Universidade do Minho
Escola de Engenharia

Henrique Manuel Fernandes de Castro

Formal Verification of Security Proofs

Setembro de 2010

Universidade do Minho
Escola de Engenharia

Henrique Manuel Fernandes de Castro

Formal Verification of Security Proofs

Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Bacelar Almeida

Setembro de 2010

Declaração

Nome: Henrique Manuel Fernandes de Castro

Endereço Electrónico: hmfdcastro@gmail.com

Telefone: 253415224

Cartão de Cidadão: 12833532

Título da Tese: Formal Verification of Security Proofs

Orientador: Professor Doutor José Bacelar Almeida

Ano de conclusão: 2010

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 1 de Setembro de 2010

Henrique Manuel Fernandes de Castro

Abstract

To conduct a security proof, the meaning of the term security must be precisely defined. Provable security aims to formalize security notions by defining both the adversarial model and the adversarial goal. A proof of security is then achieved by reducing an attack against a cryptographic scheme to the intractability of some computational problem. These proofs can be organized as sequences of games thus diminishing their complexity. In this approach adversaries are probabilistic polynomial-time Turing machines and its interaction with the cryptographic scheme is modeled as a probabilistic game. In this thesis we shall explore a framework that formalizes such approach, more precisely, we will explore David Nowak's game-based framework for cryptographic schemes' security proofs. The great share of work developed during the course of this thesis was concerned with the extension of the framework with capabilities to perform proofs in the so called random oracle model and the use of these capabilities to formalize the semantic security proof of Hashed ElGamal in this model. Also, we explore Coq's extension Ssreflect to show how its libraries can be used to support the development of cryptographic proofs.

Verificação Formal de Provas de Segurança

Resumo

Para se poder realizar uma prova de segurança, é preciso primeiro definir o que é que o termo segurança significa. Provable security tenta formalizar noções de segurança ao definir tanto o modelo do adversário como o objectivo do adversário. Uma prova de segurança é então alcançada por redução de um esquema criptográfico a uma problema computacional considerado intratável. Provas podem ser organizadas como sequências de jogos diminuindo então a complexidade destas. Nesta abordagem os adversários são máquinas de Turing que executam em tempo polinomial e a sua interacção com o esquema criptográfico é modelado como um jogo probabilístico. Nesta tese será explorada a framework que formaliza tal abordagem, mais precisamente, irá ser explorada a framework de David Nowak para provas de segurança de esquemas criptográficos baseadas em jogos. A grande parte do trabalho desenvolvido durante a tese teve como objectivo extender a framework com capacidades para realizar provas no chamado modelo de oráculo aleatório e esta extensão será usada para formalizar a prova de segurança semântica do esquema criptográfico Hashed ElGamal neste modelo. Também será explorado a extensão do Coq denominada Ssreflect para mostrar como as suas bibliotecas podem ser usadas para suportar o desenvolvimento de provas criptográficas.

Contents

List of Figures

Chapter 1

Introduction

As any proof, a security proof should convince its reader of its validity. Therefore, these should be constructed in a clear and concise way. However, the term security can be somewhat ambiguous possessing several meanings depending on the context. In order to carry out a precise mathematical cryptographic proof, security must be precisely defined so that the notion it tries to prove cannot raise different interpretations.

The term provable security was first coined by Goldwasser and Micali [GM82] and the idea behind it is to formally define security notions for cryptographic problems [Bel98]. This security notion has at its core the definition of both the adversarial goal and the adversarial model. The adversarial goal captures what it means to break the cryptographic scheme while the model describes the capabilities that the adversary possesses in order to succeed [Sma05]. This allows the mathematical specification of the scheme's security under the chosen definition. As a result, a security proof only ensures security against attacks in the chosen model and not against other types of attacks (for example, side channel attacks).

The security of the scheme is achieved by reducing it to the intractability of some computational problem, for example, the computation of discrete logarithms in finite fields. If the problem is indeed intractable, then it is proved that the scheme is secure under the chosen definition.

However, cryptographic proofs have been conducted with several different approaches and a standardized model to construct these kind of proofs was lacking. Bellare and Rogaway [BR06] add that many cryptographic proofs had become unverifiable mostly due to their complexity while Halevi [Hal05] affirms that there is a problem with cryptographic proofs, mainly as a result of a greater number of proofs generated than proofs carefully verified resulting in some erroneous cryptographic proofs [Sho01].

Game-playing is a technique that intends to solve these problems by structuring proofs and creating a unified development model for cryptographic proofs that makes them less error-prone and more easily verifiable [BR06]. In this approach the adversary is a probabilistic process and is modeled as a polynomial-time Turing machine while its interaction with the cryptographic primitive is modeled as a probabilistic game. The notion of security is usually modeled as a particular event S that occurs inside the game and is connected to the breaking of the cryptographic primitive by the adversary.

Security is then achieved by proving that the probability of an adversary winning the game, that is the probability that the security event S occurs, is negligibly close to a target probability. A function $\mu(n)$ is called a negligible function in the security parameter n if for every polynomial $p(.)$ and for large enough n it holds that:

$$|\mu(n)| < \frac{1}{p(n)}$$

Claiming that this probability is negligible close to a target probability does not imply that the adversary bears no possibility of breaking the cryptographic primitive. Instead, one achieves a game-based proof of security by showing that the advantage the adversary has in breaking the cryptographic primitive over an uninformed adversary is negligible.

Games can also be seen as code and this approach is called code-based game playing [KR96]. In this approach, game-playing centers around making disciplined transformations to code that rely on programming language theory. By taking a code-centric view of games, code-based game-playing prevents the occurrence of subtle mistakes in the development of proofs since every transformation must be formalized as code. These games can also be machine checked, for example in the proof assistant Coq, adding a significant layer of trust to these proofs as a result of the proof assistant correct implementation.

[Hal05] emphasized the importance of a tool that could deal with the "mundane" parts of cryptographic game-based proofs. This tool would help automatize the proof and the creative part of the proof could be entrusted to the user. The so called "mundane" parts consist of proof steps that are usually the most difficult to write and verify and can be of the following type:

- simple transformations like code movement, variable substitution and elimination of code that do not affect the output of the game, also called as dead code.
- algebraic manipulations. This kind of transformations can be obtained if the tool possesses a library where these transformations are formalized and proved correct.
- transformations by mean of reductions where the reduction itself can also be formalized as a game sequence.

Additionally, Halevi proposed that this tool should allow impermissible transformations in proofs. These transformations are not checked by the tool and as a consequence the user is responsible for supplying a free-text justification of the transformation.

1.1 Security Notions and Proofs for Public-Key Schemes

Public-Key Encryption Schemes A public-key encryption scheme $\mathcal{E}$ consists of two probabilistic polynomial time (PPT) algorithms for key-generation and encryption, and a deterministic polynomial time algorithm for decryption. The key-generation algorithm $\mathsf{KeyGen}(1^k)$ takes as input a security parameter[1] k and generates the public-key/secret-key pair (pk,sk). The encryption algorithm $\mathsf{Enc}(pk,m)$ takes as input the public key and a message m and returns the ciphertext ψ. The decryption algorithm $\mathsf{Dec}(sk,\psi)$ takes the secret key and the ciphertext ψ, returning the decrypted message m'.

The reason for not letting the decryption algorithm be probabilistic steams from the following *correctness* criteria: in any public-key encryption scheme one expects that decryption undoes encryption. To formally capture this property, consider the following probabilistic experiment (or *game*):

```
$\textsc{Correct}(k) \doteq$
    $(pk,sk) \leftarrow \mathsf{KeyGen}(1^k)$
    $m \overset{\$}{\leftarrow} \mathsf{MessSpace}$
    $\psi \leftarrow \mathsf{Enc}(pk,m)$
    $m' \leftarrow \mathsf{Dec}(sk,\psi)$
    return $(m = m')$
```

In this thesis, to express variable assignment the symbol "$\leftarrow$" will be used while random assignments to variables shall be conveyed with the symbol "$\overset{\$}{\leftarrow}$". The game starts by the generation of a valid key-pair, then a randomly chosen message is encrypted, and the correspondent ciphertext is decrypted. The result of the

[1]For technical reasons regarding its complexity class, it receives the security parameter k as a string of k bits.

```
$\mathsf{KeyGen}(1^k) \doteq$
    $(\mathbf{G}, \gamma, q) \leftarrow \mathcal{GP}(1^k)$
    $x \xleftarrow{\$} \mathbb{Z}_q$
    $\alpha \leftarrow \gamma^x$
    $sk \leftarrow ((\mathbf{G}, \gamma, q), x)$
    $pk \leftarrow ((\mathbf{G},\gamma, q), \alpha)$
    return $(pk,sk)$

$\mathsf{Enc}(pk, m) \doteq$
    $((\mathbf{G},\gamma,q),h) \leftarrow pk$
    $r \xleftarrow{\$} \mathbb{Z}_q$
    $c_1 \leftarrow \gamma^r$
    $c_2 \leftarrow m \cdot h^r$
    return $(c_1,c_2)$

$\mathsf{Dec}(sk, c) \doteq$
    $((\mathbf{G},\gamma,q),x) \leftarrow sk$
    $(c_1, c_2) \leftarrow c$
    $m \leftarrow c_2 \cdot c_1^{-x}$
    return $m$
```

Figure 1.1: ElGamal Encryption Scheme

game is the outcome of the boolean test that checks if the decrypted message is the one that had been previously encrypted. Clearly, in a correct encryption scheme this game shall always return true. Denoting by $\textsc{Correct}(k) \Rightarrow \mathsf{true}$ the event that running the game its outcome is true, the correctness of the scheme is characterized by the following assertion:

$$\Pr[\textsc{Correct}(k) \Rightarrow \mathsf{true}] = 1.$$

ElGamal Encryption Scheme ElGamal [Gam85] is a probabilistic public-key encryption scheme and its definition is given in Figure 1.1. In the key-generation algorithm, $\mathcal{GP}(1^k)$ randomly chooses a cyclic group $\mathbf{G}$ with generator γ and order q (k-bit long). In the encryption algorithm, m is any element of $\mathbf{G}$. The ElGamal scheme can be proved semantically secure by reducing it to the Decisional Diffie-Hellman (DDH) assumption. This assumption is based on the problem of distinguishing triples of the form $(\gamma^x,\gamma^y,\gamma^{xy})$ and $(\gamma^x, \gamma^y, \gamma^z)$, where x, y, z are random elements of $\mathbb{Z}_q$. This problem is conjectured to be difficult, which means that for any adversary A its advantage defined by:

$$\left|\Pr[x,y \xleftarrow{\$} \mathbb{Z}_q : A(\gamma^x,\gamma^y,\gamma^{xy}) = 1] - \Pr[x,y,z \xleftarrow{\$} \mathbb{Z}_q : A(\gamma^x,\gamma^y,\gamma^z) = 1]\right|$$

is negligible.

Semantic Security The term semantic security, also known as ciphertext indistinguishability, was first introduced in [GM82] and it captures the notion that an efficient adversary, given the ciphertext and the corresponding public encryption key, cannot obtain any extra information about the message encrypted. The adversary is a passive one and if a scheme is semantically secure then the encrypted messages are regarded as indistinguishable.

In the game of semantic security the adversary is modeled as two algorithms A_1 and A_2. These algorithms are deterministic and efficient (i.e. polynomial time). The game is initialized by calling the key-generation algorithm, outputting the public and secret keys. The public key and a random seed are passed to A_1 which returns the pair of messages (m_1,m_2). The encryption algorithm receives one of the messages chosen randomly and the resulting ciphertext is passed to A_2, that, with the knowledge of the public key and the random seed, tries to guess which one was indeed encrypted. Formally, the game can be defined as follows :

```
IND-CPA$(k) \doteq$
    $(pk,sk) \leftarrow \mathsf{KeyGen}(1^k)$
    $r \xleftarrow{\$} \mathbf{R}$
    $(m_1,m_2) \leftarrow A_1(r,pk)$
    $b \xleftarrow{\$} \{0,1\}$
    $\psi \leftarrow \mathsf{Enc}(pk,m_b)$
    $\hat{b} \leftarrow A_2(r,pk,\psi)$
    return $(b = \hat{b})$
```

Since there are only two messages, it should be noted that a truly random algorithm would succeed in guessing the encrypted message with probability $\frac{1}{2}$. Therefore, an encryption scheme is semantically secure if the adversarial probability in winning the game is not significantly greater than $\frac{1}{2}$. The advantage of an adversary is defined as:

$$\mathbf{Adv}_{A}^{\text{IND-CPA}}(k) = \left| \Pr\left[\text{IND-CPA}(k) \Rightarrow \mathsf{true}\right] - \frac{1}{2} \right|$$

A scheme is said to be semantically secure if the advantage of any adversary is negligible with respect to the security parameter k.

Security Proofs Let $\Pr[S_i]$ denote the probability that the event S occurs in game G_i where $i = 0...n$. The initial game G_0 depicts the attack game between the adversary and the cryptographic primitive. To develop the proof, one needs to incrementally refine G_0, thus producing a finite chain of games, until it achieves G_n where $\Pr[S_n]$ can be bounded and is trivially negligibly close to the target probability. Since the number of games is constant and every transition from consecutive games is proved to be negligible, that is the difference between $\Pr[S_i]$ and $\Pr[S_{i+1}]$ is negligible, $\Pr[S_0]$ is negligibly close to the target probability. Each game transformation should be as simple as possible so that each step can be easily analyzed [Sho04]. This prevents the growth of the proof complexity.

The transitions between games can be of three different kinds [Sho04]:

- *Transitions based on indistinguishability* are changes made in a game that are indistinguishable by the adversary. These are normally based on statistical or computational assumptions and the adversary's ability to distinguish between the different games would imply the existence of an efficient algorithm to differentiate between two distributions that are believed to be indistinguishable. For example, the proof of ElGamal semantic security uses the computational intractability of DDH as a transition based on indistinguishability.

