cryptolib: Security Proofs in the Lean Theorem Prover

Joey Lupo

Master of Science

Cyber Security, Privacy and Trust

School of Informatics

University of Edinburgh

2021

Abstract

Cryptographic protocols secure communication over a public channel, protecting our data and dollars on the web from attackers. It is imperative that we be able to mathematically prove that a protocol satisfies some precise notion of security. However, writing security proofs is a notoriously complex and error-prone process. As a result, cryptographers and security researchers have increasingly looked to theorem provers as a way to improve trust in security proofs. In this paper, we present cryptolib, a framework for security proofs in the Lean theorem prover. As a proof of concept, we formalize the proofs of correctness and semantic security for the ElGamal public key encryption protocol. We further compare cryptolib to similar projects in other theorem provers and discuss advantages and disadvantages of working within the Lean ecosystem.

Acknowledgments

- Professor Paul Jackson, who supervised this project, for introducing me to formal verification, and providing valuable feedback as to the direction of the project and the content of this dissertation.

- Ramon Fernández Mir, the PhD student of Prof. Jackson, for helping me overcome a number of early stumbling blocks in Lean. Your tips saved me literally tens of hours that would have been spent debugging and searching for solutions.

- The Lean community on Zulip, for helping with several questions regarding this thesis.

- The St. Andrew's Society of the State of New York, for awarding me a scholarship to study at a Scottish university. Without your generous support, continuing my studies at the prestigious University Edinburgh would not have been possible.

- Amherst College, for awarding me an Amherst College Fellowship to continue my studies in graduate school. I had a wonderful four years at Amherst, and the College is the gift that keeps on giving (literally, in this case).

- My parents, for always supporting me and for providing a wonderful environment in which to work during this hectic past year.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Joey Lupo*)

Table of Contents

1 Introduction **1**

2 Background **3**

2.1 The Lean theorem prover 3

2.2 Provable security 6

2.3 Related work 8

3 Provable Security in Lean **11**

3.1 Modeling games as `pmfs` 11

3.2 Public key encryption protocols 14

3.2.1 Correctness 16

3.2.2 Semantic security 17

3.3 Proving equivalence between `pmfs` 19

3.4 Negligibility 23

4 ElGamal **25**

4.1 The protocol 26

4.2 Proof of correctness 27

4.3 The DDH assumption 29

4.4 Proof of semantic security 30

5 Evaluation and Comparison **35**

6 Conclusion and Future Work **40**

Bibliography **41**

A A Step-by-Step Proof in Lean **46**

B Summary of cryptolib **48**

Chapter 1

Introduction

A theorem prover is a computer program which provides the means for writing formal specifications and formal proofs. In contrast to proofs found in mathematical textbooks or research papers, formal proofs can be machine checked for correctness, meaning all deductions can be traced back to a base set of axioms [AdMK21, pp. 1–2]. Producing formal proofs can be a tedious process; every step must be rigorously proven with no room for intuition or proof by authority. Consequently, theorem provers often interactively track assumptions and proof goals, as well as provide libraries and tools to ease the burden on the user.

Lean is one such theorem prover which has gained traction in recent years, particularly among mathematicians. The primary goal of project is to produce a case study development in Lean. To that end, we present cryptolib,[1] a Lean library of 7 files, 883 lines of code, 23 definitions, 12 theorems, and 25 lemmas. In Appendix B, we summarize the contents of cryptolib. In cryptolib, we demonstrate one way to formalize proofs of security for cryptographic protocols in Lean. In particular, we formalize the properties of correctness and semantic security for public key encryption protocols, and use this formalization to verify the proofs that the ElGamal public key encryption protocol satisfies correctness and semantic security. At the heart of our formalization is the `pmf` (probability mass function) definition from mathlib, Lean's mathematical library. In addition to proving several lemmas for future inclusion in mathlib, we provide two new tactics to help prove equivalences between `pmf`s.

A secondary goal of this project is to compare our development with previous work in other theorem provers in order to better understand the advantages and disadvantages of working within the Lean ecosystem. In Ch. 5, we provide an extended comparison of

[1] Available at `https://github.com/Loops7/cryptolib`.

cryptolib with a selection of previous projects in other theorem provers, namely EasyCrypt (Coq) [BGHB11], CryptHOL (Isabelle/HOL) [BLS20], Nowak's toolbox (Coq) [Now07, Now08], and the Foundational Cryptographic Framework (Coq) [PM15, Pet15].

To motivate cryptolib, consider that cryptographic protocols secure our data and dollars on the web from attackers. Designing a secure cryptographic protocol is a very difficult proposition. As one observer notes, "Since the appearance of public key cryptography in the Diffie-Hellman seminal paper, many schemes have been proposed, but many have been broken" [Poi05, p. 133]. Take for example the Needham-Schroeder public key encryption protocol [NS78], later shown broken in [Low96], or the Chor-Rivest encryption protocol [CR88], later shown broken in [Vau98]. Any protocol which is to be taken seriously needs an accompanying proof of security, a concept we explore in Sec. 2.2.

As protocols increase in complexity, so do the corresponding security proofs: "many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor" [BR04, p. 3]. In order to address this "crisis of rigor," many researchers in security have looked to theorem provers as a way to improve trust in new results, see e.g. [ANY12, BGHB11, BLS20, Now07, Now08, PM15]. Cryptolib represents the first such effort in Lean.

This paper is organized as follows. In Ch. 2, we provide necessary background information on the Lean theorem prover and provable security, and give a brief survey of existing projects in other theorem provers for formalizing security proofs. In Ch. 3, we introduce the formula for modeling games as probability distributions via mathlib's `pmf` object and Lean's `do` syntax. Next, we use the language of games to formalize correctness and semantic security of a public key encryption protocol. We use the definitions from Ch. 3 to formalize the proofs of correctness and semantic security for the ElGamal public key encryption protocol in Ch. 4. In Ch. 5, we provide a critical evaluation of cryptolib and compare our development with similar developments in other provers. We conclude in Ch. 6 with a brief discussion of ways to extend cryptolib.

