * v0) = g1 by smt. move => ?.
rewrite mulCD. rewrite H25. rewrite mul1. smt.
qed.
```

Appendix D

Proof of Smoothness

D.1 Lemmas for lemma 1

```
(* Section for proving lemma 1: For all alpha(hk)
    and for all t in G, there exists a hk` s.t.
    alpha(hk) = alpha(hk`) AND
      H_hk`(x) = t. *)

(* First we prove some lemmas that are
    useful for the lemma *)

lemma aeq1_in_G (a : group) : log(a) = F.zero
<=> a = G.g1 by smt.

lemma exists_m : forall a, exists m, a = G.g ^ m.
proof.
move => ?. exists (log a). smt.
qed.

lemma exists_m2 : forall a, exists m n,
a = (G.g^m,G.g^n) by smt.

lemma exp_g : G.g ^ F.one = G.g by smt.

lemma exists_b (a : group) : exists b,
a <> G.g1 => a ^ b = G.g.
proof.
cut := exists_m a. progress.
exists (inv (log (G.g^m))). progress.
rewrite pow_pow. rewrite log_gpow. rewrite mulfV.
smt. rewrite exp_g. smt.
qed.
```

```
lemma Alpha_exists1 (a:group) (x1 y1 y2:t) :
exists x', x' = x1 + y1 - y2 => y1<>y2 =>
G.g^x1 * a^y1 = G.g^x' * a^y2 /\ x'<>x1 by smt.

lemma Alpha_exists2 (a b:group) (x1 y1 y2:t) :
y1<>y2 => exists x', x' = x1 + y1 - y2 =>
a^x1 * b^y1 = a^x' * b^y2 /\ x'<>x1 by smt.

lemma Alpha_exists3 (a:group) (x1 y1 y2:t) :
y1<>y2 => exists x', x' = x1 + (y1-y2)*log a =>
G.g^x1 * a^y1 = G.g^x' * a^y2 /\ x'<>x1 by smt.

lemma mulfVreverse (x:t) : x<>F.zero =>
((inv x) * x) = F.one by smt.

lemma alpha_existshk (a:group) (hk1:hkey) :
exists hk2, hk1<>hk2 => G.g^hk1.`1 * a^hk1.`2
= G.g^hk2.`1 * a^hk2.`2 by smt.

lemma alpha_existshk2 (a b:group) (hk1:hkey) :
exists hk2, hk1<>hk2 => a^hk1.`1 * b^hk1.`2
= a^hk2.`1 * b^hk2.`2 by smt.

lemma alpha_existshk3 (a b:X) (hk1:hkey) :
exists hk2, hk1<>hk2 => alphaop a hk1
= alphaop b hk2 by smt.

lemma alpha_existsfk (a:group) (hk1:hkey)
(t:G) (w:W) : exists hk2, hk1<>hk2 =>
G.g^hk1.`1 * a^hk1.`2 = G.g^hk2.`1 * a^hk2.`2 /\
G.g^hk2.`1^w.`1 * a^hk2.`2^w.`2 = t by smt.

lemma alpha_existsfk2 (a b:group) (hk1:hkey)
(t:G) (w:W) : exists hk2, hk1<>hk2 =>
a^hk1.`1 * b^hk1.`2 = a^hk2.`1 * b^hk2.`2 /\
a^hk2.`1^w.`1 * b^hk2.`2^w.`2 = t by smt.


        (********* Proof of lemma 1 ************)

lemma lemma1 (x nl:X) (hk1:hkey) (t:G)
(m:t) (w:W) : exists hk2, hk1<>hk2 =>
x = generatorop m => nl = notpiop x w =>
alphaop nl hk1 = alphaop nl hk2 /\
fkop nl hk2 = t by smt.
```

D.2 Lemmas for lemma 2

```
(* Section for proving lemma 2:
    For all alpha(hk), for all t in G,
    for all alpha(hk`), IF alpha(hk) = alpha(hk`)
    AND H_hk(x) = t = H_hk`(x) THEN hk=hk`. *)

(* First a lot of lemmas to be used later *)

lemma move_right (a b c d:t) : a + b = c + d <=>
a = c + d - b by smt.

lemma erase_both_sides (a b c d e f:t) :
a + b*c - b*d + e*b*f*d = a + e*b*f*c <=>
b*c - b*d + e*b*f*d = e*b*f*c by smt.

lemma mul1f (x:t) : F.one * x = x by smt.