- *Transitions based on failure events* occur between identical-until-*bad* games. These games are identical except when some failure event happens setting the *bad* flag to true. After *bad* has been set to true it cannot be reset to false. As an example it can be seen that game Game 0 and Game 1 are identical-until-*bad* games:

```
Game0 $\doteq$
    $y \overset{\$}{\leftarrow} \{0,1\}^n$
    if $y \in \text{img}(x)$ then bad $\leftarrow$ true
    while $y \in \text{img}(x)$ do $y \overset{\$}{\leftarrow} \{0,1\}^n$
    return $y$
```

```
Game1 $\doteq$
    $y \overset{\$}{\leftarrow} \{0,1\}^n$
    if $y \in \text{img}(x)$ then bad $\leftarrow$ true
    return $y$
```

  The difference in probability between the two games can be bounded using the Difference Lemma. Let A and B be a pair of identical-until-*bad* games and let F denote a failure event. The Difference Lemma states that:

$$|Pr[A] - Pr[B]| \leq Pr[F]$$

  This shows that to transition from A to B it is simply needed to know the probability of F occurring.

  As a result, in the previous example the difference in probability between G_0 and G_1 is less or equal than the probability that y belongs to the image of x. In the asymptotic model, to prove that game G_0 is negligibly close to G_1 it suffices to prove that the occurrence of F is negligible.

- *Bridging steps* have no effect on the probability, that is $\Pr[S_i] = \Pr[S_{i+1}]$. They simply alter some steps of the game in a way that is equivalent. Despite the fact that a proof can be conducted without the use of bridging steps, these are useful in structuring the proof allowing it to be easily followed.

So far, the security model implicitly used has been the asymptotic one. In this model, reductions between games only need to be proved negligible according to the security parameter. Besides the asymptotic one, there also exists the exact security model where the cost of reductions must be precisely determined thus allowing a more accurate characterization of the adversarial advantage in breaking the cryptographic primitive.

Random Oracle Model The random oracle model is a computational model that was introduced by Bellare and Rogaway [BR93]. This model tries to bridge theory and practice by providing an efficient way of designing protocols. Unlike the standard way of designing protocols, where one first designs a protocol and then over the time one tries to find a successful attack and while this does not happen the protocol is considered secure, in the random oracle model one proves the security of the protocol. This proof is done in an idealized model but even so this approach is practically more efficient than waiting for the appearance of successful attacks. The ROM methodology has been successfully applied in the design of PSS [BR96] and OAEP [BR94] .

1.2 The Coq Proof Assistant

The Coq system is a computer tool for verifying theorem proofs that is powerful and expressive both for reasoning and programming [BC04]. Coq uses a typed λ-calculus, the Calculus of Inductive Constructions (**CIC**) [CH88] as its formalism. One important property of the Coq system is that it possesses strong normalization [Wer94], meaning that computation always terminates.

To check the correctness of proofs, Coq uses a small checking kernel that implements the typing rules of CIC. Proof terms are built by using tactics that use the proof context, for example, hypotheses or previously proved lemmas, to achieve the proof of certain lemma. The use of tactics eases the construction of proofs and Coq allows user to code new tactics by using Ltac. Ltac is the tactic language for Coq that was developed with intent to enrich the existent tactics combinators and to provide proof automation [Del00], thus preventing the user from making fatal mistakes. The specification language used by Coq (called Gallina) allows to develop mathematical theories and to prove specifications of programs [dt09].

As a result of these properties, Coq is well suited in environments where absolute trust is a necessity such as in cryptographic proofs. The generic model and random oracle model were formalized in Coq [BCT04] with the security proof of ElGamal encryption but the game-playing approach was not used. Later, in both [Now07] and [BGB09] game-based frameworks for cryptographic proofs were developed on top of Coq. These approaches will be explained in-depth in Chapter 2.

1.3 Outline

Most part of this thesis' work was devoted to extending David Nowak's framework for game-based security proofs with capabilities to perform proofs in the random oracle model. The motivations for such implementation are:

- Study of the game-based approach to cryptographic security proofs and study of a framework formalized on top of an interactive theorem prover (Coq)
- Employ such knowledge to the development of proofs in a different security model (random oracle model).

This involved, at an initial stage, the exploration of the different existing game-based frameworks. In Chapter 2 these frameworks are analysed and a comparison of the different approaches to the formalization of the game-based methodology is provided. The proof of ElGamal's semantic security is given in each framework as a running example.

Nowak's framework is given an in-depth look in Chapter 3. The framework has as its core the formalization of probabilities and the formalization of the game-based methodology. The probabilistic nature of the framework is formalized with recourse to the definition of a distribution monad, while games are defined as functions returning a distribution. Some automatic tactics to deal with bridging steps are also defined. The framework is supported by the proof of several mathematical results and we shall give more emphasis to its formalization of group theory. It is also demonstrated how the framework can be used in practice by providing the proof of semantic security of Hashed ElGamal in the standard model.

The implementation of the random oracle methodology is examined in Chapter 4. The state monad defined by David Nowak is used to define the stateful version of the game-based approach and the fundamental lemma of game-playing is formalized along with the explanation of our contributions. The chapter is concluded by demonstrating the implemented capabilities with the proof of Hashed ElGamal's semantic security in the random oracle model.

Besides the work mentioned before, one of the purposes of this thesis is the study of Ssreflect. This is an extension to Coq which adds support for the use of small scale reflection and also provides general purpose features. Ssreflect is explored in Chapter 5. We use Ssreflect's libraries, composed with a vast number of mathematical results, to adapt Hashed ElGamal's proof of security in Coq into Ssreflect in order to take advantage of Ssreflect's features and libraries. This is illustrative of how Ssreflect's libraries can be used and of the benefits of their application. The conclusions and future work are given in Chapter 6.

Chapter 2

State of the art

In this chapter the existing game-based security proofs frameworks are analysed. David Nowak's framework is explored in Section 2.1, CertiCrypt in Section 2.2 and CryptoVerif in Section 2.3. For each framework we will provide both strong and weak points along with the comparison between themselves. This comparison will be illustrated with the running example of the semantic security proof of the ElGamal cryptosystem.

2.1 David Nowak's Framework

Nowak [Now07, Now09] formalizes in Coq a framework for game-based security proofs. In this framework both oracles and games are probabilistic algorithms and are modeled as functions returning finite probability distributions. To model probabilistic choices, such as random value assignments to variables, Nowak uses monads since probabilistic choices can be seen as side effects. The choice of Coq, or any proof assistant for that matter, as the implementation language is an important property of the framework. By using a proof assistant, game specification and game transformations are certified in the sense that in order to trust the proof of cryptographic scheme it suffices to trust the proof assistant correctness. In Coq's case, its correctness is supported by a small proof checking kernel. Also, the use of a proof assistant implies that games must be given a precise mathematical meaning and every step of a proof, even the smallest and trivial ones, need to be explicited and subsequently proved.

To formalize this game-based framework, Nowak developed an extension to some of the Coq standard libraries by proving several properties of cyclic groups and properties of probabilities over cyclic groups. In order to provide some degree of automation to the framework, automatic tactics were constructed in Ltac to deal with game transformations. However, shallow embedding [GW07] is used in the modeling of games. In shallow embedding games are seen as functions and as a result the game structure is not accessible. This limits the framework's support for automation and, more importantly, the framework does not have direct control over the computational cost of evaluating these functions. As a consequence, even though the framework uses the asymptotic security model, the notion of efficient and negligible were not precisely defined. Also, Nowak did not implement the random oracle model in his framework. In Chapter 4 we describe the implementation of such model developed during this thesis.

ElGamal Proof The semantic security proof of the ElGamal cryptographic scheme in Nowak's framework will now be shown.

Game 0. The previously defined semantic security game is used as the starting point and the generic algorithms are replaced with their definitions in ElGamal thus obtaining the following game :

$$\begin{aligned}
&\text{Game } 0 \doteq \\
&\quad x \overset{\$}{\leftarrow} \mathbb{Z}_q \ ; \ \alpha \leftarrow \gamma^x \ ; \ r \overset{\$}{\leftarrow} R \ ; \\
&\quad (m_0,m_1) \leftarrow A_1(r,\alpha) \ ; \\
&\quad b \overset{\$}{\leftarrow} \{0,1\} \ ; \ y \overset{\$}{\leftarrow} \mathbb{Z}_q \ ; \\
&\quad \beta \leftarrow \gamma^x \ ; \\
&\quad \delta \leftarrow \alpha^y \ ; \ \zeta \leftarrow \delta.m_b \ ; \\
&\quad \hat{b} \leftarrow A_2(r,\alpha,\beta,\zeta) \ ;
\end{aligned}$$

Let S_0 be the event where adversary A wins the game, that is, when $b = \hat{b}$. Then, semantic security adversarial advantage can be defined as $|Pr[S_0] - \frac{1}{2}|$. This advantage must be proved negligible in order to achieve semantic security.

Game 1. This game differs from Game 0 in the computation of δ. Game 0 computed δ as α^y while this new game randomly chooses z from $\mathbb{Z}_q$ to compute δ as γ^z. The resulting game is the following:

$$\begin{aligned}
&\text{Game } 1 \doteq \\
&\quad x \overset{\$}{\leftarrow} \mathbb{Z}_q \ ; \ \alpha \leftarrow \gamma^x \ ; \ r \overset{\$}{\leftarrow} R \ ; \\
&\quad (m_0,m_1) \leftarrow A_1(r,\alpha) \ ; \\
&\quad b \overset{\$}{\leftarrow} \{0,1\} \ ; \ y \overset{\$}{\leftarrow} \mathbb{Z}_q \ ; \\
&\quad \beta \leftarrow \gamma^x \ ; \\
&\quad z \leftarrow \mathbb{Z}_q \ ; \ \delta \leftarrow \gamma^z \ ; \ \zeta \leftarrow \delta \cdot m_b \ ; \\
&\quad \hat{b} \leftarrow A_2(r,\alpha,\beta,\zeta) \ ;
\end{aligned}$$

This is considered a transition based on indistinguishability. Essentially, from Game 0 to Game 1 what occurred was a transformation of the triple (α,β,δ). In Game 0 this was of the form $(\gamma^x,\gamma^y,\gamma^{xy})$ while in Game 1 this triple takes the form $(\gamma^x,\gamma^y,\gamma^z)$. Under the DDH assumption these triples are considered indistinguishable and as consequence the change of probability between games, $|Pr[S_0] - P[S_1]|$, is negligible.

Game 2. In this game, the random choice of the variable z and the computation of ζ as $\delta.m_b$ are replaced by the computation of ζ as a random element from the group G.

```
Game 2 $\doteq$
    $x \overset{\$}{\leftarrow} \mathbb{Z}_q$ ; $\alpha \leftarrow \gamma^x$ ; $r \overset{\$}{\leftarrow} R$ ;
    $(m_0,m_1) \leftarrow A_1(r,\alpha)$ ;
    $b \overset{\$}{\leftarrow} \{0,1\}$ ; $y \overset{\$}{\leftarrow} \mathbb{Z}_q$ ;
    $\beta \leftarrow \gamma^x$ ; $\zeta \leftarrow G$ ;
    $\hat{b} \leftarrow A_2(r,\alpha,\beta,\zeta)$ ;
```

This transition is considered a bridging step. The change in the computation of ζ is a algebraic property of cyclic groups. When a uniformly distributed element of the group (δ) is multiplied by another element (m_b), the result is uniformly distributed (random element of G). In order to be used, this property had to proved in the framework. This transformation is indistinguishable by the adversary, thus $Pr[S_1] - P[S_2] = 0$.

In this final game the encrypted message has no relation to the chosen message (m_b). As a result, the adversary cannot obtain any information about the chosen message by observing the resulting ciphertext. The adversary can only guess which message was encrypted and, as a result, the probability of success is $\frac{1}{2}$, $Pr[S_2] = \frac{1}{2}$.

The probability of the adversary winning the game can then be bounded by the probability of the final game. Game transformations used were either transitions based on indistinguishability, where the changes of probability are negligible, or bridging steps that maintain the probability unchanged. Consequently, $|Pr[S_0] - P[S_2]|$ is negligible and $Pr[S_0] \approx \frac{1}{2}$. This means the adversary possesses no advantage in breaking the semantic security game of ElGamal.

2.2 CertiCrypt

It is now presented the game-based framework CertiCrypt [BGB09]. CertiCrypt provides certified tools to reason about games whose implementation involves theory of many branches such as probability, algorithm complexity, algebra and semantics of programming languages. Additionally, by taking a code-centric view of games, where game transformations are program transformations, games become easier to verify but proofs tend to be complicated. Both these properties make CertiCrypt an inherently complex framework.
CertiCrypt can be seen as an implementation of the tool that Halevi suggested to certify game-based proofs. CertiCrypt provides some enhancements to Halevi's vision by being a unified framework for developing full proofs, and as such, Halevi's impermissible transitions are not allowed. Also, the use of Coq allows proofs to be independently verified. At its lower layer, CertiCrypt formalizes an imperative probabilistic programming language (pWhile) with intent to faithfully and rigorously encode games. pWhile programs' typability is assured by the use of a deep and dependently-typed embedding of the syntax and its probabilistic semantics is defined on top of ALEA – a library for reasoning on randomized algorithms in Coq [APM09]. Additionally, the semantics is capable of capturing the running cost of programs, thus allowing the formalization of program's complexity requirements (such as being PPT).