Chapter 2

Background

In this chapter, we give the background information necessary for subsequent chapters. In Sec. 2.1, we introduce the Lean theorem prover, using [AdMK21] as a reference. We place Lean in context by mentioning a number of other theorem provers. In Sec. 2.2, we draw from [Poi05] to give an overview of the different sets of assumptions and models used in security proofs for cryptographic protocols. The chapter concludes in Sec. 2.3 with a brief survey of tools and libraries for formalizing security proofs in other theorem provers.

2.1 The Lean theorem prover

Leonardo de Moura originally developed the Lean theorem prover[1] at Microsoft Research in 2013. The latest version is Lean 4, which is not backwards compatible with previous versions. Lean 3 is still maintained by the Lean community and includes mathlib,[2] an extensive library of formalized mathematics. The mathlib project started as an effort by Professor Kevin Buzzard of University College London (UCL) to formalize the entire undergraduate math curriculum at UCL, but has since grown into a diverse community of mathematicians and computer scientists working to formalize entire swaths of mathematics. For a broad overview of what is and what is not in mathlib, see [3]. The Lean community is also active on the Zulip chat platform,[4] where users ask questions, discuss formalization efforts, and coordinate work. Major projects in Lean 3 include the formalization of schemes (from algebraic geometry) [BHL+21]

[1] https://leanprover.github.io/

[2] https://leanprover-community.github.io/index.html

[3] https://leanprover-community.github.io/mathlib-overview.html

[4] https://leanprover.zulipchat.com/

and perfectoid spaces [BCM20], and the Liquid Tensor Experiment [Sch21], wherein a small team of mathematicians formalized a cutting-edge proof by mathematician Peter Scholze. Due to mathlib and its active community on Zulip, our project is written in Lean 3. Henceforth any mention of "Lean" is understood to mean Lean 3. Two popular introductions to Lean are "Theorem Proving in Lean" [AdMK21] and The Natural Number Game,[5] where players prove facts about the natural numbers using only the Peano axioms.

In the following exposition, we draw from [AdMK21, Chs. 2–5] to develop the basics of what it means to prove a theorem in Lean, and how we prove a theorem in practice. The logical foundation of Lean is a type theory known as the *calculus of inductive constructions* (CIC), wherein every expression is considered as a *term* of an associated *type*. In Lean, the syntax `(a : A)` means `a` is a term of type `A`. For two types `A` and `B`, we denote by `A × B` the type of coordinate pairs where the first coordinate has type `A` and the second coordinate has type `B`. We denote by `A → B` the type of functions which take as input a term of type `A` and output a term of type `B`. Lean is also a functional programming language, so that applying a function `(f : A → B)` to `(a : A)` is denoted `f a`, i.e. `(f a : B)`. Users can define new functions using lambda notation. For example, `(λ (a : ℤ), 2 * a)` denotes a term of type $\mathbb{Z} \to \mathbb{Z}$ which satisfies `(λ (a : ℤ), 2 * a) m = 2 * m` for `(m : ℤ)`.

Constructing new types from other types, as we did with $\alpha \times \beta$ and $\alpha \to \beta$, falls under the umbrella of *simple type theory*. Lean provides the capabilities to define types which exist outside of simple type theory. For instance, Lean has *inductive types*, which are defined with constructors and recursion. The natural numbers $\mathbb{N}$ can be defined as an inductive type via the constructors $0 : \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$. Lean also has *dependent function types*, where the type of the output of the function depends on the type of the input. Even more, types have a type! The type of `Type` is `Type 1`, and the type of `Type 1` is `Type 2`, and so on.

In CIC there is a special type called `Prop` which corresponds to a mathematical statement. Considering a proposition as a type is also known as the *Curry-Howard isomorphism* [AdMK21, Sec. 3.1]. Under this isomorphism, we "prove" a theorem, formally specified as a term `p` of type `Prop`, by exhibiting a term of type `p`. In other words, `p` is the type of proofs of `P`. Consider the following formal definition `is_even` for an even integer:

[5] `https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/`

```
def is_even (n : ℤ) : Prop := ∃ m, n = 2 * m
```

The function `is_even` takes as input an integer and returns a term of type `Prop`. (Note that Lean supports unicode characters, so `is_even` appears exactly as it would in a Lean file.) To prove that 2 is even according to our Lean definition is to exhibit a term of type `is_even 2`. In practice, we build proof terms by using Lean's *tactic mode*. In tactic mode, Lean provides a panel which lists all current assumptions, as well as the current *goal*, which is what we have left to prove. The initial goal state is exactly the theorem we want to prove. Then we iteratively apply tactics to transform the goal until it is an axiom or a tautology. Generally, tactics incorporate a certain level of automation so that we are not manually proving deductions down to the base axioms of Lean. Below is a tactic mode proof of `is_even 2`:

```
def is_even_two : is_even 2 := by {use 1, ring}
```

The `by{...}` syntax suffices to enclose short tactic mode proofs. For the longer proofs of later sections, we delineate tactic mode via the `begin ... end` syntax. The `use` tactic helps to solve goals which involve the existential quantifier ∃. Recall from above that the goal `is_even 2` is proving `∃ m, 2 = 2 * m`. In effect, `use 1` makes the claim to Lean that `1` is the integer `m` such that `2 = 2 * 1`, and so the goal is transformed to `2 = 2 * 1` in ℤ. Finally, the `ring` tactic applies algebraic identities in the ring ℤ to automatically close the goal.