lemma move_right2 (a b c d:t) : a - b + c = d <=>
- b + c =  d - a by smt.

lemma mulf1D (x y z:t) : x * z + y * z
= (x + y) * z by smt.


lemma negativex (x z:t) : - x * z = (-x) * z
by algebra.

lemma mulf1D2 (x y z:t) : -x * z + y * z
= (-x + y) * z by smt.

lemma mulf1D3 (a b c d:t) : (-a*b) + c*a*d*b
= ((-a) + c*a*d)*b by algebra.

lemma inv_not_one (x y:t) : x<>y => y<>F.zero =>
(x * (inv y)) <> F.one.
proof.
move => ? ?. have : x * inv y * y <> F.one * y.
rewrite mulC. rewrite mulA. rewrite mulC.
rewrite mulA. rewrite mulfVreverse. smt. smt. smt.
qed.

lemma inv_not_zero (x:t) : x<>F.zero =>
inv x <> F.zero by smt.

lemma invxy_not_zero (x y:t) : y<>F.zero =>
x<>F.zero => x * inv y <> F.zero.
proof.
move => ? ?. have : x * inv y * y <> F.zero * y.
```

```
rewrite mulC. rewrite mulA. rewrite mulC.
rewrite mulA. rewrite mulfVreverse. smt. smt. smt.
qed.

lemma plusnegative (a b c:t) :
a*(b* inv c) + -a = a*(b*inv c) - a by smt.

(* This lemma uses Alt-Ergo *)
lemma factorininv (a b c:t) :
inv a * (a * (b * inv c) - a)
= inv a * a * (b * inv c) - inv a * a.
proof.
smt prover=[+"Alt-Ergo"].
qed.

lemma zerotimes (a:t) : a * F.zero = F.zero by smt.

lemma m_not_zero (a b c:t) :
a<>F.zero => b<>F.zero => c<>F.zero => b<>c =>
(-a) + a * (b * inv c) <> F.zero.
proof.
move => ? ? ? ?. have : b * inv c <> F.zero. smt.
move => ?. have : (-a) = a * (- F.one). smt.
move => ?. rewrite addC. rewrite plusnegative.
have : inv a * ( a * (b * inv c) - a ) <>
inv a * F.zero. rewrite factorininv.
rewrite mulfVreverse. smt. rewrite zerotimes.
rewrite mulC. rewrite mulf1.
have : b * inv c - F.one + F.one <> F.zero + F.one.
have : b * inv c <> F.zero + F.one. rewrite addC.
rewrite addf0. smt. move => ?. smt. smt. smt.
qed.
```

D.3 keqk

```
(* Proof of if the hash function returns the same
value, the to hash keys have to be the same in a
general way, using algebra. This is the base for the
actual proof!!!! *)

lemma keqk (hk1 hk2 hk3:hkey) (m:t) (w:W) :
G.g^hk1.`1 * G.g ^ m ^hk1.`2 =
G.g^hk2.`1 * G.g^m^hk2.`2 =>
G.g^hk2.`1 * G.g^m^hk2.`2 = G.g^hk3.`1 * G.g^m^hk3.`2
=> (G.g^w.`1)^hk2.`1 * (G.g^m^w.`2)^hk2.`2 =
(G.g^w.`1)^hk3.`1 * (G.g^m^w.`2)^hk3.`2 =>
w.`1<>F.zero => w.`2<>F.zero => w.`1<>w.`2 =>
```