In comparison with Nowak's framework, CertiCrypt formalizes the random oracle model, supports proofs with failure events and captures the notion of well formed adversaries by controlling adversarial access to variables and to procedure calls. Additionally, CertiCrypt uses deep embedding for game modeling, instead

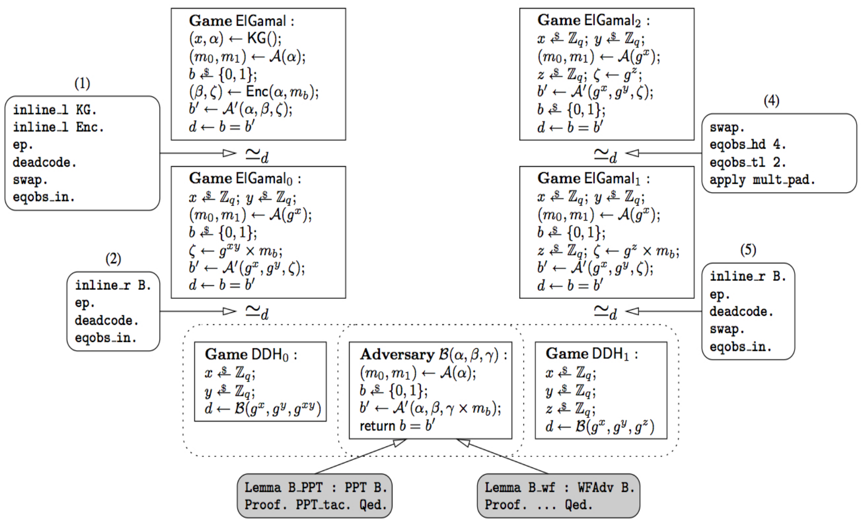
Game ElGamal :
$(x,\alpha) \leftarrow \mathsf{KG}()$;
$(m_0,m_1) \leftarrow \mathcal{A}(\alpha)$;
$b \xleftarrow{\$} \{0,1\}$;
$(\beta,\zeta) \leftarrow \mathsf{Enc}(\alpha,m_b)$;
$b' \leftarrow \mathcal{A}'(\alpha,\beta,\zeta)$;
$d \leftarrow b = b'$
(1)
inline_l KG.
inline_l Enc.
ep.
deadcode.
swap.
eqobs_in.
$\simeq_d$
Game ElGamal$_0$:
$x \xleftarrow{\$} \mathbb{Z}_q$; $y \xleftarrow{\$} \mathbb{Z}_q$;
$(m_0,m_1) \leftarrow \mathcal{A}(g^x)$;
$b \xleftarrow{\$} \{0,1\}$;
$\zeta \leftarrow g^{xy} \times m_b$;
$b' \leftarrow \mathcal{A}'(g^x,g^y,\zeta)$;
$d \leftarrow b = b'$
(2)
inline_r B.
ep.
deadcode.
eqobs_in.
$\simeq_d$
Game ElGamal$_2$:
$x \xleftarrow{\$} \mathbb{Z}_q$; $y \xleftarrow{\$} \mathbb{Z}_q$;
$(m_0,m_1) \leftarrow \mathcal{A}(g^x)$;
$z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z$;
$b' \leftarrow \mathcal{A}'(g^x,g^y,\zeta)$;
$b \xleftarrow{\$} \{0,1\}$;
$d \leftarrow b = b'$
(4)
swap.
eqobs_hd 4.
eqobs_tl 2.
apply mult_pad.
$\simeq_d$
Game ElGamal$_1$:
$x \xleftarrow{\$} \mathbb{Z}_q$; $y \xleftarrow{\$} \mathbb{Z}_q$;
$(m_0,m_1) \leftarrow \mathcal{A}(g^x)$;
$b \xleftarrow{\$} \{0,1\}$;
$z \xleftarrow{\$} \mathbb{Z}_q$; $\zeta \leftarrow g^z \times m_b$;
$b' \leftarrow \mathcal{A}'(g^x,g^y,\zeta)$;
$d \leftarrow b = b'$
(5)
inline_r B.
ep.
deadcode.
swap.
eqobs_in.
$\simeq_d$
Game DDH$_0$:
$x \xleftarrow{\$} \mathbb{Z}_q$;
$y \xleftarrow{\$} \mathbb{Z}_q$;
$d \leftarrow \mathcal{B}(g^x,g^y,g^{xy})$
Adversary $\mathcal{B}(\alpha,\beta,\gamma)$:
$(m_0,m_1) \leftarrow \mathcal{A}(\alpha)$;
$b \xleftarrow{\$} \{0,1\}$;
$b' \leftarrow \mathcal{A}'(\alpha,\beta,\gamma \times m_b)$;
return $b = b'$
Game DDH$_1$:
$x \xleftarrow{\$} \mathbb{Z}_q$;
$y \xleftarrow{\$} \mathbb{Z}_q$;
$z \xleftarrow{\$} \mathbb{Z}_q$;
$d \leftarrow \mathcal{B}(g^x,g^y,g^z)$
Lemma B_PPT : PPT B.
Proof. PPT_tac. Qed.
Lemma B_wf : WFAdv B.
Proof. ... Qed.

Figure 2.1: CertiCrypt's ElGamal Proof [BGB09]

of the shallow embedding used in Nowak's framework. This provides support for proof automation by facilitating syntactic transformations.

ElGamal Proof As usual the generic semantic security game will be the starting point. The proof is illustrated in Figure 2.1. Game ElGamal$_0$ is achieved by using the tactics in (1). The *inline* tactic is responsible for procedure call inlining and unfolds the definition of the key-generation and encryption algorithms. The *ep* tactic is used for expression propagation and, in this case, substitutes the pair (β,α) in A_2 by (g^x,g^y). Since β and α are no longer used, their variable assignments do not influence the game's output. These assignments can be eliminated with the *deadcode* tactic. The *swap* tactic is used to alter the order of some instructions, namely the random assignment of the variable y. To finalize, *eqobs_in* decides observational equivalence and as a result this transformation is a permissible one.

Transitions (2) and (5) can be summarized as the use of the DDH assumption to change the triple of the form (γ^x,γ^y,$\gamma^x y$) to (γ^x,γ^y,γ^z).

In the final game, the computation of ζ changes from $g^z \times m_b$ to g^z. The *eqobs* tactic used proves that both games are identical except for the the ζ assignment. It is then applied the algebraic property of groups (*mult_pad*) that proves that the different assignments of ζ induce the same distribution. This last transition removes the dependency of ζ on b and thus the dependency of $\hat{b}$ on b. As a result, the probability of success is $\frac{1}{2}$.

2.3 CryptoVerif

CryptoVerif [BP06, Bla07] is a game-based framework that aspires to blend the realistic computational model with the so called Dolev-Yao model [DY83]. This is a formal and abstract model in which cryptographic primitives are modeled as ideal black boxes and proofs can be done automatically. However, these proofs are made under strong and unrealistic assumptions.

CryptoVerif employs the best of each model by designing a framework that is capable of achieving automatic provability under realistic computational assumptions. A process calculus based on the pi calculus [MPW92] is formalized to model games. The calculus was designed to facilitate the automation of cryptographic protocols proofs and game semantics are purely probabilistic. Also, processes run in polynomial time. As CertiCrypt, CryptoVerif also implements the random oracle model and uses the exact security model by precisely bounding the probability of an attack.

To develop the game-based proofs, security assumptions must be stated as equivalences. These equivalences can be reused thus they must be proved only once and play an essential role in the construction of the various games in a proof. However, these proofs have to be performed outside the scope of the framework, since they cannot be handled by CryptoVerif, and are inherently difficult. More-over, CryptoVerif possesses many lines of uncertified O'Caml code. This means that, contrarily to Nowak's framework and CertiCrypt, CryptoVerif is not a fully certified framework.

ElGamal proof To achieve a proof, CryptoVerif provides sequences of indistinguishable games that lead to the final game where adversarial advantage is 0 and to transition between games it tries to either use the security definitions of the cryptographic primitive or syntactic transformations. Normal proofs in CryptoVerif originate a great number of games as a result of game transformations being as simple as possible. As an example, the ElGamal proof of semantic security uses a sequence of 9 games.

To personalize proofs, users can interact with prover by providing the sequences of games that they expect the prover to follow. In practice, the user needs to know how the proof is expected to be developed in order to provide to the prover the exact equivalences needed even in an automatic proof. As it has been seen in sections 2.1 and 2.2, the proof of ElGamal needs the DDH assumption. This property needs to be stated in order to aid CryptoVerif finding the correct sequence of games:

```
equiv
    ! n new x : Z ; new y : Z ; (
      ( ) -> exp (gamma , x ) ,
      ( ) -> exp (gamma , y ) ,
      ( ) -> exp (gamma , mult ( x , y ) ) )
           <=(pDDH( time ) * n)=>
    ! n new x : Z ; new y : Z ; new z : Z ; (
      ( ) -> exp (gamma , x ) ,
      ( ) -> exp (gamma , y ) ,
            ( ) -> exp (gamma , z ) ) .
```

This illustrates that under the DDH assumption the DDH triples are indistinguishable by the adversary with probability dependent on the security parameter (n). Since CryptoVerif uses the exact security model, this probability is not considered negligible but is calculated instead. Some other equivalences also need to be defined like, for example, the property that states that multiplying a random element of a group with another element of the group results in a random element of the group :

```
equiv
    ( y :G) n > new x :G; dot ( x , y ) [ all ]
        <=(0)=>
    ( y :G) n > new x :G; x .
```

This is a bridging step and so this transformation has no effect in the probability (<=(0)=>).

By using these equivalences, game simplification and elimination of the so called dead code, CryptoVerif tries to generate a sequence of valid games ending in a game where the adversary has no advantage. Adversarial advantage can then be bounded from the changes in probability induced by game transformations.

2.4 Conclusion

In this chapter it was shown how the game-playing technique can be used to achieve security in the provable security sense and how this approach simplifies cryptographic proofs. Nowak's framework uses a shallow embedding of games in Coq placing some limitations on proof steps automation while CryptoVerif provides automatic provability but requires the specification of the equivalences needed in the proof from the user and it is not a fully certified framework. CertiCrypt is a more complete framework with, for example, formalization of a relational Hoare logic and the well formedness of PPT adversaries, making it an inherently complex framework.

Chapter 3

David Nowak's Framework

This chapter will provide an in depth look at David Nowak's framework for cryptographic game based proofs. The framework is built on top of the proof assistant Coq and the cryptographic proofs that result from the framework are expected to be formal enough to be machine checked and also comprehensible enough to be humanly verified [Now07]. The framework has been used in the semantic security proof of both the ElGamal (hashed and non-hashed versions) and Goldwasser-Micali asymmetric cryptosystems.

Nowak's development is structured with the following three building blocks:

- the formalization of probabilities
- mathematical results
- the formalization of game based approach to cryptographic proofs

3.1 Probabilities

Cryptographic game based proofs require the demonstration that the adversarial advantage is negligible. This approach is heavily dependent on probabilities and its corresponding theory.

In order to model probabilities in this framework, it is only taken into consideration finite probability distributions that are implemented as lists of pairs that contain a value and its associated weight in the distribution. A weight is a real number and is modeled using Coq's reals whose type is *R*.

```
Definition T' (A:Type) : Type :=
  list (A*R).
```

This is a kind of pre-distribution because it does not enforce that the sum of probabilities adds up to 1. The apostrophe symbol suffixed to the name of a definition or lemma indicates that we are working with such pre-distributions.

Pre-distributions have been shown to possess a monadic structure [APM09]. The definition of the monadic operations follows the usual formulation of monads, the so called Kleisli triple [Mog91]:

- The type constructor of a monad receives a type A and constructs the corresponding monadic structure. One can easily see that in this framework the type constructor is *T'* and its instantiation with type A will result in a pre-distribution *T' A*.

- The unit function maps a value of type A to a distribution with only one element. As a result, the probability of occurrence of this value is 1:

  ```
  Definition ret' (A:Type) (a:A): T' A :=
    [ (a, 1) ].
  ```

- The binding operation takes a value of type *T' A* and a function F of type *(A → T' B)* and gives as a result a value of type *T' B*. In the case of distributions, this is achieved by pondering, for each $a \in A$, the distribution *(F a)* by the probability of a in the first distribution:

```
Fixpoint bind' (A B:Type)(d:T' A)(F:A -> T' B) : T' B :=
 match d with
  | nil => nil
  | (a,p) :: d' => ponder p (F a) ++ bind' d' F
  end.
```

The reserved word *Fixpoint* is used to Coq for the definition of recursive functions. The ponder function maps the multiplication of a real number p over the distribution:

```
Fixpoint ponder (A:Type)(p:R)(d:T' A) {struct d} : T' A :=
  match d with
  | nil => nil
  | (a,q)::d' => (a,p*q) :: ponder p d'
  end.
```

The actual definition of the distribution monad is based on the type of pre-distributions but, additionally, it also ensures that the pre-distribution is well formed with respect to the distribution sum law:

```
Definition T (A:Type) : Type :=
  {d:T' A | sum d = 1}.
```

sum d is the sum of probabilities in a distribution while the notation "$\{x : A|\ P\ x\}$" is used to define the subset of elements of type A that satisfy the P predicate. In other words, an element of the type T contains a pre-distribution and a proof that the pre-distribution is well formed.

Cryptographic games are defined with regards to an event that represents the meaning of breaking the cryptographic scheme. As a result, it is crucial to compute the probability of an event over a distribution. This is achieved by checking for each value of the distribution if the event, modeled as a decidable predicate, is true or not. This can be defined as a recursive function over the distribution:

```
Fixpoint probability' {A:Type}(P:A->bool)(d:T' A) : R :=
  match d with
  | nil => 0
  | (a,p)::d' => if P a then Rabs p + probability' P d'
                              else probability' P d'
  end.
```

Rabs p is the absolute value of the real *p*.

From here on forward, we will use the following notations for the binding operations used in the framework:

- $x \;\Leftarrow\; c1 \;\; ; \; c2$ for the binding of *c1* with (*fun* $x \;\Rightarrow\; c2\ x$).
- $x \;\leftarrow\; a \; ; \; c$ for the binding of a with (*fun* $x \;\Rightarrow\; c$) where a is the distribution with only one element. This is the same as $x \;\Leftarrow\; ret\ a \; ; \; c$.
- $x \overset{\$}{\leftarrow} l \; ; \; c$ for the binding of l with c where l is a nonempty list of elements and x is randomly sampled from the uniform distribution over the elements of l.

In order for this to be a correct implementation of a monadic structure it has to obey the following fundamental monadic laws: left identity, right identity and the associativity of monadic structures.

These monadic laws are proved correct with regards to the defined distributions by proving the following lemmas:

- Left-identity :

  Theorem unit_left : ∀ (A B : Type) (a : A) (F : A → T B),

  $$\mathrm{x} \leftarrow \mathrm{a} \; ; \mathrm{F\ x} = \mathrm{F\ a}.$$

- Right-identity :

  Theorem unit_right : ∀ (A : Type) (d : T A),

  $\quad$ x ⇐ d ; ret x = d.

- Associativity :

  Theorem associativity : ∀ (A B C) (d:T A) (F: A → T B) (G: B → T C),

  $\quad$ y ⇐ (x ⇐ d; F x) ; G y = x ⇐ d ; y ⇐ F x ; G y.

3.2 Mathematical Results

The definition of cryptographic schemes involves the use of mathematical structures and the correctness and security of the schemes are dependent on these structures' properties. The framework formalizes such structures, like bit strings and groups, along with its operations and also provides the proof of the corresponding properties.