Working within tactic mode is often a process of trial and error which involves trying out many different lines of attack on the goal. In response, Lean provides real time feedback to each tactic application by changing the hypotheses and/or the goal. In this sense, Lean is an *interactive* theorem prover. Certain tactics such as `ring` and `simp` can automatically prove more complicated goals, and `library_search` can check the database of results in Lean to try to solve the current goal. Hence, Lean is simultaneously an *automatic* theorem prover. Documentation for the dozens of tactics in Lean and mathlib can be found at [6]. In Appendix A, we give a step-by-step walkthrough of a non-trivial proof in Lean to better demonstrate tactic mode.

Lean exists within a diverse ecosystem of theorem provers. Its closest relative is Coq [dt21], which also based on CIC. Coq includes many of the same features as Lean

[6] `https://leanprover-community.github.io/mathlib_docs/tactics.html`

such as tactic mode and is generally comprehensible to someone well versed in Lean. Agda [BDN09] is another theorem prover which uses CIC as its logical foundation, but unlike Lean and Coq, Agda does not provide a tactic-mode for constructing proof terms. Isabelle [NPW02] is a proof assistant which can implement several different logics but which is usually paired with higher order logic (HOL) to yield Isabelle/HOL. Unlike theorem provers based on CIC, Isabelle/HOL cannot express dependent types. However, Isabelle/HOL is generally better than tactic-based provers at using automation to dispense goals. ACL2 [KMM13] is a theorem prover based on a different logical framework than HOL or CIC. In particular, ACL2 is based on quantifier-free first-order logic. Researchers have used ACL2 extensively in the domains of software and hardware verification.

2.2 Provable security

Since we rely on cryptographic protocols to secure the modern web, it is paramount that the protocols guarantee some form of security. Historically, such guarantees fell short of mathematical proof: "The simple fact that a cryptographic algorithm withstood cryptanalytic attacks for several years has often been considered as a kind of validation procedure" [Poi05, p. 134]. As the name implies, *provable security* is the attempt to provide such mathematical proofs of security.

Provable security dates to the work of Rabin [Rab79], who presented an encryption protocol based on integer factorization and proved that an attacker A can efficiently break the protocol only if A can also factor an integer that is the product of two large primes. The problem of integer factorization is considered *hard*, in the sense that it is believed that there is no efficient algorithm which solves it. All security proofs for cryptographic protocols rely on some problem assumed to be hard, which we term alternatively as a *complexity* or *hardness assumption*. In Sec. 4.3, we formalize the decisional Diffie-Hellman (DDH) problem, and in Sec. 4.4 we show that breaking the ElGamal encryption protocol implies solving the DDH problem.

There are several models of computation for security proofs for cryptographic protocols. In the *standard model*, we work only with a complexity assumption. Security is proven by showing that any attacker A which can efficiently break the protocol can also efficiently solve the problem assumed to be hard [Poi05, p. 137]. Since it can be difficult to provide such reductions for complex protocols, researchers have developed other models of computation which approximate the standard model but ease the proof

burden. For instance, in the *random oracle model*, any cryptographic hash function which a protocol uses is treated instead as an idealized hash function [BR93]. The version of ElGamal encryption for which we prove security uses no hash function, and so our proof is given in the standard model.

Besides specifying the model of computation, security proofs of cryptographic protocols exist in one of two models: the Dolev-Yao model and the computational model. (Note a "model" of computation and the Dolev-Yao/computational "model" coexist within the same set of assumptions; the use of "model" is overloaded and often confusing.) In the *Dolev-Yao* (or *symbolic*) *model*, an attacker cannot break any cryptographic primitive. For example, if a protocol uses an encryption function in its execution, we assume that an attacker cannot break the encryption. Proofs in the Dolev-Yao model often employ algebraic methods since cryptographic primitives can be treated as black-boxes [Bla12, p. 5].

On the other hand, in the *computational model*, we assume that an adversary can break a cryptographic primitive with negligible probability. In general, results in the computational model are harder to achieve, but are preferred since they more closely model the real world. All proofs in cryptolib are in the computational model.

An *efficient algorithm* as it is used in the preceding exposition is informally defined as a probabilistic algorithm which runs in time polynomial in a fixed security parameter $\eta \in \mathbb{Z}_{>0}$. The difficulty of the complexity assumption is also a function of η (e.g. the integer to factor is an η-bit number), as is the probability that an attacker can break a cryptographic protocol in the computational model. A completely rigorous treatment of the computational model (as defined by the terms "hardness assumption", "efficient algorithm", and "negligible function") in terms of the security parameter η is given in [BS15, Sec. 2.3]. Except for a brief detour in Sec. 3.4, we omit the security parameter in our discussion and formalization. This is possible by showing that the value we are interested in is exactly equal to a different value assumed to be negligible by a hardness assumption.

Finally, we introduce the so called *game hopping* or *game-based* method for structuring security proofs in the computational model, as suggested by [Sho04] and [BR04]. In this methodology, we phrase security as an *attack game* (or simply *game*) between an efficient attacker A and an impartial challenger C. Consider the toy game G:

1. C samples from $\mathbb{Z}_2$ uniformly at random, and secretly writes down the sampled bit b.

2. A runs an efficient algorithm and submits a guess b'.

We say that A wins the game G if $b = b'$. Denote by $G(A)$ the event that A wins the game. We outline how a game-based proof of $\Pr[G(A)] = 1/2$ might proceed.