```
m<>F.zero => hk2=hk3.
proof.
move => ? ? ? ? ? ? ?. have : hk2.`1 + m*hk2.`2 =
hk3.`1 + m*hk3.`2. have :
log ( G.g ^ hk2.`1 * G.g ^ m ^ hk2.`2)
= log ( G.g ^ hk3.`1 * G.g ^ m ^ hk3.`2). smt.
rewrite log_mul. rewrite log_mul. rewrite pow_pow.
rewrite pow_pow. rewrite log_gpow. rewrite log_gpow.
rewrite log_gpow. rewrite log_gpow. smt. move => ?.
have : hk2.`1 = hk3.`1 + m*hk3.`2 - m*hk2.`2. smt.
move => ?. have : hk2.`2 = hk3.`2. have :
G.g ^ (w.`1 * hk2.`1 + m * w.`2 * hk2.`2)
= G.g ^ (w.`1 * hk3.`1 + m * w.`2 * hk3.`2). smt.
move => ?. have : w.`1*hk2.`1 + m*w.`2*hk2.`2
= w.`1*hk3.`1 + m*w.`2*hk3.`2. smt. have <- :
w.`1*(hk3.`1 + m*hk3.`2 - m*hk2.`2) + m*w.`2*hk2.`2
= w.`1*hk3.`1 + m*w.`2*hk3.`2. smt. move => ?.
have :  w.`1 * (hk3.`1 + m * hk3.`2 - m * hk2.`2)
+ m * w.`2 * hk2.`2
= w.`1*hk3.`1 + m*w.`2*hk3.`2. smt. have :
inv  w.`1*(w.`1*(hk3.`1 + m*hk3.`2 - m*hk2.`2)
+ m*w.`2*hk2.`2)
= inv w.`1*(w.`1*hk3.`1 + m*w.`2*hk3.`2). smt.
rewrite -mulfDl. rewrite -mulfDl. move => ?.
have : inv  w.`1*w.`1*(hk3.`1 + m*hk3.`2 - m*hk2.`2)
+ inv w.`1*m*w.`2*hk2.`2
= inv w.`1*w.`1*hk3.`1 + inv w.`1*m*w.`2*hk3.`2.
smt. rewrite mulfVreverse. smt. rewrite mulC.
rewrite mulf1. rewrite mul1f. move => ?. have :
m * hk3.`2 - m * hk2.`2 + inv w.`1 * m * w.`2
* hk2.`2 = inv w.`1 * m * w.`2 * hk3.`2. have :
hk3.`1 + m * hk3.`2 - m * hk2.`2 +
inv w.`1 * m * w.`2 * hk2.`2
= hk3.`1 + inv w.`1 * m * w.`2 * hk3.`2.
rewrite erase_both_sides. smt. move => ?. smt.
move => ?.
have : - m * hk2.`2 + inv w.`1 * m * w.`2 * hk2.`2
= - m*hk3.`2 + inv w.`1 * m * w.`2 * hk3.`2.
have : m * hk3.`2 - m * hk2.`2 +
inv w.`1 * m * w.`2 * hk2.`2
= inv w.`1 * m * w.`2 * hk3.`2. smt.
rewrite move_right2. smt. move => ?.
have : (( -m) + inv w.`1*m*w.`2)*hk2.`2
= (( -m) + inv w.`1*m*w.`2)*hk3.`2. have :
(- m * hk2.`2) + inv w.`1 * m * w.`2 * hk2.`2
= (- m * hk3.`2) + inv w.`1 * m * w.`2 * hk3.`2.
smt. rewrite mulflD3. smt. move => ?.
have : inv ((-m) + inv w.`1 * m * w.`2) * ((-m) +
inv w.`1 * m * w.`2) * hk2.`2
= inv ((-m) + inv w.`1 * m * w.`2) * ((-m) +
inv w.`1 * m * w.`2) * hk3.`2. smt.
rewrite mulfVreverse. rewrite mulC. rewrite mulA.
rewrite mulC. have : w.`2 * inv w.`1 <> F.one.
have : w.`2 <> w.`1. smt. move => ?. smt.
move => ?. smt. smt. smt.
qed.
```

D.4 Lemmas for smoothness

```
(* Section for proving the smoothness
    requirement with lemma 1 + 2 *)

module M1 = {
  proc main() = {
    var x;
    x <$ FDistr.dt;
    return x;
  }
}.

lemma m1 &m : forall(n:t),
Pr[M1.main() @ &m : res = n] = 1%r/F.q%r.
proof.
progress. byphoare => //. proc. rnd. auto. progress.
rewrite FDistr.dt1E. smt.
qed.

lemma m12 &m (n:t) :
Pr[M1.main() @ &m : res = n] = 1%r/F.q%r.
proof.
byphoare => //. proc. rnd. auto. progress.
rewrite FDistr.dt1E. smt.
qed.

module M2 = {
  proc main() = {
    var x;
    x <$ Dgroup.dgroup;
    return x;
  }
}.

lemma m2 &m : forall(n:Cyclic_group_prime.group),
Pr[M2.main() @ &m : res = n] = 1%r/Prime_field.q%r.
proof.
progress. byphoare => //. proc. rnd. auto.
progress. rewrite mul_def_in. smt.
```

```
qed.

module M3 = {
  proc main() = {
    var x;
    x <$ FDistr.dt;
    return G.g^x;
  }
}.