Concerning the bit strings, the framework mostly focus on its xor operation and the properties that this operation possesses. The group definition is comprised of general properties of groups, such as, associativity or closure, and more specific results like properties of cyclic groups.

To formalize the definition of a group the *Record* construction of Coq is used. This allows the definition of records like in many programming languages but, additionally, it also provides a way of defining a signature for the module. Any instantiation of this type must be proved correct with respect to the type signature defined:

```
Record Group : Type := {
  carrier :> Type ;
  eq_dec : forall a b:carrier, {a=b}+{a<>b} ;
  neutral : carrier ;
  mult : carrier -> carrier -> carrier ;
  inv : carrier -> carrier ;
  neutral_left : forall a, mult neutral a = a;
  neutral_right : forall a, mult a neutral = a;
  inv_left : forall a, mult (inv a) a = neutral;
  inv_right : forall a, mult a (inv a) = neutral;
  associative : forall a b c, mult a (mult b c) = mult (mult a b) c
}.
```

The carrier is the type of elements of the group and the symbol :> represents a coercion from the type *Group* to the type of the carrier. That means that the type system automatically inserts the coercion function when a value of type *Group* is used in a context where the type *Type* is expected. *eq_ dec* is the decidability property of the group's equality and *neutral* is the neutral element of the group operation. Any instantiation of the *Group* type must define every field, like the operation of the group and the inverse of an element. Then it must be proved that the instantiation indeed forms a group by proving the properties that a group must possess: associativity and both identity and invertibility of the neutral element. The closure property of groups is already trivially assured since the signature of the group operation is defined as receiving two elements of the group and producing a element of the group.

The framework makes use of one instantiation of the group construction, namely the definition of group of integers modulo n. In order to formalize it, every single field of the group record must be defined. The type of elements of the groups are integers between 0 and *n-1* that are relative prime to *n*:

```
Definition carrier : Set := { x:Z | 0 <= x < n /\ rel_prime x n }.
```

The *eq_ dec* field is defined as the equality operation between integers, the group operation is multiplication modulo n and the inverse operation is the modular multiplicative inverse calculated by using the extended Euclidean algorithm. The neutral element is the integer 1:

```
Definition neutral : carrier := exist _ 1 neutral_spec.
```

where *neutral_ spec* is the proof that 1 is an element of the group, that is, that the following statement is valid:

```
Lemma neutral_spec : 0 <= 1 < n /\ rel_prime 1 n.
```

To complete the instantiation it is then needed to prove that the behavior of this group specification is indeed the expected one by proving the group's properties for this instantiation.

As expected, the formalization of these results required an extensive development library to support it. To that effect, the development includes several extensions to the standard library by providing lemmas, definitions and tactics that are of general use and can be added to the actual standard library. The major extension was the one developed over Coq's lists and Coq's representation of binary integers while minor additions were done to Coq's library about real numbers, logic, peano arithmetic and mathematical relations.

The framework also makes use in its development of the Pocklington library available in Coq's users' contribution. This library was developed by Olga Caprotti and Martijn Oostdijk in which the Pocklington-Lehmer [BSS99] primality test is formalized. Besides this formalization, the Pocklington library also provides a proof of Fermat's little theorem:

$$a^p = a \ (mod \ p)$$

for all a of type *integer* and p prime. The framework incorporates this library solely for the use of the Fermat's little theorem proof.

3.3 Games

This framework uses the sequence of games approach to the development of security proofs. In the game based approach, the proof of security, whatever the security notion in question may be, is achieved by reduction. This can be done by showing that an adversary capable of breaking the security notion can be used to solve a hard problem in an efficient way. In order to attain such proof, one usually starts with an initial game that captures the interaction between an adversary and the cryptographic scheme that is being analysed. The initial game is successively altered thus originating a sequence of games that culminates in a game where the adversarial advantage over a truly random adversary is trivially negligible. Every transition between games should be as simple as possible in order to prove its correctness.

In this framework, games can be seen as functions that return a distribution and in order to manipulate games the *Equiv* construct was defined:

```
Definition Equiv (A:Type)(P:A->bool)(epsilon:R)(d1 d2:T A) : Prop :=
  Rle (Rabs (probability P d1 - probability P d2)) epsilon.
```

Rle is the $\leq$ relation for elements of type *R*. *Equiv* receives two distributions (*d1*,*d2*), the event P and the real number ϵ and asserts that $|Pr\ (P\ d1) - Pr\ (P\ d2)| \leq \epsilon$.

This is used to formalize both the security assumptions, such as DDH, and the security notions. For example, the semantic security notion is formalized as a negligible difference between the probability of breaking the semantic security game and guessing a random coin toss:

```
Equiv (eqb true) epsilon
  (k <<= keygen ;
   r <$ randomness ;
  mm <- A1 r (fst k) ;
  b <$ [true;false] ;
  c <<= encrypt (fst k) (if b then fst mm else snd mm) ;
  b' <- A2 r (fst k) c ;
  ret (eqb b' b)
 )
 (b <$ [true; false];
  ret b
 )
```

The game event simply checks if the returning boolean is true by using the boolean equality operation *eqb*. In the top game the return value represents the result of winning the game while in the bottom the game it is a random value. The top game can be seen as the initial game and to prove the semantic security of any scheme, this initial game needs to be incrementally refined in order to achieve the bottom game. The change in probability made by these refinement steps must be bounded by ϵ.

To transition between games the framework provides the tactic *transitive*. This receives as input the new game and the difference in probability between the top game and the new game that we wish to transition to. Then, in order to transition from one game to the other, its equivalence, with regards to the given difference, needs to proved.

Because these differences between successive games are generally trivial, some automatic tactics are already available that prove these transitions:

- *flatten* uses the associativity lemma to try to prove that two distributions are equal.

- *propagate* uses the identity left property of monads to prove simple transi-

```
KeyGen(1^k) ≐
    $(\mathbf{G}, \gamma, q) \leftarrow \mathcal{GP}(1^k)$
    $x \xleftarrow{\$} \mathbb{Z}_q$
    $k \xleftarrow{\$} K$
    $\alpha \leftarrow \gamma^x$
    $sk \leftarrow ((\mathbf{G}, \gamma, q), x, k)$
    $pk \leftarrow ((\mathbf{G},\gamma, q), \alpha, k)$
    return $(pk,sk)$

Enc(pk, m) ≐
    $((\mathbf{G},\gamma,q),h,k) \leftarrow pk$
    $r \xleftarrow{\$} \mathbb{Z}_q$
    $c_1 \leftarrow \gamma^r$
    $c_2 \leftarrow m \oplus H_k(h^r)$
    return $(c_1,c_2)$

Dec(sk, c) ≐
    $((\mathbf{G},\gamma,q),x,k) \leftarrow sk$
    $(c_1, c_2) \leftarrow c$
    $m \leftarrow c_2 \oplus H_k(c_1^x)$
    return $m$
```

Figure 3.1: Hashed ElGamal Encryption Scheme

tions such as propagation of definitions.

- *flatten_propagate*, as the name implies, combines both *flatten* and *propagate* into a single tactic.
- *movebind* proves the reordering of the execution of two steps if this change does not interfere with the correct behavior of the game.

The intuition behind these tactics and its usefulness will be better understood in the following section where some examples of proofs will be given.

3.4 Hashed ElGamal Proof

It is generally preferable to work with bit strings as messages instead of group elements. Therefore, the ElGamal encryption scheme can be adapted by using an hash function. This variant is called Hashed ElGamal and its definition is given in Figure 3.1. H_k (with $k \in K$) is a family of hash functions where each H_k is a function from G to $\{0,1\}^l$. The Hashed ElGamal scheme is semantically secure under the DDH assumption and under the assumption that the family of hash function is entropy smoothing (ES).

The DDH assumption has already been discussed (Section 1.1) and is based on the problem of distinguishing triples of the form $(\gamma^x,\gamma^y,\gamma^{xy})$ and $(\gamma^x,\gamma^y,\gamma^z)$, where x, y, z are random elements of $\mathbb{Z}_q$.

The entropy smoothing assumption states that it is hard to distinguish pairs of the form $(k, H_k(m))$ from pairs of the form (k,h) where k is randomly sampled from K, m from G and h from $\{0,1\}^l$. This can be stated in a formal way by defining the advantage of breaking the ES assumption as:

$$\mathbf{Adv}_A^{\mathrm{ES}}(k) = \left|\Pr[k \overset{\$}{\leftarrow} K, m \overset{\$}{\leftarrow} G : A(k,H_k(m)) = 1] - \Pr[k \overset{\$}{\leftarrow} K, h \overset{\$}{\leftarrow} \{0,1\}^l : A(k,h) = 1]\right|$$

H_k is entropy smoothing if every efficient adversary A has negligible advantage.

We will now give a in-depth look to the proof in the standard model of Hashed ElGamal already done in Nowak's framework. This will be illustrative of the capabilities that the framework possesses.

3.4.1 Correctness Proof

We begin with the proof of correctness whose definition in terms of game equivalence is the following:

```
Lemma correctness : forall m,
 Equiv (eqb true) 0
  (k <<= keygen;
   c <<= encrypt (fst k) m;
   m' <<= decrypt (snd k) c;
   ret (equal _ m m')
  )
  (ret true
  ).
```

As a result, a cryptographic scheme is correct if the game of comparing a message *m* with the message *m'* resulting of encrypting and decrypting has the same probability as the game that always returns true. Since both games must be equivalent, every transition has to be a bridging step

We start the proof and unfold the definition of keygen, encrypt and decrypt and transition to the following game:

```
transitive 0 (
  x  <$ seqNE 0 (order G);
  k  <$ hashkeys;
  kp <- ((g^x, k), (x, k));
  y  <$ seqNE 0 (order G);
  c  <- (g^y, hash (snd (fst kp)) (fst (fst kp) ^ y) # m);
  m' <- hash (snd (snd kp)) (fst c ^ fst (snd kp)) # snd c;
  ret (equal m m')
).
```

seqNE 0 (*order G*) is a sequence from 0 to *order G* (size of the group *G*) with the assurance that the sequence is not empty and the # symbol is a notation for the bit string xor operation. This transition consists in inlining the calls to keygen, encrypt and decrypt. This is a simple bridging step in order to organize the game that uses the associativity property of monads and thus can be proven by using the *flatten* tactic.

In order to show that *m* and *m'* are indeed equal, we would like to simplify the computation of *m'*:

```
transitive 0 (
  x <$ seqNE 0 (order G);
  k <$ hashkeys;
  y <$ seqNE 0 (order G);
  ret (equal m (hash k ((g^y)^x) # (hash k ((g^x)^y) # m)))
).
```

This transition merely consists in the propagation of all deterministic assignments to the return of the game and thus it is also considered a bridging step that can be solved by the *propagate* tactic. This is where the mathematical results present in the framework come into action. The result of encrypting and decrypting m is now $H_k(g^{yx}) \oplus H_k(g^{xy}) \oplus m$. By using the property that the xor of a bit string with itself produces the neutral element (all zeros bit string), $m = H_k(g^{xy}) \oplus H_k(g^{xy}) \oplus m$ can now be rewritten into m and thus we obtain the equality test between the same message which is equivalent to the game that always returns true.

3.4.2 Security Proof

We now show the proof of semantic security of the Hashed ElGamal encryption under the the DDH assumption and the entropy smoothing assumption on the hash family. Both assumptions are defined as hypotheses in order to be used during the proof. We start with the initial semantic security game and perform some bridging steps, such as unfolding of definitions, definitions propagation and reordering of steps. We then arrive at the following goal:

```
Equiv (eqb true) (epsilon_DDH + epsilon_ES)
  (x <$ seqNE 0 (order G);
   y <$ seqNE 0 (order G);
   k <$ hashkeys;
   r <$ randomness;
   b <$ [true;false];
   mm <- A1 r (g^x,k);
   c <- (g^y, hash k ((g^x)^y) # (if b then fst mm else snd mm));
   b' <- A2 r (g^x, k) c;
   ret (eqb b' b)
   )
   (b <$ [true; false] ;
    ret b
   )
```

Notice that, unlike the correctness proof, these games are not strictly equivalent but instead, their difference in probability is bounded by $\epsilon_{DDH} + \epsilon_{ES}$. As a result, this proof will have two transitions based on indistinguishability: one using the DDH assumption and the other the ES assumption. We have propagated only the assignment of the key pair to the return of the game for clarity's sake. We now proceed with the first transition based on indistinguishability:

```
transitive epsilon_DDH (
      (x <$ seqNE 0 (order G);
       y <$ seqNE 0 (order G);
       z <$ seqNE 0 (order G);
       k <$ hashkeys;
       r <$ randomness;
       mm <- A1 r (g ^ x, k);
       b <$ [true; false] ;
       c <- (g ^ y, hash k g^z # (if b then fst mm else snd mm));
       b' <- A2 r (g ^ x, k) c;
       ret (eqb b' b))
).
```

This game differs from the previous one by the sampling of the random variable z and by replacing g^{xy} by g^z. Under the DDH assumption this transition is negligible and can be proved by applying DDH hypothesis.

Since z is a random value, it is possible to show that g^z is also random. This is a property of cyclic groups formalized in the framework. We now proceed with the last transition based on indistinguishability:

```
transitive epsilon_ES (
  (x <$ seqNE 0 (order G);
   y <$ seqNE 0 (order G);
   z <$ seqNE 0 (order G);
   k <$ hashkeys;
   r <$ randomness;
   mm <- A1 r (g^x, k);
   b <$ [true;false];
   c <- (g^y, s # (if b then fst mm else snd mm));
   b' <- A2 r (g^x, k) c;
   ret (eqb b' b))
).
```

This can proved by applying the ES assumption because, with the knowledge that g^z is random and under the ES assumption, $hash\ k\ g^z$ is also random and is represented by the random sampled variable s.

With the last transition, we are now in position to show that the resulting game is equivalent to guessing a random toss. Being s a random bit string, the xor of s with the encryption of the chosen message is also a random bit string. As a result, the ciphertext no longer depends on the chosen plaintext. Accordingly, the adversary does not possess any information about the chosen plaintext thus it has no other choice than to guess. It is then proved that the difference in probability between the initial and final is negligible, more precisely, this differences is equal to $\epsilon_{DDH} + \epsilon_{ES}$.