In this paper, we follow Shoup [Sho04] in considering games as random variables on $\{0,1\}$, or equivalently $\mathbb{Z}_2$, the integers modulo 2. Consider the random variable $G_0(A)$ defined by the following procedure:

1. Sample b from $\mathbb{Z}_2$ uniformly at random.
2. Sample b' from A (considered as a probabilistic, polynomial-time algorithm).
3. Return $(1+b+b') \in \mathbb{Z}_2$

By construction, the event $G(A)$ is $G_0(A) = 1$, so that $\Pr[G(A)] = \Pr[G_0(A) = 1]$. To reason about $\Pr[G(A)]$, we construct a sequence of intermediate games $G_1(A), G_2(A), \ldots$ which satisfy $\Pr[G_i(A) = 1] = \Pr[G_{i+1}(A) = 1]$ for each i, until we arrive at $G_n(A)$ defined by

1. Sample b from $\mathbb{Z}_2$ uniformly at random.
2. Return $b \in \mathbb{Z}_2$.

Then it is obvious that $\Pr[G_n(A) = 1] = 1/2$. Hence we deduce that A can do no better than random guessing in the initial game. The name "game hopping" refers to the fact that we arrive at $G_n(A)$ by "hopping" from $G_0(A)$ to $G_1(A)$, to $G_2(A)$, and so on. In general, hops where $\Pr[G_i(A) = 1]$ and $\Pr[G_{i+1}(A) = 1]$ are negligibly close are also permitted in game-based proofs.

Note that the random variable $G_0(A)$ induces a distribution on $\mathbb{Z}_2$. We often abuse notation and refer to a game as a probability distribution with the understanding that this induced distribution on $\mathbb{Z}_2$ is the one to which we refer. In Sec. 3.2.2, we define the semantic security game for a public key encryption protocol and define semantic security as the property that any efficient attacker A can do no better than random guessing to win the game. In Sec. 4.4, we formally prove semantic security for the ElGamal encryption protocol using the game-based proof from [Sho04].

2.3 Related work

Due to the complexity of security proofs for cryptographic protocols, and the potentially catastrophic consequences of an incorrect proof, researchers have increasingly

turned to theorem provers within the last fifteen years to address the "crisis of rigor" [BR04, p. 3]. Early projects such as ProVerif [Bla01] provided libraries for formalizing proofs in the Dolev-Yao model. Due to the model's simplifying assumptions, the current state of formalization in the Dolev-Yao model is quite mature [Bla12, p. 19].

Much of the recent activity in the field concerns formalizing and automating game-based proofs in the computational model. Two distinct paradigms have emerged to model games in a theorem prover: deep embedding and shallow embedding. In a *deep embedding*, games are not considered as random variables or distributions. Instead, games are syntactic objects, i.e. code written in a programming language, and game hops are formal syntactic manipulations. This is the approach suggested by Bellare and Rogaway [BR04], who offer a sample formal language $\mathcal{L}$ in which to write game-based proofs. To deeply embed games within a theorem prover means to write an additional language within the host language. EasyCrypt [BGHB11, BDG+13] is one framework which uses a deep embedding in Coq to formalize security proofs. A game-based proof of semantic security for a hashed version of ElGamal is used as a proof of concept. In general, automation of proofs is easier in a deep embedding, but comes at the cost of writing a significant amount of code to get off the ground [Now08, p. 369] and expanding the trusted computing base beyond the kernel of the theorem prover in which the language is deeply embedded [PM15, p. 68].

On the other hand, a *shallow embedding* of games within a theorem prover follows Shoup [Sho04] by modeling a game as a random variable, or equivalently, the probability distribution induced by a random variable. Thus the problem of modeling games in a theorem prover is reduced to formally defining probability distributions in the theorem prover. In turn, a game hop is a proof in the theorem prover relating distributions. Importantly, these proofs rely solely on the axioms of the theorem prover and can leverage any existing tactics or libraries within the host ecosystem [Pet15, p. 33].

One of the first projects to use a shallow embedding of games as probability distributions was Nowak's Coq toolbox [Now07]. As in cryptolib, Nowak proves semantic security for ElGamal under the decisional Diffie-Hellman assumption as a proof of concept. He further proves that the hashed version of ElGamal is semantically secure in the random oracle model under the computational Diffie-Hellman assumption. In [Now08], Nowak extends his toolbox by formalizing the quadratic residuosity problem in order to formalize security proofs for two other cryptographic protocols.

In [PM15], Petcher and Morrisett introduce the Foundational Cryptographic Framework (FCF), a library which uses a shallow embedding in Coq similar to that of Nowak.

FCF incorporates some aspects of automation and can model more complex game-hops than the toolbox provided by Nowak. FCF was originally outlined in Adam Petcher's PhD thesis [Pet15], wherein he uses FCF to formalize the proof of semantic security for ElGamal and hashed ElGamal, as well as the Searchable Symmetric Encryption scheme of Cash et. al. [CJJ+13].

Another recent project is CryptHOL [BLS20, Loc17, LSB17], which uses a shallow embedding of games for security proofs in Isabelle/HOL. Example proofs include semantic security for ElGamal and hashed ElGamal. Researchers have also adapted CryptHOL to formalize security proofs for secure multi-party computation protocols [BAG17], oblivious transfer protocols [BAG20], and commitment schemes [BLAG21].

Chapter 3

Provable Security in Lean

In this chapter, we describe how cryptolib formalizes security proofs in Lean via a shallow embedding of games. In Sec. 3.1, show how mathlib's `pmf` monad is used with Lean's `do` syntax to model games. In Sec. 3.2, we use the `pmf` + `do` combination to formalize the properties of correctness and semantic security for a public key encryption protocol. Next, we explain in Sec. 3.3 how we can use the monadic laws as well as two custom tactics from cryptolib to prove that two `pmf`s are equal. We describe a stand-alone formalization of negligible functions in Sec. 3.4.