lemma m3 &m : forall(n:group),
Pr[M3.main() @ &m : res = n] = 1%r/F.q%r.
proof.
move => n. byphoare => //. proc.
conseq (_:_ ==> x = log n / log G.g). progress.
smt. smt. rnd. auto => />. rewrite FDistr.dt1E. smt.
qed.


module M4 = {
  proc main(a:group) = {
    var x;
    x <$ FDistr.dt;
    return a^x;
  }
}.

lemma logone : log G.g1 = F.zero by smt.

lemma m4 &m (a:group) : forall(n:group), a<>g1 =>
Pr[M4.main(a) @ &m : res = n] = 1%r/F.q%r.
proof.
progress. byphoare (_ : arg = a /\ a<>g1 ==> _).
proc. conseq (_:_ ==> x = log n / log a). progress.
rewrite log_pow. rewrite mulC. smt. smt. rnd. auto.
progress. rewrite FDistr.dt1E. smt. smt. smt.
qed.


module M5 = {
  proc main(alpha:G, m:t) = {
    var u,v,w,b,a,x;
    b <$ FDistr.dt;
    u <- G.g;
    v <- G.g ^ m;
    w <- G.log v;
    alpha = G.g^(a+b*w);
    a <- log alpha - b*w;
    x <- u^a*v^b;
    return x;
  }
}.

lemma g1g1 (a b:group): a*b*G.g1=a*b.
proof.
rewrite mulC. rewrite mulA.
have : g1 * a = a by smt. smt.
qed.

lemma g1g12 (a b:int) : a*b*1=a*b by smt.

lemma mulf1inv (x:t) : F.one * x = x by smt.

lemma m5 &m (alpha:G) (m:t): forall(n:group),
m<>F.zero =>
Pr[M5.main(alpha,m) @ &m : res = n] = 1%r/F.q%r.
proof.
progress. byphoare
(_ : arg = (alpha, m) /\ m<>F.zero ==> _). proc.
wp. progress. conseq (_: ==> b = (log n-a)/m).
progress. rewrite log_pow log_pow. rewrite log_mul.
rewrite log_pow log_pow log_pow.
have : log G.g = F.one by smt. move => ?.
rewrite H1. rewrite mulf1inv mulf1inv mulf1inv.
smt. smt. rnd. auto. progress. rewrite FDistr.dt1E.
smt. smt. smt.
qed.


module M7 = {
  proc main(alpha:G, w1:t, w2:t, m:t) = {
    var x1,x2,hk0,hk1,x;
    hk1 <$ FDistr.dt;
    x1 <- G.g ^ w1;
    x2 <- G.g ^ m ^ w2;
    alpha = G.g^(hk0+hk1*m);
    hk0 <- log alpha - hk1*m;
    x <- x1^hk0*x2^hk1;
    return x;
  }
}.

lemma m7 &m (alpha:G) (w1 w2 m:t) :
forall(n:group), w1<>F.zero => w2<>F.zero =>
m<>F.zero =>
Pr[M7.main(alpha,w1,w2,m) @ &m : res = n]
= 1%r/F.q%r.
proof.
progress. byphoare
(_ : arg = (alpha,w1,w2,m) /\ w1<>F.zero /\
w2<>F.zero /\ m<>F.zero ==> _). proc. wp. progress.
conseq (_: ==> hk1 = (log n - w1*hk0)/(m*w2)).
progress. rewrite log_pow. rewrite log_mul.
rewrite log_pow log_pow log_pow log_pow log_pow.
rewrite log_g. rewrite mulf1inv mulf1inv mulf1inv.
have : hk10 * m{hr} - hk10 * m{hr} = F.zero by smt.
move => ?. have : (hk0{hr} + hk10 * m{hr} -
hk10 * m{hr}) = hk0{hr}.
rewrite addC. rewrite sub_def. rewrite addC.
rewrite addA. smt. move => ?. rewrite H6.
have : (w1{hr} * hk0{hr} + m{hr} * w2{hr} *
hk10 - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) = m{hr} * w2{hr} * hk10 /
(m{hr} * w2{hr}) by smt. smt. rewrite log_pow.
rewrite log_g. have : log (G.g ^ w1{hr} ^
(F.one * (hk0{hr} + (log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) * m{hr}) -
(log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) * m{hr}) *
G.g ^ m{hr} ^ w2{hr} ^ ((log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}))) = log n <=>
G.g ^ w1{hr} ^ (F.one * (hk0{hr} +
(log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) * m{hr}) -
(log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) * m{hr}) *
G.g ^ m{hr} ^ w2{hr} ^ ((log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr})) = n by smt. move => ?. apply H5.
rewrite log_mul. rewrite log_pow log_pow log_pow
log_pow log_pow. rewrite log_g.
have : (F.one * (hk0{hr} +
(log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) * m{hr})
- (log n - w1{hr} * hk0{hr}) /
(m{hr} * w2{hr}) * m{hr})
= hk0{hr} by smt. move => ?. rewrite H6.
have : F.one * m{hr} * w2{hr} *
((log n - w1{hr} * hk0{hr}) / (m{hr} * w2{hr}))
= log n - w1{hr} * hk0{hr} by smt. move => ?.
rewrite H7. smt. rnd. auto. progress.
rewrite FDistr.dt1E. smt. smt. smt.
qed.
```

D.5 Proof of smoothness