3.5 Conclusion

In this chapter we have explored the building blocks of the framework developed by David Nowak. We have seen the formalization of probabilities that have as its core the definition of a distribution monad, the approach utilized to formalize the game methodology and the mathematical results present in the framework that support its proofs with emphasis on the formalization of group theory. An example of a proof in the framework was provided with the Hashed ElGamal's proof of both correctness and semantic security.

Chapter 4

Random Oracle Methodology

In this chapter, the implementation of the random oracle model in Nowak's framework will be described. The starting point of this implementation is the definition of stateful distributions (Nowak's contribution). All the remaining abstractions needed in the formalization are original, and developed specifically for this thesis. The capabilities of the framework's extensions will be illustrated with the proof in the random oracle model of the semantic security of the Hashed ElGamal encryption scheme.

4.1 Random Oracle Model

The random oracle model is a computational model used to prove the security of a scheme in the reductionist sense that was introduced by Bellare and Rogaway [BR93] but its idea originates from Fiat and Shamir [FS86]. By reductionist we mean that in order to prove a security notion we need to show that a computational assumption of a known hard problem P, such as the problem of the discrete logarithm, implies the hardness of breaking the scheme. This can be achieved by showing that an algorithm that could break the scheme can be used to solve P in polynomial time.

What makes this model different from others, such as the standard model, is that one or more components of the protocol are idealized as random oracles (for example, hash functions). A random oracle is a black box that maps every possible query to a random value from its output domain. In the random oracle model every participant obtains access to the public oracle. To obtain an answer participants need to directly query the oracle and no one has information about the mapping of a value x until x has been queried by any of the participants.

In order for the protocol to be used in the real world it needs to be instantiated. That is, we need to replace the random oracle with an existing implementation of the idealized component in order to obtain a concrete protocol. Since no instantiation of the random oracle can implement a truly random component, the proof of security in the random oracle model does not translate into a proof in the standard model. Also, it has been shown that some protocols that can be proven secure in ROM, have no secure implementation in the real world [CGH04]. However, these are artificial counter-examples that do not model real world protocols.

Despite its limitations, a proof of security in the random oracle model provides a strong indication about its security, namely that, in the idealization of the hash function, if an adversary does not break any of the security assumptions, then he must find a flaw in the chosen instantiating hash function in order to break the security notion that is being proved. For example, since generic attacks do not take into account specifics about the primitives implementation, since they already assume the hash function as being random, a secure protocol in the random oracle model is trivially secure against these type of attacks.

Despite the instantiation process in ROM being of heuristic nature, some properties of hash functions that maintain the security of protocols instantiated in the standard model have been studied [CMR98, Can97, GHR99]. As a consequence, the process of designing protocols and providing proofs of their security, assuming the ideal behavior of hash functions, can be detached from the process of designing hash functions that possess the needed properties.

4.2 Framework Extensions

The implementation of the random oracle model involves, as expected, a significant enhancement to the capabilities of the existing framework. This involves not only the formalization of random oracles and its interaction with both the adversary and the cryptosystem, but also the development of tools that allow the reasoning about cryptographic games in the random oracle model.

4.2.1 State Monad

In order to store the mapping between the queried group elements and their corresponding hash value, a state monad was defined and combined with the distribution monad. This monad was formalized by David Nowak and it is comprised of computations that depend on the current state, at the moment of each step's execution, with each computation being able to modify the current state. By using this monad, the state can be 'hidden' internally. This provides a neat solution to the problem of implementing the state feature because there is no need for the explicit data flow of the state from step to step. Like in the definition of the distribution monad, we provide the definition of Kleisli triple:

- The type constructor receives the type S of the state and the type A of computations and constructs the type that given the initial state produces a distribution monad whose values are pairs of type *(S * A)*:

  ```
  Definition T (State A:Type) := State ->
  ```

```
    Distribution.T (State * A).
```

- The unit function simply receives the current state and leaves it unchanged:

  ```
  Definition ret (State A:Type)(a:A) : T State A :=
    fun s => Distribution.ret (s, a).
  ```

  Distribution.x is used to access the x construct defined in the Distribution library. In this case, the monad constructor of distributions is used.

- The binding function propagates the resulting state from F:

  ```
  Definition bind (State A B)(d:T State A)(F:A->T State B) :
   T State B := fun s =>
    Distribution.bind (d s) (fun s' => F (snd s') (fst s')).
  ```

The fundamental monadic laws that were proved correct regarding the definition of the distribution monad are also proved correct with respect to the definition of the state monad. In order to explicitly manipulate the state, the fetch and assign functions are defined. Fetch returns the current state as a value:

```
Definition fetch : T State State :=
  fun s => Distribution.ret (s, s).
```

Assign replaces the current state with the state given as input to assign:

```
Definition assign (s:State) : T State unit :=
  fun _ => Distribution.ret (s, tt).
```

Also, it is possible to add a value to a state mapping and to check whether a certain value is present in the state.

4.2.2 Basic Probabilities Laws

In order to compose events and reason about its probability, the framework was extended with the formalization of basic laws of probability. The following type of events were included:

- The conjunction of two events.

  ```
  Definition AND {A} (P Q:A->bool) (x:A) := andb (P x) (Q x).
  ```

- The disjunction of two events.

  ```
  Definition OR {A} (P Q:A->bool) (x:A) := orb (P x) (Q x).
  ```

- The negation of an event.

  ```
  Definition NEG {A} (P:A->bool) (x:A) := negb (P x).
  ```

- The certain event that is always true.

  ```
  Definition Ptop {A:Type} (x:A) : bool := true.
  ```

- The impossible event that is always false.

  ```
  Definition Pbot {A:Type} (x:A) : bool := false.
  ```

To deal with probabilities about these type of events, the probability theory was extended with the formalization of some fundamental laws of probability. Namely, straightforward properties like the probability of the certain event being 1 and the probability of the impossible event being 0 were added. The probability of an event's negation is calculated by the complement law:

```
probability (NEG E) d = 1 - (probability E d)
```

E is an event and d is a distribution. From the complement law, one can infer the law to decompose an event :

```
probability E d =
 probability (AND E E') d + probability (AND E (NEG E')) d
```

The probability of the event that results from the disjunction of some two events can be determined by applying the union rule:

```
probability (OR E E') d =
 probability E d + probability E' d - probability (AND E E') d
```

4.2.3 EqP

Distributions are pairs composed by values and their corresponding probability of occurrence. It is often desirable to restrain our scope to the list of values of a distribution and disregard their weight in the distribution. We named the list containing these values as the support of a distribution. Our definition differs from the generally conveyed meaning of support which discards values that possess zero probability in the distribution. Our definition does not need to filter these values because the way the support definition is used throughout the framework already takes this into account.

```
Definition support' {A} (d:T' A) : list A := List.map (@fst A R) d.
```

The support of a distribution can be used to define the support of an event over the distribution. This restricts the support to a subset that satisfies the given event and this is called *supP*:

```
Definition supP' {A} (P:A->bool) (d:T' A) : list A :=
  List.filter P (support' d).
```

Both these definitions allows us to define a stronger relation of equality between two distributions with respect to a certain predicate than the already defined *Equiv*:

```
Definition EqP (P:A->bool) (d d':T A) :=
 forall x, List.In x ((supP P d) ++ (supP P d')) ->
 probability (eq_fun eq_dec x) d = probability (eq_fun eq_dec x) d'.
```

eq_fun eq_ dec x represents a singular event *x* that checks the presence of *x* in the distribution and the *EqP* equivalence states that for every value *x* of the support that satisfies *P*, the probability of *x* is the same in both *d* and *d'* distributions. This is a stricter constrain on the equivalence than the *Equiv* one because *Equiv* only states that according to a predicate *P*, the probability of the predicate being satisfied is the same in both distributions. Being a stronger relation of equality, it is obvious that the *EqP* equality between two distributions implies the *Equiv* one. Also, when met with a proof of an *Equiv* relation it suffices to prove the *EqP* for a weaker event than the one used for *Equiv*:

```
Lemma EqP_Equiv_weak : forall A eq_dec (P P':A->bool) (d1 d2 : T A),
 (forall x, P' x = true -> P x = true) -> EqP eq_dec P d1 d2 ->
 Equiv P' 0 d1 d2.
```

This is illustrative of the power of *EqP* because, when met with a proof of an Equiv relation for an event of the form $A \wedge B$, we can transition to the *EqP* relation and it is enough to prove the *EqP* relation for just one of either A or B, whichever produces an easier proof. This is valid because, if we prove the *EqP* relation for the event A, then forcefully both subsets of distributions *d1* and *d2* that satisfy A are exactly the same and so, for the event $A \wedge B$, it will also be the same because any element of the distribution that satisfies A is contained in

$A \ \wedge \ B$. As a result, from the *EqP* relation of the event A, the *EqP* relation for the conjunction of A with any event can be automatically inferred.

The same reasoning cannot be applied to the *Equiv* relation because the probability of the event A may be the same in both distributions but the distributions may differ in the elements' weight to the probability. This could result in different probabilities for the event $A \ \wedge \ B$. This is representative of the difference of both equality relations with regards to the chosen event. For example, for the certain event, the *Equiv* relation is trivial (both probabilities are 1) while the *EqP* relation requires the proof that both distributions are extensionally equal and as result, *EqP* can be seen as a conditional equality between distributions. Also, the *EqP* relation, like *Equiv*, is also a reflexive, symmetric and transitive relation.

It is possible to move variables from the top of the game, if they occur in both games, to the context by using the extensionality lemma. This helps simplifying the respective games:

```
Lemma EqP_extensionality : forall (A B : Type) eqPdec (P : B -> bool)
 (d:T A) (F G : A -> T  B),
  (forall x, EqP eqPdec P (F x)  (G x))%distrib ->
  (EqP eqPdec P (x <<= d; F x)  (x <<= d; G x))%distrib
```

The % symbol is used to locally use a delimiting scope than the one currently opened. When faced with an *EqP* equivalence whose proof is not trivial it can be a good approach to decompose the proof of the *EqP* equivalence into smaller parts. The equivalence between subsets of both games can then be more easily verified than trying to prove the whole equivalence. The lemma *EqP_ bind* allows us to do just that:

```
Lemma EqP_bind : forall P (P':A->bool) (d1 d1':T A) (d2 d2':A->T B),
  forces P d2 P' -> forces P d2' P' ->
```

```
EqP eq_decA P' d1 d1' ->
(forall a, (P' a)=true -> EqP eq_decB P (d2 a) (d2' a)) ->
EqP eq_decB P (bind d1 d2) (bind d1' d2').
```

forces P d P' ensures that if *P* is valid after the execution of game *d* then *P'* must also be valid before the game is executed. The *forces* construct behaviour and its properties will be described in Section 4.2.6. The *EqP_bind* lemma allows us to prove the *EqP* relation for games *d* and *d'* by proving the following:

- the *EqP* relation between *d1* and *d1'* for a new chosen event (*P'*).
- The new event *P'* must force the event *P* in both *d2* and *d2'*.
- If *P'* is valid for any *a* then the EqP relation for *(d2 a)* and *(d2' a)* must also be valid.

If any part of the bind is the same in both games then variants of *EqP_bind* can be used. Namely the *EqP_bind1* and *EqP_bind2* lemmas can be used to prove *EqP eq_decB P (bind d1 d2) (bind d1' d2)* and *EqP eq_decB P (bind d1 d2) (bind d1 d2')* respectively.

4.2.4 Fundamental Lemma of Game-playing

In cryptographic game-based proofs, bridging steps and transitions based on indistinguishability are ubiquitous. However, these type of transitions are not always sufficient. It is also useful to be able to perform transitions based on failure events and in order to implement these type of transitions we first need to prove the correctness of the fundamental lemma of game-playing in our development. We follow [Sho04] approach and show that to prove that both games are equivalent it is enough to show that both games have the same distribution if the failure event does not occur:

```
Lemma FUNDAMENTAL : forall {A} eq_dec (E F:A->bool) epsilon
  (d1 d2: T State.T A) (s:State.T),
  probability F d2 s <= epsilon ->
  SEqP eq_dec (NEG F) d1 d2 s ->
  Equiv E epsilon d1 s d2 s.
```

The *SEqP* construct is similar to the *EqP* construct except for the fact that it deals with stateful distributions. From here on forward, any construct with a capital S prefixed to its name represents the stateful version of the construct.

The second premise is the crux of the lemma. By one side, it faithfully captures the Bellare and Rogaway [BR06] definition of identical-until-bad games, since it enforces that both distributions are extensionally equal whenever the negation of F occurs. On the other side, and due to the *SEqP* weakening property described before (*EqP_Equiv_weak*), it also ensures that the probability of $((NEG\ F) \wedge E)$ is also equal in both distributions. It is instructive to compare this formulation with an alternative formulation already formalized by David Nowak in the framework:

```
Lemma difference : forall A B (E:A->bool)(F:B->bool) d1 d2 d1' d2' s,
  Equiv E 0 d1 s (fst d1') s ->
  Equiv E 0 d2 s (fst d2') s ->
  Equiv F 0 (snd d1') s (snd d2') s ->
  Equiv (AND E (NEG F)) 0 d1' s d2' s ->
  Equiv P (probability F (snd d1') s) d1 s d2 s.
```

In this formalization both games *d1* and *d2* are extended with the information needed to deal with the failure event thus originating both *d1'* and *d2'* whose first projection is the original game's return and the second contains the return of the added information. The two first premises are responsible for showing that these extensions are well formed, that is, the first projection is the return of the original game. Our lemma of game-playing does not incorporate this extension as they are performed before the application of the lemma. The main difference between the lemmas is that Nowak's one requires the proof that the probability of the failure event F in both games is the same (third premise) and this involves the reduction to the hard problem for both games.

We opted to formalize a different game-playing lemma than the one already formalized so that we could reap the benefits from the definition of *EqP*. In our implementation of the game-playing lemma, it is sufficient to prove that the probability is bounded in only one of the games because by showing that both games are the same in the *EqP* sense for the event F then it can be concluded that the probability of the negation of the event is also the same. As a result, a proof in one of the games is enough. The possibility of only choosing one of the games in which to do the proof is quite helpful because generally, and in the Hashed ElGamal case this is also true, the proof which is often performed by reduction to the hard problem, is only possible in one of the games.