3.1 Modeling games as `pmf`s

Cryptolib uses a shallow embedding of games as probability distributions in Lean, following the description by Shoup [Sho04]. We can define a distribution over the finite sample space $\mathbb{Z}_2$ via a **probability mass function** (**pmf**), which in this setting is a function $\mathbb{Z}_2 \to \mathbb{R}_{\geq 0}$ which assigns to each element in the domain its probability in the underlying distribution. For example, a coin flip (where "Heads" and "Tails" are identified with 0 and 1) corresponds to the pmf $f : \mathbb{Z}_2 \to \mathbb{R}_{\geq 0}$ given by $f(0) = 1/2$ and $f(1) = 1/2$. More generally, we can consider probability mass functions over $\mathbb{Z}_q$ where q is a positive integer. For instance, the pmf $r : \mathbb{Z}_6 \to \mathbb{R}_{\geq 0}$ given by $r(a) = 1/6$ for all $a \in \mathbb{Z}_6$ corresponds to the roll of a fair six-sided die. Even more generally, we can define a probability mass function on any finite group.

Mathlib provides a definition of a probability mass function via the `pmf` definition in `measure_theory/probability_mass_function.lean` (which we simplify slightly for the purposes of exposition):

```
def pmf (α : Type) : Type := {f : α → ℝ ≥ 0 // has_sum f 1}
```

In mathematical terms, for a given type α, `pmf α` is the type of functions $\alpha \to \mathbb{R}_{\geq 0}$ for which $\sum_{a \in \alpha} f(a)$ is well defined and has value 1. Since a probability mass function uniquely specifies a distribution, we often refer to a `pmf` as a distribution. In the cryptolib file `uniform.lean`, we define several special `pmf`s. First, the uniform distribution on finite groups:

```
noncomputable theory
variables (G : Type) [fintype G] [group G] [decidable_eq G]
def uniform_grp : pmf G :=
pmf.of_multiset (fintype.elems G).val (group.multiset_ne_zero G)
```

We describe the effect of the `variables` command in Section 3.2 since it is not important for our purposes yet. The commands `[fintype_eq G]`, `[group G]`, `[decidable_eq G]` are type class requirements (see [AdMK21, Ch. 10]). Informally, the type class requirements mean that anyone who wants to use the definition `uniform_grp` for a type `G` must supply to Lean a proof that `G` satisfies the formal properties of a `fintype` (finite type), `group`, and `decidable_eq` (decidable equality). In the file `to_mathlib`, we supply proofs that the type `zmod q` is an `instance` of a `group` and `fintype` for all positive integers `q`. As we discuss in Ch. 5, we avoid having to supply an instance proof of `decidable_eq` by invoking classical logic with the `noncomputable` keyword. The two instance proofs and `noncomputable theory` let us define `pmf`s over `zmod q`.

The function `pmf.of_multiset` from mathlib returns a `pmf` which corresponds to the uniform distribution over a finite type, but we have to provide the proof `group.multiset_ne_zero G` (from the cryptolib file `to_mathlib`) of the fact that our arbitrary finite group `G` is non-empty. Next, we define the special case of $G = \mathbb{Z}_q$ using the `zmod` definition from mathlib:

```
import data.zmod.basic
def uniform_zmod (n : ℕ) [fact (0 < n)] : pmf (zmod n) :=
   uniform_grp (zmod n)
```

The bracketed `[fact (0 < n)]` is again a type class requirement that requires the provided `n` to satisfy the fact `0 < n`. Since we so often need the uniform distribution

on $\mathbb{Z}_2$, we define a further shortcut with `uniform_2`:

```
def uniform_2 : pmf (zmod 2) := uniform_zmod 2
```

Because we use the `pmf.multiset` function to define `uniform_grp`, it actually takes some effort to prove the following lemma:

```
lemma uniform_grp_prob : ∀ (g : G),
    (uniform_grp G) g = 1 / multiset.card (fintype.elems G).val
```

In words, `uniform_grp_prob` states that the probability of any element $g \in G$ is $1/|G|$ for the `pmf` defined by `uniform_grp`. Similarly, we prove the special case of the previous lemma for `uniform_zmod`:

```
lemma uniform_zmod_prob {n : ℕ} [fact (0 < n)] : ∀ (a : zmod n),
   (uniform_zmod n) a = 1 / n
```

The use of the curly braces for the parameter `{n : ℕ}` tells Lean that `n` is implicit; it can be inferred from `(a : zmod n)`. Hence for `(a : zmod n)`, `uniform_zmod_prob a` is a proof of `(uniform_zmod n) a = 1 / n`. If we used parentheses `(n : ℕ)` instead, we would have to invoke the lemma via `uniform_zmod_prob n a`.

To fully model games, we need more than just static distributions. Recall how we constructed the distribution $G_0(A)$ in Sec. 2.3 by a executing a sequence of samples from different distributions and then returning some function on those sampled values. It turns out that this is exactly the behavior of a *monad*. The recognition that probability distributions have a monadic structure is due to Giry [Gir82]. In particular, a `pmf` is a monad.

Let α and β denote types. That `pmf` is a monad means it comes equipped with two special functions (in mathlib), `pure` and `bind`:

```
def pure (a : α) : pmf α

def bind (p : pmf α) (f : α → pmf β) : pmf β
```

which satisfy three relations, known as the monad laws. We present the monad laws for `pmf`s in Sec. 3.3. For `(a : α)`, `pure a` defines the distribution on α which assigns

probability 1 to `a` and 0 to all `a' ≠ a`. Informally, `p.bind f` is the distribution which results from sampling from `p` and passing the sampled value to `f`.