```
lemma smoothness &m : forall(n:group), forall(s:G),
forall(w1 w2 m:t), w1<>F.zero => w2<>F.zero =>
```

```
m<>F.zero => w1<>w2 =>
Pr[Smoothness.main(s,w1,w2,m) @ &m : res = n]
= 1%r/F.q%r.
proof.
progress.  byphoare (_ : arg = (s,w1,w2,m) /\
w1<>F.zero /\ w2<>F.zero /\ m<>F.zero /\
w1<>w2==> _). proc. wp. progress.
conseq (_: ==> hk1 = (log n - w1*log s)/(m*w2-m*w1)).
progress. rewrite pow_pow pow_pow pow_pow.
rewrite log_mul. rewrite log_pow log_pow.
rewrite log_g. rewrite mulf1inv mulf1inv.
have : (w1{hr} * (log s{hr} - hk10 * m{hr})) =
w1{hr} * log s{hr} - w1{hr} * hk10 * m{hr}
by algebra. move => ?. rewrite H7.
have : (w1{hr} * log s{hr} - w1{hr} * hk10 * m{hr} +
m{hr} * (w2{hr} * hk10) - w1{hr} * log s{hr})
=  - w1{hr} * hk10 * m{hr} + m{hr} * (w2{hr} * hk10)
by algebra. move => ?. rewrite H8. rewrite mulC.
rewrite mulA. rewrite mulC. rewrite addC.
rewrite -sub_def. rewrite mulA. rewrite mulC.
have : (hk10 * (m{hr} * w2{hr}) -
hk10 * (m{hr} * w1{hr})) = hk10 * (m{hr}*w2{hr} -
m{hr}*w1{hr}) by smt. move => ?. rewrite H9. algebra.
smt. rewrite pow_pow pow_pow pow_pow.
have : log (G.g ^ (w1{hr} * (log s{hr} -
(log n - w1{hr} * log s{hr}) /
(m{hr} * w2{hr} - m{hr} * w1{hr}) * m{hr})) *
G.g ^ (m{hr} * (w2{hr} *
((log n - w1{hr} * log s{hr}) /
(m{hr} * w2{hr} - m{hr} * w1{hr}))))) = log n <=>
G.g ^ (w1{hr} * (log s{hr} -
(log n - w1{hr} * log s{hr}) /
(m{hr} * w2{hr} - m{hr} * w1{hr}) * m{hr})) *
G.g ^ (m{hr} * (w2{hr} *
((log n - w1{hr} * log s{hr})
/ (m{hr} * w2{hr} - m{hr} * w1{hr})))) = n.
have : log (G.g ^ (w1{hr} * (log s{hr} -
(log n - w1{hr} * log s{hr}) /
(m{hr} * w2{hr} - m{hr} * w1{hr}) * m{hr})) *
G.g ^ (m{hr} * (w2{hr} *
((log n - w1{hr} * log s{hr}) /
(m{hr} * w2{hr} - m{hr} * w1{hr}))))) = log n.
algebra. rewrite mulC. rewrite mulfDl. smt. smt.
move => ?. apply H7. rewrite log_mul.
rewrite log_pow log_pow. rewrite log_g.
rewrite mulf1inv mulf1inv.
have : (log n - w1{hr} * log s{hr}) /
(m{hr} * w2{hr} - m{hr} * w1{hr}) * m{hr} =
log n / (w2{hr} - w1{hr}) - w1{hr}*log s{hr} /
(w2{hr}-w1{hr}). rewrite mulC. algebra. smt.
```