4.2.5 SInvM

The *EqP* construct can also be used in order to formalize the notion of a distribution that does not depend on a specific value present in the state for a given event:

```
Definition SInvM (m:Dom.T) (P:A->bool) (d:T State.T A) :=
  forall s h,
    EqP eqP_dec (fun x=>P (snd x))
      (d (add m h s))%distrib
      (x <<= (d s);
      ret (add m h (fst x), snd x))%distrib.
```

The add function receives a state, a value of its domain and the corresponding value of the co-domain and stores this association in the current state. *SInvM* uses the *EqP* construct to show that adding the m value to the state, before or after the occurrence of *d*, produces the same probability distribution for the P event. The use of *Equiv* instead of *EqP* would not be enough to guarantee this because its use would only ensure that the probability of the P event would stay the same but the weight of the distribution's elements that satisfy P could vary.

This notion is needed in proofs in the ROM to assert that the adversary only has access to oracle values he queries. Concretely, in the ROM proof of the Hashed ElGamal, when we need to reason about the failure event, which in this proof's case is the event where the adversary makes the oracle query about g^{xy}, it is necessary to show that, from the adversary's point of view, if the adversary does not make the oracle query about g^{xy} then computing h from the hash of g^{xy} is equivalent to randomly sampling h.

4.2.6 Forces

In order to propagate the occurrence of certain events from the bottom of a game to the top, we define the *forces* construction:

```
Definition forces (P:B->bool) (d:A->T B) (P':A->bool) :=
  forall a, P' a=true \/ ((supP P (d a))=List.nil).
```

For a given game *d*, a predicate *P*, that is checked after the game execution, forces a predicate *P'*, checked at the start of the execution, if the validity of the *P* predicate implies the validity of the *P'* predicate.

To better understand the way it works we show some of its properties. We begin with the obvious ones, namely the ones that show that if P is always true than any *P'* forces P and if *P'* is always false then it also forces any P:

```
Lemma forces_1 : forall (P:B->bool) (d:A->T B),
  forces P d (fun _=>true).


Lemma forces_0 : forall (P':A->bool) (d:A->T B),
  forces (fun _=>false) d P'.
```

In our development, many of the lemmas use as a premise proof of validity of a *forces* construct. Thus, like for the *EqP* construct, it is quite helpful that, in a proof about forces, one can deal with the binding of several game steps. The *forces_ bind* lemma is always used with that intention in mind:

```
Lemma forces_bind : forall A B C (P:C->bool) (P':A->bool)
 (P'':B*A->bool) (d1: A->T B) (d2:B->A->T C),
  forces P'' (fun a=>x <<= d1 a; ret (x,a)) P' ->
  (forall a, forces P (fun x=> d2 x a) (fun x=>P'' (x,a))) ->
  forces P (fun a => x <<= d1 a; d2 x a) P'.
```

During proofs that involve achieving the truthfulness of a forces construct, it is also beneficial, aside from the binding of *forces*, to transform the events present in *forces*. This can be done by either weakening the forcing event or strengthening the event that is being forced. These lemmas are respectively called *forces_ weak* and *forces_ impl*.

```
Lemma forces_weak : forall (P1 P2:B->bool) (P':A->bool) d,
  (forall x, P2 x=true -> P1 x=true) -> forces P1 d P' ->
   forces P2 d P'.


Lemma forces_impl : forall (P' P:A->bool) (PB:B->bool) d,
  (forall a, P a=true -> P' a=true) -> forces PB d P ->
   forces PB d P'.
```

The forces construct will be used in the Hashed ElGamal proof to show that if the adversary has not made the query about the value γ^{xy} by game's end then, at any point inside the game execution, the query has also not been made.

4.3 Hashed ElGamal in the Random Oracle Model

In this section we will provide a in-depth look to the semantic security proof of the Hashed ElGamal encryption scheme in the random oracle methodology. This proof requires the use of the framework's extensions that were developed in the scope of this work.

4.3.1 List CDH assumption

The ROM proof of Hashed ElGamal uses a weaker assumption of security than the proof in the standard model. Instead of the assumption of the intractability of DDH problem, it is only needed to assume the intractability of a variant of the computational Diffie-Hellman (CDH) called list computational Diffie-Hellman (list-CDH) [Sho04]. By weaker assumption we mean that an adversary that can break this weaker assumption can use this solution to efficiently break the stronger assumption. The reverse statement is not true, otherwise the problems would be considered equivalent.

The CDH assumption conveys that the problem of computing the value of γ^{xy} given (γ^x,γ^y) is hard. Formally, the advantage an efficient adversary A possesses in computing γ^{xy} is negligible:

$$\Pr[x,y \overset{\$}{\leftarrow} \mathbb{Z}_q : A(\gamma^x,\gamma^y) = \gamma^{xy}]$$

It can easily be seen that an adversary with non-negligible advantage in CDH also has non-negligible advantage in DDH. When given a triple of the form $(\gamma^x,\gamma^y,\gamma^r)$ it is trivial to decide whether r is equal to either one of *(x,y,z)* by using the CDH to compute γ^{xy} from γ^x and γ^y and compare it to γ^r. As such, the CDH assumption implies the DDH assumption but the contrary does not always holds since there exists several groups where the CDH assumption holds but the DDH one does not [JN03]. Thus, the CDH assumption is a considerably weaker assumption than DDH.

The list CDH assumption is a variant of the CDH assumption in which the adversary is now able to output a bounded list of group elements, instead of just one, and its advantage is the following:

$$\Pr[x,y \overset{\$}{\leftarrow} \mathbb{Z}_q : \gamma^{xy} \in A(\gamma^x,\gamma^y)]$$

For any efficient adversary its advantage is negligible under the list CDH assumption. The CDH and list-CDH are considered as equivalent assumptions. If the CDH advantage is non negligible then clearly the list-CDH advantage is also non negligible. On the other hand, if the list-CDH advantage is non negligible then a probabilistic algorithm can be built by randomly choosing a group element from the list of group elements [Sho04].

As in previous proof examples this computational assumption will be given as an hypothesis.

```
Definition ListCDH (n:nat) (epsilon:R) : Prop :=
 forall D : G -> G -> Distribution.T (vec G n),
  probability (eqb true)
  (x <$ seqNE 0 (order G);
   y <$ seqNE 0 (order G);
   l <<= D (g^x) (g^y);
   b <- vec_in (Group.eq_dec G) ((g^x)^y) l;
   ret b
  ) <= epsilon.
```

D represents the cryptographic scheme adversary and given g^x and g^y it outputs the list that is checked to see if it contains g^{xy}.

4.3.2 Adversarial loop of queries

Obviously, being a proof in the random oracle methodology, the adversary has access to random oracles. In the case of the Hashed ElGamal only one oracle is available. Given a group element the oracle produces a random hash of its value and stores the mapping between group element and hash value. When the oracle is given a query of a group element that has already been queried it returns the stored mapping.

With the random oracle functionality defined, it only remains to define the access conditions available to the adversary. The adversary can query the oracle before he chooses the pair of messages and can also make queries after receiving the encrypted message. The number of queries is not unlimited but, instead, it is bounded by a constant (which, in turn, is bounded polynomially on the security parameter).

In order to control the number of queries made by the adversary we have used dependent types to define the list of queries made by the adversary:

```
Definition vec (A:Type)(n:nat) : Type :=
    {l: list A |(beq_nat (length l) n) = true}.
```

beq_nat is the boolean equality operator on naturals. By defining *vec n* as a type with a *list* and a proof that its size is equal to n, we can define any adversarial loop of queries as returning the type *vec q*, where q is the maximum number of queries allowed in the loop, and we automatically obtain the proof that the list of queries is well formed and its size is q. This avoids reasoning explicitly on the number of queries made by the adversary.

The adversarial loop is defined as a recursive function on the number of queries:

```
Fixpoint ADVloop {A m} (n:nat)(lIN:vec A m)(d: list A -> T State.T A)
: T State.T (vec A (n+m)) :=
  match n return T State.T (vec A (n+m)) with
    | O => ret lIN
    | S n’ => (x <<= d (vec2list lIN);
                v <<= ADVloop n’ lIN d;
                ret (vec_cons x v))
  end.
```

vec2list returns the list inside the *vec* type, *vec_ cons* x v inserts v into v of type *vec* and d is responsible for executing the adversarial query.

The loop receives as input the list of queries already done and increments it with each query made by d. In the Hashed ElGamal proof, d will be instantiated with:

```
(fun l : list (G * Bitstring.T len) =>
 m <- A1 r (fst k) l; h <<= hash m; ret (m, h))
```

The adversarial view of the oracle’s state generally differs form the actual contents of the oracle. In the Hashed ElGamal proof the adversarial view only differs in the storing of the value g^{xy} but this distinction becomes more important in other proofs, where, for example, more than one random oracle may exist.

4.3.3 Semantic Security proof

As usual we start with the initial semantic security game instantiated with the Hashed ElGamal in the random oracle model. Several bridging steps are performed, namely propagation of definitions, reordering of some independent steps and moving variables to the context, to obtain the following game:

```
Equiv (eqb true) epsilon_lCDH
  (x <$ seqNE 0 (order G);
   y <$ seqNE 0 (order G);
   b <$ [true;false];
   l1 <<=
   ADVloop q1 vec_nil
     (fun l : list (G * Bitstring.T len) =>
      m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
   mm <- A2 r (g^x) (vec2list l1);
   h <<= hash ((g^x)^y);
   c <- (g^y, h # (if b then fst mm else snd mm));
   l2 <<= ADVloop q2 l1
     (fun l : list (G * Bitstring.T len) =>
      m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
   b' <- A4 r (g^x) (vec2list l2) c;
   ret (eqb b' b)
   ) (emptyS Codom.T)
   (b <$ [true;false];
    ret b
   ) (emptyS Codom.T)
```

This *Equiv* is stateful and *emptyS Codom.T* is the initial state for each game, and as expected, this represents the state with no mapping stored. Notice that the first adversarial loop receives an empty list (vec_nil) and both loops are bounded by the parameters *q1* and *q2*. We would like to state that the game where one tries to guess a random coin toss is equivalent to the top with h being randomly sampled instead of being the result of querying the oracle. This requires a transitions but this is a transition from the bottom games instead of the usual top game. This is also possible because *Equiv* is a symmetric equivalence and so both games are interchangeable.

This transition is a bridging step and can be proved correct proved by showing that since h is now random then the result of its xor with the chosen message is also random. Since the encrypted message no longer depends on the chosen message, guessing which message was chosen is the same as guessing a random coin toss. With this transition proved we obtain the following goal:

```
Equiv (eqb true) (epsilon_1CDH)
  (b <$ [true;false];
   l1 <<= ADVloop q1 vec_nil
     (fun l : list (G * Bitstring.T len) =>
      m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
   mm <- A2 r (g^x) (vec2list l1);
   h <$ Codom.to_list;
   c <- (g^y, h # (if b then fst mm else snd mm));
   l2 <<= ADVloop q2 l1
     (fun l : list (G * Bitstring.T len) =>
      m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
   b' <- A4 r (g^x) (vec2list l2) c;
   ret (eqb b' b)
) (emptyS Codom.T)
(b <$ [true;false];
l1 <<= ADVloop q1 vec_nil
  (fun l : list (G * Bitstring.T len) =>
   m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
mm <- A2 r (g^x) (vec2list l1);
h <$ hash ((g^x)^y);
c <- (g^y, h # (if b then fst mm else snd mm));
l2 <<= ADVloop q2 l1
  (fun l : list (G * Bitstring.T len) =>
   m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
b' <- A4 r (g^x) (vec2list l2) c;
ret (eqb b' b)
) (emptyS Codom.T)
```

Notice the sampling of x, y and b no longer appear in this goal since they were previously put in the context for both games. This is achieved by using the extensionality lemma and it was used to simplify both games. The proof of this goal is where all the extensions developed for the framework will come to work. In order to be able to use the information about the queries that were made during the game execution by the adversary and the value used in the encryption we need to somehow propagate these values to the return of the game so that the event can work with them. This is needed so that we can perform a transition based on a failure event.

Failure Event

So that one may check whether or not the query has been made by the adversary, we have defined the *NQueryP* predicate :

```
Definition NQueryP n (m:Dom.T) (l:vec (Dom.T*Codom.T) n) : bool :=
  eqb (vec_in Dom.eq_dec m (vec_map (@fst Dom.T Codom.T) l)) false.
```

The use of vec_map in the definition shows that we are only interested in the list of domain values present in the *vec*, thus discarding the information about the hash values, and this list is checked for the value m by vec_in. The failure event is based on *NQueryP* and simply receives the input from the game with g^{xy} and the list of queries made during both adversarial loop and checks if g^{xy} is present in the list:

```
Definition NQueryGxy {n} (x:(G*vec (G * Bitstring.T len) n) ) :=
  match x with (gxy, l2) => NQueryP gxy l2
  end.
```

With the failure event precisely defined, we can now apply the game-playing lemma in the following way :

```
apply FUNDAMENTAL with (F:=(fun x=> NQueryGxy (snd x))).
```

By applying the game-playing lemma, we are then obliged to prove the two following sub goals: the proof of the reduction of one of the games to the list-CDH problem and the proof that both games have the same supporting distribution if the adversary has not made the query.

Reduction to the list-CDH problem

In the sub goal where one needs to prove the reduction, we have the following proof objective:

```
probability
 (NEG (fun x : bool * (G * vec (G * Bitstring.T len) (q1 + q2)) =>
         NQueryGxy (snd x)))
```

```
(b <$ [true;false];
 l1 <<= ADVloop q1 vec_nil
   (fun l : list (G * Bitstring.T len) =>
    m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
 mm <- A2 r (g^x) (vec2list l1);
 h <$ Codom.to_list ;
 c <- (g ^ y, h # (if b then fst mm else snd mm));
 l2 <<= ADVloop q2 l1
   (fun l : list (G * Bitstring.T len) =>
    m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
 b' <- A4 r (g^x) (vec2list l2) c;
 ret (eqb b' b)
) (emptyS Codom.T) <= epsilon_lCDH
```

In this goal, we need to prove that the probability of the adversary querying the value g^{xy} is bounded by ϵ_{lCDH}. Notice that we chose the game where h is randomly sampled. This allows to consider the following subset of the game as being the adversary:

```
(b <$ [true;false];
 l1 <<= ADVloop q1 vec_nil
       (fun l : list (G * Bitstring.T len) =>
        m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
 mm <- A2 r (g^x) (vec2list l1);
 h <$ Codom.to_list ;
 c <- (g^y, h # (if b then fst mm else snd mm));
 l2 <<= ADVloop q2 l1
       (fun l : list (G * Bitstring.T len) =>
        m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
```

```
  )
```

By representing the adversary in such way we are able to form an instance of the list-CDH and, as a result, we are able to prove the goal by applying the existing hypothesis about the list-CDH assumption.