The `bind` function is exactly how we chain together a sequence of samples to create a new distribution, as in $G_0(A)$. In fact, we can model $G_0(A)$ with a `pmf`:

```
def G_0 (A : pmf (zmod 2)) : pmf (zmod 2) :=
  (uniform_zmod 2).bind (λ b, A.bind (λ b', pure (1 + b + b')))
```

Lean provides a nicer way to define the same distribution using the `do` syntax:

```
def G_0 (A : pmf (zmod 2)) : pmf (zmod 2) :=
do
  b ← uniform_zmod 2,
  b' ← A,
  pure (1 + b + b')
```

The `do` syntax makes the presentation of the distribution much more clearly resemble the description of $G_0(A)$. The `pmf` + `do` syntax is the crucial combination which cryptolib leverages to formalize game-based proofs in Lean. In addition to access to all of formalizations and tactics in mathlib, the expressiveness of the `pmf` + `do` syntax justifies our choice to use a shallow embedding for cryptolib.

3.2 Public key encryption protocols

Public key encryption is a fundamental cryptographic primitive that offers a way for two parties to communicate a private message over a public channel without having a pre-shared private key. From [BS15, Def. 11.1], we define a **public key encryption protocol** $\mathcal{P} = (K, E, D)$ as a triple of algorithms:

- K is an efficient key generation algorithm which takes no inputs and returns a pair (pk, sk), where pk denotes the public key and sk denotes the private (or secret) key,
- E is an efficient encryption algorithm which takes as input a public key pk and a message m and returns a ciphertext c,
- D is a deterministic decryption algorithm which takes as input a private key sk and a ciphertext c, and returns a message m.

Further, we assume for $\mathcal{P}$ that all messages are elements of some set $\mathcal{M}$, known as the message space. Similarly, all ciphertexts are elements of a ciphertext space $\mathcal{C}$.

Suppose Alice wants to use $\mathcal{P}$ to communicate private information to her friend Bob over a public channel. One solution is for Bob to run K to obtain a key pair (pk, sk). He then posts the public key pk on the public channel for all to see. Now Alice encrypts her plaintext message m using E and sends the ciphertext $c = E(pk, m)$ over the public channel. Bob uses D to decrypt c and obtain a plaintext $m' = D(sk, c)$.

If we want $\mathcal{P}$ to actually implement private communication, we require that $m' = m$ and that there is some guarantee that it is hard for an attacker on the public channel to obtain m from c. These basic properties of correctness and security are introduced and formalized in Sec. 3.2.1 and Sec. 3.2.2, respectively. In Sec. 3.2.1 and Sec. 3.2.2, we fix the following notation:

$$\begin{aligned} \mathcal{P} = (K, E, D) &:= \text{a public key encryption protocol,} \\ \mathcal{M} &:= \text{the message space for } \mathcal{P}, \\ \mathcal{C} &:= \text{the ciphertext space for } \mathcal{P}. \end{aligned}$$

These hypotheses are formalized at the top of `pke.lean` as follows:

```
noncomputable theory
variables {G1 G2 M C: Type} [decidable_eq M]
          (keygen : pmf (G1 × G2))
          (encrypt : G1 → M → pmf C)
          (decrypt : G2 → C → M)
```

We use the more descriptively named `keygen`, `encrypt`, and `decrypt` to formalize K, E, and D. Here `G1` is the type of public keys and `G2` is the type of private keys. The curly braces tell Lean to infer `G1`, `G2`, `M`, and `C` from `keygen`, `encrypt`, and `decrypt`. By declaring `keygen`, `encrypt`, and `decrypt` at the top of `pke.lean` as `variables`, all subsequent definitions in the file which reference these variables will be given an additional parameter. As in Sec. 3.1, `[decidable_eq M]` is a type class requirement which requires anyone who provides instance of an `encrypt` function (e.g. to use the later definitions in the file) to also provide a proof that the type `M` satisfies decidable equality. We avoid having to supply such a proof by declaring `noncomputable theory`, as we discuss in Ch. 5.

3.2.1 Correctness

Drawing from [BS15, Def. 11.1], we say $\mathcal{P}$ satisfies **correctness** if for all possible key pairs (pk, sk) output by K, and all messages $m \in \mathcal{M}$,

$$\Pr[\, D\,(sk,\, E\,(pk,\, m)) = m\,] = 1. \tag{3.1}$$

Intuitively, $\mathcal{P}$ satisfies the property of correctness if decryption is the inverse of encryption. However, since K and E are probabilistic algorithms, it makes sense that statements about their outputs will also be probabilistic. To formalize correctness in Lean, we first write a probabilistic program `enc_dec` which simulates running the protocol and returns 1 with probability 1 if Eq. (3.1) holds:

```
def enc_dec  (m : M) : pmf (zmod 2) :=
do
   k ← keygen,
   c ← encrypt k.1 m,
   pure (if decrypt k.2 c = m then 1 else 0)
```

First notice the use of the projections `k.1` and `k.2` in the definition of `enc_dec`, and consider that we could have written `enc_dec` in a way which improves readability:

```
def enc_dec  (m : M) : pmf (zmod 2) :=
do
   (pk, sk) ← keygen,
   c ← encrypt pk m,
   pure (if decrypt sk c = m then 1 else 0)
```

In fact, we initially wrote `enc_dec` in exactly this way, under the assumption that Lean would treat the two identically. However, Lean handles the `(pk, sk)` on the left side of the arrow in a very clunky manner by introducing a lambda function into the goal that is hard to break through with tactics. In a comment on Zulip,[1] a user mentioned encountering similar difficulties, and said that he solved the issue by using projection notation. After spending multiple days on the problem, we were willing to sacrifice readability in the interest of progress.