```
move => ?. rewrite H8. algebra. smt. rnd. auto.
progress. rewrite FDistr.dt1E. smt. smt. smt.
qed.
```

Appendix E

Hard Subset Membership Problem Reduction

```
require import AllCore FSet DBool Bool Int
Real Distr.
require (*  *) CyclicGroup.

clone export CyclicGroup as G.

require RndExcept.

axiom gt1_q : 1 < F.q.

theory Ad1.

  clone import RndExcept as RndE with
    type input <- unit,
    type t     <- F.t,
    op   d     <- fun _ => FDistr.dt,
    type out   <- bool
    proof *.
    realize d_ll.
    move=> _;apply FDistr.dt_ll. qed.

  clone include Adversary1_1 with
    op n <- F.q
    proof *.
  realize gt1_n by apply gt1_q.
  realize d_uni by move=> _ x;apply FDistr.dt1E.

end Ad1.
```

```
theory SMP.

module type Adversary = {
  proc guess(g gm gw1 gmw:G.group): bool
}.

  module SMP0 (A:Adversary) = {
    proc main() : bool = {
      var b, m, w1;
      m <$ FDistr.dt;
      w1 <$ FDistr.dt;
      b <- A.guess
      (G.g, G.g ^ m, G.g ^ w1, G.g ^ (m*w1));
      return b;
    }
  }.

  module SMP1 (A:Adversary) = {
    proc main() : bool = {
      var b, m, w1, w2;

      m <$ FDistr.dt;
      w1 <$ FDistr.dt;
      w2 <$ FDistr.dt;
      b <- A.guess
      (G.g, G.g ^ m, G.g ^ w1, G.g ^ (m*w2));
      return b;
    }
  }.

module DDH0 (A:Adversary) = {
  proc main() : bool = {
    var b, x, y;
    x <$ FDistr.dt;
    y <$ FDistr.dt;
    b <- A.guess
    (G.g, G.g ^ x, G.g ^ y, G.g ^ (x*y));
    return b;
  }
}.

module DDH1 (A:Adversary) = {
  proc main() : bool = {
    var b, x, y, z;

    x <$ FDistr.dt;
    y <$ FDistr.dt;
    z <$ FDistr.dt;
    b <- A.guess
    (G.g, G.g ^ x, G.g ^ y, G.g ^ (x*z));
```

```
        return b;
      }
    }.

      section PROOFS.

      declare module A:Adversary.

      axiom A_ll : islossless A.guess.

      local module Addh0 : Ad1.ADV = {
        proc a1 () = { return ((), F.zero); }
        proc a2 (x:t) = {
          var b, y;
          y <$ FDistr.dt;
          b <- A.guess
          (G.g, G.g ^ x, G.g ^ y, G.g ^ (x*y));
          return b;
        }
      }.

      local module Addh1 = {
        proc a1 = Addh0.a1
        proc a2 (x:t) = {
          var b, y, z;
          y <$ FDistr.dt;
          z <$ FDistr.dt;
          b <- A.guess
          (G.g, G.g ^ x, G.g ^ y, G.g ^ (x*z));
          return b;
        }
      }.

      local lemma a1_ll : islossless Addh0.a1.
      proof. proc;auto. qed.

      lemma adv_DDH_SMP &m :
          `| Pr[SMP0(A).main()@ &m : res] -
          Pr[SMP1(A).main()@ &m : res] | <=
          `| Pr[DDH0(A).main()@ &m : res] -
          Pr[DDH1(A).main()@ &m : res] |.
      proof.
        have <- :
        Pr[Ad1.Main(Addh0).main() @ &m : res]
        = Pr[SMP0(A).main() @ &m : res].
        + by byequiv => //;proc;inline *;sim;auto.
        have <- :
        Pr[Ad1.Main(Addh1).main() @ &m : res]
        = Pr[SMP1(A).main() @ &m : res].
        by byequiv => //;proc;inline *;sim;auto.
        have <- :
        Pr[Ad1.Main(Addh0).main() @ &m : res]
        = Pr[DDH0(A).main() @ &m : res].
        + by byequiv => //;proc;inline *;sim;auto.
        have <- :
        Pr[Ad1.Main(Addh1).main() @ &m : res]
        = Pr[DDH1(A).main() @ &m : res].
        by byequiv => //;proc;inline *;sim;auto. smt.
      qed.

    end section PROOFS.
```