SEqP with the failure event

With the reduction to the list-CDH problem proved we now are faced with the proof of the next remaining sub goal in which the *SEqP* equivalence between both games with regards to the event where the adversary does not make the query about g^{xy} needs to be proved.

First we need to swap the order of execution of the first adversarial loop of queries and the sampling of h in the game where h is obtained by querying the random oracle. The purpose of this transition will be latter explained and the resulting goal from the transition is the following:

```
SEqP (eqP_dec RandomOracle.State.eq_dec (eq_decX (q2 + (q1 + 0))%nat))
 (fun x0 : bool * (G * vec (G * Bitstring.T len) (q2 + (q1 + 0))) =>
    NQueryGxy (snd x0))
  (h <<= hash ((g^x)^y);
   l1 <<= ADVloop q1 vec_nil
     (fun l : list (G * Bitstring.T len) =>
      m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
   mm <- A2 r (g^x) (vec2list l1);
   c <- (g^y, h # (if b then fst mm else snd mm));
   l2 <<= ADVloop q2 l1
     (fun l : list (G * Bitstring.T len) =>
      m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
   b' <- A4 r (g^x) (vec2list l2) c;
   ret (eqb b' b)
   )
   (h <$ Codom.to_list;
    l1 <<= ADVloop q1 vec_nil
      (fun l : list (G * Bitstring.T len) =>
       m <- A1 r (g^x) l; h <<= hash m; ret (m, h));
   mm <- A2 r (g^x) (vec2list l1);
   c <- (g^y, h # (if b then fst mm else snd mm));
   l2 <<= ADVloop q2 l1
      (fun l : list (G * Bitstring.T len) =>
       m <- A3 r (g^x) l c; h0 <<= hash m; ret (m, h0));
   b' <- A4 r (g^x) (vec2list l2) c;
   ret (eqb b' b)
   )
   (emptyS Codom.T)
```

The theorem responsible for swapping a game's order of execution of the hashing step is called *SEqP_hash*:

```
Theorem SEqP_hash : forall {A B} eq_decA eq_decB (P':A->bool)
 (P:B->bool) (d1:T _ A) (d2:A->Codom.T->T _ B) m s,
  SInvM eq_decA m P' d1 ->
  (forall h, Sforces P (fun x=> d2 x h) P') ->
  SEqP eq_decB P
      (x <<= d1; h <<= hash m; d2 x h)
      (h <<= hash m; x <<= d1; d2 x h) s.
```

So now we apply the *SEqP_hash* lemma where *P'* will be the event that checks whether the adversary has made the query about g^{xy}:

```
apply SEqP_hash with (P':= NQueryP ((g^x)^y)).
```

As expected, this generates two sub goals : the proof that the first adversarial loop is *SInvM* and the proof that if the query about g^{xy} has not been made by game's end then, forcefully, it has not been made after the resulting swap of steps.

The first sub goal can be proved by applying the lemma that assures the validity of *SInvM* for a general loop with *NQueryP*:

```
Lemma SInvM_ADVloop : forall eq_dec (m:Dom.T) ls
 (d:list (Dom.T*Codom.T)->T State.T (Dom.T*Codom.T))
 (n:nat) (l:vec (Dom.T*Codom.T) ls),
 (forall ls l, @NQueryP ls m l=true ->
  SInvM eq_dec m (fun x => negb
  (if Dom.eq_dec m (fst x) then true else false)) (d (vec2list l))) ->
  SInvM (veq_dec eq_dec) m (NQueryP m) (ADVloop n l d).
```

Being a more general lemma that what we need, we are forced to instantiate d with the body of our loop and show that if the adversary has not made the query about g^{xy} then any query chosen by the adversary does not depend on the hash value of g^{xy} present in the random oracle state.

The other sub goal is the *forces* of the *NQueryP* predicate for the adversarial loop:

```
Lemma Sforces_ADVloop : forall X m q ls l d,
  Sforces (fun x=> NQueryP m x)
          (fun a=>@ADVloop _ ls q l (d a))
          (fun _ : X => NQueryP m l).
```

This can be proved by showing that the *ADVloop* construct is monotonic with regards to the negated membership predicate and as a result, if a value has not been query at a certain point of execution then, forcefully, it has not been queried before that point.

The objective of doing this swap was to simplify the proof of the theorem *SEqP_SInvM_hash*:

```
Theorem SEqP_SInvM_hash : forall {A} eq_decA m (P:A->bool) d ,
  (forall h, SInvM eq_decA m P (d h)) ->
  SEqP (eqP_dec State.eq_dec eq_decA) P
    (h <<= hash m; d h)
    (h <$ Codom.to_list; d h) (emptyS Codom.T).
```

This lemma shows that if we start with the empty State then for a given event P, querying the random oracle is the same as randomly sampling h by assuming that the remaining game does not depend on the hash value of m stored in the random oracle state. In this proof's case, m will be the value g^{xy} and the event P will be event of the adversary not querying g^{xy}.

By only having to deal with empty state this lemma's proof is made simpler because if the state is empty then any initial query will not retrieve any value from the random oracle state and thus the random oracle will also randomly sample the value of h. The only difference is the storing in the random oracle state of the association between the h value and its correspondent hash value. If we hadn't swapped the loop with the sampling of h then, in order to apply the *SEqP_SInvM_hash* lemma, we would have to pass the first adversarial loop. This transition would alter the initial state, since the adversarial queries would populate the oracle, and thus the lemma could not be applied because of the non empty state.

We now apply *SEqP_SInvM_hash* that produces the goal in which we need to prove that the resulting game without the computing of h is *SInvM* with regards to *NQueryGxy*. This is proved by progressing through the game using *SInvM_bind* and by proving that each step of the game is *SInvM*. The main difficulties are both adversarial loops and, like before, the *SInvM_ ADVloop* lemma is used in order to achieve their proofs.

4.4 Conclusion

We have provided an in-depth look at the random oracle methodology and described the work done in the course of this thesis to extend the framework with capabilities to perform game-based cryptographic proofs in the random oracle model. To this end, a considerable amount of auxiliary definitions are needed, together with the associated lemmas. Due to space limitations only the statements of the main results were briefly alluded. The reader is deferred to the full development for further details. The use of this extension is illustrated with the proof of semantic security of Hashed ElGamal in the random oracle model along with the explanation of the main steps.

Chapter 5

Ssreflect

Ssreflect is an extension to the interactive theorem prover Coq developed by George Gonthier that was first used in the formalization of the Four Color Theorem. It allows the use of small scale reflection proofs and it is currently in version 1.2 that is compatible with Coq's version 8.2pl1.

As it name indicates, Ssreflect extends Coq with the use of small scale reflection which is characterized by the use of computation with symbolic representation [GM08]. Coq's use of reflection, in tactics like *ring* or *romega*, is performed in big scale where computational reflection is done in a transparent way to the user. In Ssreflect, the use of symbolic representation becomes visible to the user and this representation may appear in any lemma or sub-goal. The user is thus given control over the representation and he is able to perform translation from logical to symbolic representation. The main advantage of the use of small scale reflection is that different representations, as simple as they may be, provide useful procedures [GM08].

Ssreflect uses reflection to work with booleans as propositions thus enhancing the performance of propositional reasoning. It is then possible to use the computational behaviour of boolean predicates to perform rewriting in any goal because their logical equivalence can be seen as the equality of their value [Gon06]. This kind of reflection allows the use of classical reasoning, in spite of Coq's intuitionistic logic, because boolean predicates are decidable and thus inherently provide classical reasoning. Therefore, classical principles like the principle of excluded middle can be used when needed.

Despite the advantages of the boolean representation, logical propositions are still needed for some tasks, such as applying primitive tactics, and a way to transition back and forth between both representations is required. To this effect, Ssreflect defines the predicate *reflect* where *reflect* P b, for a proposition P and a boolean value b, indicates a reflection between P and $b = \mathit{true}$. For example,

```
Lemma orP : reflect (b1 \/ b2) (b1 || b2).
```

illustrates a reflection between the logical "or" and its boolean counterpart.

These kind of lemmas are called view lemmas and represent equivalences between the boolean and the propositional domain. View lemmas are generally postfixed by a "P". The application of view lemmas is characterized by the use of the "/" switch and it can be combined with most existing tactics. With a goal of the form:

```
(b1==b2) || b2
```

the command "apply/orP" replaces the boolean or with the logical one:

```
(b1==b2) \/ b2
```

There is no need to explicitly indicate the intended direction of the reflection because the view mechanism is capable of automatically inferring the right direction.

It is interesting to notice that, even though *(b1==b2)* or *b2* belong to the type *bool*, they appear to the user as being of type *Prop*. This comes as result of Ssreflect injecting booleans into propositions by the use of the `is_true` coercion:

```
Coercion is_true (b : bool) := b = true.
```

This means that a boolean expression can be used whenever a proposition is expected. These kind of coercions are transparent to the user.

Ssreflect goes beyond the addition of reflection and provides many general purpose features. The *rewrite* tactic was heavily extended. It is possible to rewrite more than one hypothesis in a single command and the mechanism of occurrence selection inside a *rewrite* is more robust. Additionally, the *rewrite* tactic can be used to simplify goals, to fold or unfold definitions and to prove resulting trivial goals:

```
rewrite /def {2}H1 /= H2 //.
```

This tactic unfolds the definition of *def*, rewrites the second occurrence of the pattern in the hypothesis *H1*, simplifies the goal, rewrites the hypothesis *H2* and tries to solve trivially the resulting sub-goals.

5.1 Bookkeeping

Ssreflect provides the user with an efficient way of doing bookkeeping during a proof, in which bookkeeping operations are responsible for moving things between the goal and the context while changing their shape [TZ08]. These operations can be seen as changes in data flow that are logically trivial, such as choosing the names of hypothesis, and thus do not actually prove anything new. Albeit this fact, these kind of step are ubiquitous in proof scrips and exist in great number.

Good bookkeeping proves to be important to define an organized structure to proof scripts and Ssreflect tries to achieve this by enforcing the explicit naming of all hypotheses and the removal of hypothesis that are no longer needed.

The tactic intros, used in Coq to introduce new hypothesis, gives the user the liberty to decide whether to supply or not a name for each hypothesis. In order to ensure good bookkeeping Ssreflect replaces the tactic intros with the tactic move that enforces a strict naming of hypothesis. This tactic is more powerful than intros and provides the user with trivial bookkeeping operations such as introducing new hypothesis and clearing existing ones. In addition to move, in Ssreflect almost every tactic can perform bookkeeping steps by using the tacticals '=>' and ':'. This avoids the repetitive use of tactics that uniquely perform bookkeeping steps.

```
move => a b <- []; apply: H.
```

This sequence of commands introduces the new variables/hypothesis a and b, rewrites with the hypothesis that is on top of the proof stack and immediately discards it and performs case-splitting. Finally it applies H and then clears it from the context.

5.2 Miscellaneous Features

In a big development it is virtually impossible to keep track of every proven lemma so the use of a good search mechanism becomes crucial. Ssreflect attempts to make the task of finding a specific lemma or browsing a list of existing ones easier by extending Coq's Search command. Additionally, Ssreflect employs a naming policy for the lemmas used in their library, for example, the use of the suffix 'A' to indicate an associativity lemma. This allows a better organization of the supporting library enhancing the library's usability and thus facilitating the search of existing lemmas.

Ssreflect aims to improve proof scripts' readability by providing indentation and bullets allowing the user to structure the several proof steps. Also, when a definition in a proof is changed, the proof should fail as soon as this new definition is used in the proof so that the effect of the change can be precisely pinpointed. This desired property is ensured in Ssreflect by the use of terminators. These are used to end a single sub-goal so that each line in the proof script should correspond to the elimination of one sub-goal thus making Ssreflect proofs more compact than the ones in regular Coq.

Ssreflect employs the use of compositional tactics, allowing the chaining of many simple tactics. This is encouraged instead of the use of automation tactics such as *auto*. Also, some of Coq's primitive tactics possess a complex behavior that often do more than what the user needed and whose resulting effect cannot always be precisely anticipated.

In that respect, the chaining of small tactics enhances proof readability by giving the user a fine-grained control over the operations that are being executed since each operation can be specified in detail. These changes employed by Ssreflect in proof script style may not seem much relevant but they prove useful in large developments by improving the flexibility and maintainability of proof scripts.

5.3 Ssreflect libraries

Ssreflect possesses a great number of supporting libraries which contain the formalization of several mathematical structures that were used in the proof of the four colour theorem and later in the Feit-Thompsom theorem. In Ssreflect's libraries, algebraic structures are represented as generic interfaces rather than actual modules. These algebraic structures are implemented with the purpose of providing common notation for expressions and for proofs so that modules can benefit from the composition of these structures (e.g. multiple inheritance) [GGMR09]. In order to package these structures, mixins were used to define both the operations and axioms satisfied by a structure which are then encapsulated with their representation type. As an example, the *eqType* structure that is used throughout the development to provide types with decidable equality [GMR+07] is defined as:

```
Record eqType_mixin (T : Type) : Type := EqTypeMixin {
  op   : sort -> sort -> bool;
  eqP  : forall x y , reflect (x = y) (op x y).
}.
Record eqType : Type := EqType {
  carrier :> Type;
  spec : eqtype_mixin carrier
}.
```

sort is a coercion of an *eqType* to its carrier type, *op* the equality operation on the type and *eqP* is the axiom that must be satisfied by any implementation of the *eqType*, that is, the reflection from *op* to the Leibniz equality operator. This reflection allows the use of *op* as a rewritable relation.