[1] https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/_match_1_.20term.20in.20monad.20unfolding/near/245857781

Next, recall that since we defined `keygen`, `encrypt`, and `decrypt` as `variables`, and `enc_dec` references these variables, the actual type of `enc_dec`, which can be checked in Lean via `#check enc_dec`, is:

```
pmf (?M1 × ?M2) → (?M1 → ?M3 → pmf ?M4)
                → (?M2 → ?M4 → ?M3) → ?M3 → pmf (zmod 2)
```

The appearance of `?M1`, `?M2`, `?M3`, and `?M4` where we might expect `G1`, `G2`, `M`, and `C`, respectively, is due to the fact that we declared the latter types implicitly via curly braces. In other words, the placeholders `?M1`–`?M4` will be filled in upon input of `keygen`, `encrypt`, and `decrypt`.

Finally, we specify correctness:

```
def pke_correctness : Prop :=
  ∀ (m : M), enc_dec keygen encrypt decrypt m = pure 1
```

In words, for a public key encryption protocol `P` specified by instantiations of `keygen`, `encrypt`, and `decrypt`, we say `P` is semantically secure if running the `enc_dec` protocol with any message `m` results in the distribution which assigns probability 1 to `(1 : zmod 2)`. Since `enc_dec` is constructed to return 1 with probability 1 if Eq. (3.1) holds, our formalization matches the definition. We will see similarly long chains of inputs to construct the definitions of `pke_semantic_security` in Sec. 3.2.2 and `DDH` in Sec. 4.3.

3.2.2 Semantic security

When Alice uses $\mathcal{P}$ to send her private message to Bob, she is not only relying on the fact that $\mathcal{P}$ is correct, but also on the fact that it is hard for an attacker to recover the original message m from a ciphertext c. Consider a game between a challenger C and an attacker A defined as follows:

1. C runs K to obtain (pk, sk) and passes pk to A.
2. A runs any efficient algorithm and outputs two messages $m_0, m_1 \in \mathcal{M}$.
3. C samples $b \leftarrow \mathbb{Z}_2$ uniformly at random, and passes $c := E(pk, m_b)$ to A.
4. A runs any efficient algorithm and outputs a bit b'.

The bit b' corresponds to a guess that the challenger encrypted the message $m_{b'}$. We call this game the **semantic security game**, and say A wins the game if $b' = b$, or equivalently if $1 + b + b' = 1 \in \mathbb{Z}_2$. Intuitively, $\mathcal{P}$ is semantically secure if A can do no better than random guessing. More precisely, denote by $\mathrm{SSG}(A)$ the event that A wins the semantic security game. We say that a public key encryption protocol $\mathcal{P}$ is **semantically secure** [BS15, Def. 11.2] if for any efficient attacker A,

$$\mathrm{SSG\_Adv}(A) := |\Pr[\mathrm{SSG}(A)] - 1/2| \quad \text{is negligible.} \tag{3.2}$$

In Sec. 3.4, we precisely define the term "negligible" in terms of a security parameter $\eta \in \mathbb{Z}_{>0}$. However, at the end of this section we discuss how formalizing negligibility is unnecessary for the purposes of the ElGamal security proof.

In Lean, we can model the probabilistic attacker A as a `pmf`. However, it is unclear over what type the `pmf` should be defined, since A returns a value in $M \times M$ in step 2 of the semantic security game, and a value in $\mathbb{Z}_2$ in step 4. Furthermore, the information which A has access to in step 2 is the public key pk that it received in step 1, whereas the A in step 4 has access to pk from step 1, its output (m_0, m_1) from step 2 (as well as any intermediate calculations conducted in the execution of step 2), and the ciphertext c from step 3. Since a function must have a precisely specified domain and range in a theorem prover, we cannot model A with a single `pmf` in Lean. Instead, we mirror the construction from FCF [Pet15, p. 58] by modeling the single attacker A as the composition of two probabilistic programs `A1` and `A2`:

```
variables (A_state : Type)
          (A1 : G1 → pmf (M × M × A_state))
          (A2 : C → A_state → pmf (zmod 2))
```

Here `A1` corresponds to A in step 2 and `A2` corresponds to A in step 4. We declare `A_state` to be the type of data which `A1` passes to `A2`. In this way, we model the idea that `A1` and `A2` are really the same entity. And since `A1` and `A2` are arbitrary variables, our model encompasses an arbitrary adversary A. In Nowak's toolbox, the adversary A is similarly modeled as two probabilistic programs A_1 and A_2 [Now07, p. 320]. However, A_1 does not pass any sort of state to A_2, so it is unclear how his specification models the fact that A_1 and A_2 are same entity in the semantic security game.

Using the work of Sec. 3.1, we formalize the semantic security game as the probabilistic program `SSG`, again using projections on the left of the arrows as discussed in Sec. 3.2.1:

```
def SSG : pmf (zmod 2) :=
do
   k ← keygen,
   m ← A1 k.1,
   b ← uniform_2,
   c ← encrypt k.1 (if b = 0 then m.1 else m.2.1),
   b' ← A2 c m.2.2,
   pure (1 + b + b')
```

Then the probability that A wins the semantic security game, i.e. $\Pr[\text{SSG}(A)]$ is the probability of `(1 : zmod 2)` in the distribution induced by `SSG`. Hence, we can formalize the semantic security as given in equation (3.2):

```
local notation `Pr[SSG(A)]` := (SSG keygen encrypt A1 A2 1 : ℝ)
def pke_semantic_security (ε : nnreal) : Prop :=
   abs (Pr[SSG(A)] - 1/2) ≤ ε
```

Notice that we don't pass `decrypt` to `SSG` above. Since `SSG` does not reference `decrypt`, it is not added as a parameter.