An example of a type that possesses such structure is the type of booleans in Ssreflect. In order to equip the *bool* type with an *eqType* structure we need to define the equality operator for booleans (*eqb*) and the proof of its reflection to the Leibniz equality (*eqP*). This allows the implementation of the *eqType* structure:

```
Definition bool_eqTypeMixin := @EqTypeMixin bool eqb eqP.

Definition bool_eqType := eqType bool_eqTypeMixin
```

With *bool_ eqType* defined as possessing the *eqType* structure, one would expect the automatic inference of $a == b$ as a well formed equality comparison for all a and b of type *bool* where $(==)$ is notation for the equality operation. However, such automatic inference from the type inferer does not occur because when it needs to find an *eqType* structure for the *bool* type it does not know that the correct one is *bool_ eqType*. Canonical structures allow the inference of a specific structure for a specific type and thus allows us to solve this problem by providing hints to Coq's unification algorithm about which type to infer when an eqType structure over bool is needed [GGMR09].

```
Canonical Structure bool_eqType.
```

The use of canonical structures also allows the inheritance of proofs and sharing of notations. Throughout the ssreflect libraries the use of mixins and canonical structures is systematic and this example is representative of the approach to building the libraries [GM10]:

- Implementation of modularity by defining generic abstract structures such as *eqType*.
- Development of definitions and properties of each structure.
- Instantiation of the generic structures, normally by the use of canonical structures. The *bool_ eqType* is an example of such instantiation.
- Development of the theory pertaining the specific types that were instantiated. These types inherit all results defined for its generic types due to the use of canonical structures.

5.4 Hashed ElGamal's proof in Ssreflect

In this section we will show how Ssreflect and its corresponding libraries can be used by adapting the Hashed ElGamal proof (standard model) in Nowak's framework in order to use all the group theory developed in Ssreflect's libraries. This also will require a formalization of bit string theory from scratch which provides a good example about the use of canonical structures.

5.4.1 Bit Strings

We start by defining the type of bit strings which are constructed by using the ssreflect's type *tuple* (lists of fixed length).

```
Definition bitstring := @tuple_of n bool.
```

n is a variable of type *nat* and tuple_of is a way to construct a *tuple* by indicating its size and the type of its elements. A *tuple n* is a subset of the *seq* type (analogous to Coq's type *list*) of *n* length. As usual, this is defined as a sequence and a proof that the sequence has size *n*:

```
Structure tuple_of : Type := Tuple {tval :> seq T; _ : size tval == n}.
```

By defining *tval* as a coercion, the following is considered well formed:

```
Variable bs : bitstring 5.
Check (size bs).
```

The type inferer considers this a well formed expression because, even though *size* receives as an argument an element of type *seq*, when needed the type inference mechanism can automatically use the *seq* projection of the *tuple* as the argument. The reverse unification can also be performed but it requires both the proof that the sequence has the required size and the use of canonical structures to help the unification algorithm. This mechanism will be used in the development of bit strings theory by first defining their operations and properties for the *seq* type and later transposing them to the *bitstring* type. We are only interested in sequences of booleans and this will be defined as the *bitseq* type:

```
Definition bitseq := seq bool.
```

We start with the definition of the group operation. In the *bitstring* type, the operation that possesses the desired properties is the xor operation:

```
Definition bsxor' (bs1 bs2 : bitseq) :=
  map [fun p => p.1 (+) p.2] (zip bs1 bs2).
```

(+) is a notation for the boolean xor and zip receives two *seq* as inputs and produces a *seq* with elements of each *seq* paired with each other. So that *bsxor'* can be used as the xor operation for bit strings, the proof that the *bsxor'* of two *bitseq* of size n also produces a *bitseq* of size n is required:

```
Lemma bsxorP : forall (bs1 bs2:bitstring),
 size (bsxor' bs1 bs2) == n.
```

With this proof, *bsxor'* is now also well formed with respect to bit strings. We use canonical structures to define the xor operation for bit strings from *bsxor'*:

```
Canonical Structure bsxor bs1 bs2 : bitstring :=
 Tuple (bsxorP bs1 bs2).
```

We now would like to show that the following group axioms are valid with respect to bits strings and its group operation : associativity, existence of the identity element and existence of the inverse element. In the Ssreflect libraries these properties are already defined and we need to prove these properties for the *bitstring* structure. First we prove that the group operation (*bsxor*) is associative, that is, $\forall$ (a b c : *bitstring*), $(a \cdot b) \cdot c = \; a \cdot (b \cdot c)$:

```
Lemma bsxorA : associative (@bsxor n).
```

The identity element for bit strings is the *bitstring* that is composed of only zeros and as before we will first define it as a *bitseq* and then promote it to *bitstring*:

```
Definition bszero' : bitseq := nseq n false.
```

nseq n t is the sequence of size n where all its elements have the value t. With the trivial proof that its size is indeed n we can use the canonical structure mechanism to promote it to the identity element of bit strings:

```
Lemma bszeroP : size bszero' == n.
Canonical Structure bszero : bitstring := Tuple bszeroP.
```

To prove that *bszero* is indeed the identity element for the *bsxor* we need to prove both the left and right identity properties, that is, $\forall\ (a : bitstring), e{\cdot}a\ =\ a{\cdot}e = a$ where e is the identity element:

```
Lemma bsxor0s : left_id  (@bszero n) (@bsxor n).
Lemma bsxors0 : right_id (@bszero n) (@bsxor n).
```

In order to show the existence of the inverse element it is enough to show that the *bsxor* of any *bitstring* with itself produces the identity element (*bszero*), that is, $\forall\ (a : bitstring), a \cdot a\ = e$:

```
Lemma bsxorss : self_inverse (@bszero n) (@bsxor n).
```

With all the group properties proved, we are now able equip bit strings with a group structure:

```
Definition BS : {set (bitstring n)} := setT.
Definition BS_groupMixin :=
  FinGroup.Mixin (@bsxorA n) (@bsxor0s n) (@bsxorss n).
Canonical Structure BS_baseFinGroupType :=
  Eval hnf in @BaseFinGroupType _ (BS_groupMixin).
Canonical Structure BS_finGroupType := FinGroupType (@bsxorss n).
Canonical Structure BS_group := Eval hnf in [group of BS].
```

The group formed by bit strings of size n is first defined as a set (BS). In order to equip bit strings with the finite group structure, besides the proved group properties, we also need to equip bit strings with the *FinType* structure. This structure represents properties about finite types and is automatically inferred on bit strings because they are defined as tuples that are already equipped with the *FinType* structure. As a result, we only need to instantiate the group mixin with the group properties. The group formalization by Ssreflect is divided by the *finGrouptType* and the *baseFinGroupType* and so both must be defined.

In order to prove that bit strings also form an abelian group it suffices to show that they possess the property of commutativity, that is, $\forall\ (a\ b : bitstring), a \cdot b\ = b \cdot a$, because we have now added a group structure to bit strings:

```
Lemma bsxorC : commutative (@bsxor n).
```

This enables us to prove that bit strings form an abelian group:

```
Theorem bitstring_abelian: forall s, abelian (BS_group s).
```

5.4.2 Security of Hashed ElGamal

To adapt the Hashed ElGamal's proof, the expected changes needed, besides a new formalization of bit string theory defined before, are the use of Ssreflect's notation and types (i.e. Ssreflect's group type) and the application of Ssreflect's group theory lemmas that correspond to the ones used by the framework. The changing of syntax is a trivial task and the replacement of the framework's lemmas only consists in finding the appropriate lemma in the libraries because all group results needed in Hashed ElGamal's proof are already formalized.

The main difficulty of the adaption was changing the use of Coq's list type to Ssreflect's list type. Even though these types are equivalent, this transformation is not as straight-forward as it may seem because of the following steps present in the proof: the proof that the exponentiation of a group's generator with a randomly sampled group element produces a random value and the proof that the xor of a bit string *bs* with a random bit string also produces a random bit string. The framework defines the tactic *permutation* to deal with these kind of proofs by showing that, taking as an example the bit string step, the set of bit strings (z) from which the sampling is performed is a permutation of the set that results from the xor of *bs* with any bit string of z. This tactic uses Coq's lists to define the set of possible sampling values and as a result the permutation is also defined over lists. This requires the formalization of an isomorphism between the *list* type and the *seq* type and this isomorphism is used in order to use Ssreflect's mechanisms and lemmas to prove the permutation. The use of this morphism is illustrated with the proof's step of the random sampling of a group element:

```
permutation (fun x => g ^+x).
```

g ^+ x is the notation for the modular exponentiation of the g by x. This application of the *permutation* tactic produces the following proof's goal:

```
Permutation (to_list G) (map [eta expgn g] (seqNE 0 #[g])
```

[*eta expgn g*] receives a natural n and returns g ^+ x and both the *Permutation* and *map* constructs work over Coq's lists. Since g is an element of the Ssreflect's group type, *Permutation* and *map* need to be changed into its Ssreflect's counterparts by using the morphism. As a result, this permutation can now be proved by using Ssreflect's libraries. The proof of the xor's step has an analogous approach and the rest of the proof is a spitting image of the original one and provides no further complications.

5.5 Conclusion

We have explored Coq's extension Ssreflect and its pervasive use of small scale reflection. Besides the use of reflection, it were shown several general purpose features: improved *rewrite* tactic and search mechanism, ability to perform good bookkeeping and the use of terminators and indentation that ease the organization of large developments. A great contribution of Ssreflect are its libraries which possess varied mathematical results and were used to adapt the Hashed ElGamal's security proof of Nowak's framework to take advantage of Ssreflect capabilities.

Chapter 6

Conclusion and Future Work

In this thesis we have studied the game-based approach to cryptographic security proofs and the interactive theorem provers that can be used as support for the formalization of frameworks according to this approach. In the first part of this thesis (Chapters 3 and 4) we explored David Nowak's framework for game-based security proofs and extended this framework with the ability to perform proofs in the random oracle model.

The framework does not take into account complexity issues as proofs do not depend on the security parameter. This is indeed a major flaw in the framework, because nothing prevents the user of instantiating a non-efficient adversary in a reduction to an hard-problem. An interesting research direction is to devise strategies for preventing such ill-formed adversarial instantiations.

The implementation of the random oracle model was mainly driven by the proof of Hashed ElGamal security. As a result, it would be interesting to use the framework on new cryptosystems in order to understand how the framework would fare against different proofs in ROM and how the framework could be improved in order to become more complete. One example of such proof would the Full Domain Hash Signature security proof since this involves the use of more than one oracle.

Like in any development, the implementation can always be target of upgrades either by implementing new features or by building more efficient ones. For example, the framework could store in the state monad, besides the oracle state, the adversarial view of the game. This would avoid the explicit data flow whenever the adversarial view is needed and the use of the *forces* construct could be avoided because the adversarial view of the game would be internally dealt with.

In the second part of this thesis (Chapter 5) we studied Coq's extension Ssreflect. Already being familiarized with Coq, we found Ssreflect's learning curve not to be much steep. The general purposes features provided by Ssreflect proved to be quite useful and user friendly and the use of small scale reflection is an effective way of dealing with many proofs. The most difficult part in this familiarization was the understanding of how Ssreflect's libraries were formalized and how we could take advantage of the structures formalized in the libraries.

We used Ssreflect's in Nowak's framework proof of ElGamal as an example by adapting part of the framework and this approach could be used for the whole framework. This would be interesting, not only because of the possibility of using reflection and Ssreflect's general purpose features, but mainly for taking advantage of all the mathematical results present in the libraries which are always prevalent in security proofs.

Bibliography

[APM09] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Sci. Comput. Program.*, 74(8):568–589, 2009.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[BCT04] Gilles Barthe, Jan Cederquist, and Sabrina Tarento. A machine-checked formalization of the generic model and the random oracle model. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR*, volume 3097 of *Lecture Notes in Computer Science*, pages 385–399. Springer, 2004.

[Bel98] Mihir Bellare. Practice-oriented provable security. In Ivan Damgård, editor, *Lectures on Data Security*, volume 1561 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998.

[BGB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 90–101. ACM, 2009.

[Bla07] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *CSF*, pages 97–111. IEEE Computer Society, 2007.

[BP06] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 2006.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[BR94] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *EUROCRYPT*, pages 92–111, 1994.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with rsa and rabin. In *EUROCRYPT*, pages 399–416, 1996.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

[BSS99] Ian F. Blake, G. Seroussi, and N. P. Smart. *Elliptic curves in cryptography.* Cambridge University Press, New York, NY, USA, 1999.

[Can97] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 1997.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CMR98] Ran Canetti, Daniele Micciancio, and Omer Reingold. Perfectly one-way probabilistic hash functions (preliminary version). In *STOC*, pages 131–140, 1998.

[Del00] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[dt09] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2009. Version 8.2.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[Gam85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[GHR99] Rosario Gennaro, Shai Halevi, and Tal Rabin. Secure hash-and-sign signatures without the random oracle. In *EUROCRYPT*, pages 123–139, 1999.

[GM82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *STOC*, pages 365–377. ACM, 1982.

[GM08] Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. Research report, INRIA, 2010.

[GMR$^+$07] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Schneider and Brandt [SB07], pages 86–101.

[Gon06] Georges Gonthier. Notations of the four colour theorem proof. Technical report, Microsoft Research, 2006.

[GW07] François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In Schneider and Brandt [SB07], pages 368–382.

[Hal05] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. `http://eprint.iacr.org/`.

[JN03] Antoine Joux and Kim Nguyen. Separating decision diffie-hellman from computational diffie-hellman in cryptographic groups. *J. Cryptology*, 16(4):239–247, 2003.

[KR96] Joe Kilian and Phillip Rogaway. How to protect des against exhaustive key search. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 252–267. Springer, 1996.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[Now07] David Nowak. A framework for game-based security proofs. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2007.

[Now09] David Nowak. On formal verification of arithmetic-based cryptographic primitives. *CoRR*, abs/0904.1110, 2009.

[SB07] Klaus Schneider and Jens Brandt, editors. *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.

[Sho01] Victor Shoup. Oaep reconsidered. In Joe Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259. Springer, 2001.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. http://eprint.iacr.org/.

[Sma05] Nigel Smart. Provable security: Designs and open questions, 2005.

[TZ08] Enrico Tassi and Mura Anteo Zamboni. Interactive theorem provers: issues faced as a user and tackled as a developer. Technical report, University of Bologna, 2008.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, Mai. 1994.