Ultimately, we prove semantic security for ElGamal in Sec. 4.4 by proving for any efficient attacker A, the value $\text{SSG\_Adv}(A)$ from Eq. 3.2 is equal to a value which is negligible by hypothesis. Hence our definition of semantic security does not need a formal definition of the term negligible; an arbitrary ε suffices as a placeholder. This is also the approach in FCF [Pet15, Sec. 5.1.5].

3.3 Proving equivalence between `pmfs`

In the game hopping proof style described by Shoup [Sho04], the critical step is proving that a certain probability in a game is negligibly close to a certain probability in the next game in the sequence. For our later game-based proof of semantic security for ElGamal in Sec. 4.4, we actually show that each game in the sequence induces the same distribution over $\mathbb{Z}_2$. Since we model distributions as `pmfs` in cryptolib, it follows that we need ways to prove that two `pmfs` are equal.

Let α, β, and γ denote types in Lean. In mathlib, equality between two `pmfs` is

defined by functional extensionality:

```
lemma ext : ∀ {p q : pmf α}, (∀ a, p a = q a) → p = q
```

In words, the distributions `p` and `q` are equal if they assign the same probability to each element of α. The three monad laws (as defined in mathlib) which the functions `pure` and `bind` from Sec. 3.1 satisfy give us one way to prove that two `pmf`s are equal. The first is `pure_bind`:

```
@[simp] lemma pure_bind (a : α) (f : α → pmf β) :
   (pure α).bind f = f a
```

which says that sampling from the distribution `pure a` and passing the sampled value to `f` is the same as passing `a` to `f` deterministically. The `@[simp]` decorator on `pure_bind` means that any invocation of the `simp` tactic will automatically try to rewrite the goal using `pure_bind`. The other two monad laws also include the `@[simp]` decorator. The next monad law is `bind_pure`:

```
@[simp] lemma bind_pure (p : pmf α) : p.bind pure = p
```

which says that the distribution obtained from sampling from `p` and returning the sampled value is equal to `p`. Last is `bind_bind`:

```
@[simp] lemma bind_bind
   (p : pmf α) (f : α → pmf β) (g : β → pmf γ) :
   (p.bind f).bind g = p.bind (λ a, (f a).bind g)
```

which is a form of associativity for the `bind` operation on `pmf`s. Besides the monad laws, mathlib has another useful lemma for proving equivalence between `pmf`s:

```
lemma bind_comm (p : pmf α) (q : pmf β) (f : α → β → pmf γ) :
  p.bind (λ a, q.bind (f a)) = q.bind (λ b, p.bind (λ a, f a b))
```

In words, `bind_comm` says that if `p` and `q` are independent of each other, we can sample `a` from `p` and `b` from `q` in either order to obtain the same distribution `f a b` (over γ).

Apart from the above four lemmas in mathlib, we define the `bind_skip` and `bind_skip_const` tactics in the file `tactics.lean` to simplify proofs of equivalence between `pmf`s. The `bind_skip` tactic is similar to the `comp_skip` tactic from FCF and is based on the following lemma:

```
lemma bind_skip' (p : pmf α)(f g : α → pmf β) :
   (∀ (a : α), f a = g a) → p.bind f = p.bind g
```

The lemma can be applied when two `pmf`s (each defined as a sequence of samples via the `do` syntax) sample from the same distribution on their first step. In that setting, `bind_skip'` states that it suffices to prove that the two distributions agree on all elements of the distribution `p` which they have in common.

In Lean, we can use the `apply` tactic in conjunction with a lemma (or theorem) of type `P → Q` to change a goal `Q` to a goal `P`. The `apply` tactic is the analogue of the mathematical argument structure "By Lemma X, it suffices to show ...." We use `bind_skip` in tactic mode by writing `bind_skip with x`, where `x` can be any identifier the user wants. This invokes the sequence of tactics: (1) `apply bind_skip'`, and (2) `intro x`. The `intro` tactic handles the universal quantifier ∀ in the new goal (after `apply`) by introducing the hypothesis `(x : α)`.

Informally, `bind_skip` lets us peel back the layers of two `pmf`s defined as sequences of monadic binds which sample from the same distribution on their first step. To illustrate the use of the `bind_skip` tactic, consider the distributions `p1` and `p2`:

```
def p1 : pmf (zmod 5) :=
do
   x ← uniform_zmod 5,
   y ← uniform_zmod 5,
   pure (x + y)
```

```
def p2 : pmf (zmod 5) :=
do
   x ← uniform_zmod 5,
   y ← uniform_zmod 5,
   pure (y + x)
```

It is clear mathematically that `p1` and `p2` define the same distribution because $x+y = y+x$ in $\mathbb{Z}_5$. However, we have to peel off the monadic binds before we can use the lemma `add_comm` which says $x+y = y+x$ in $\mathbb{Z}_5$; the `bind_skip` tactic makes this possible:

```
example : p1 = p2 :=
begin
   simp [p1, p2], bind_skip with x, bind_skip with y,
   simp_rw add_comm,
end
```

The other tactic we provide in cryptolib, `bind_skip_const` , is used in a manner similar to `bind_skip` and is based on the following lemma:

```
lemma bind_skip_const' (pa : pmf α)(pb : pmf β)(f : α → pmf β):
   (∀ (a : α), f a = pb) → pa.bind f = pb
```

Intuitively, the `bind_skip_const` tactic lets us skip superfluous binds. We show its use by way of example. Consider the distribution `p3`:

```
def p3 : pmf (zmod 5) :=
do
   x ← uniform_zmod 5,
   y ← uniform_zmod 5,
   pure y
```

It should be obvious that the actual value of x after the sampling has no impact on the resulting distribution, and so `p3` is equal to `uniform_zmod 5` by the `bind_pure` monad law (applied automatically below by using `simp`):

```
example : p3 = uniform_zmod 5 :=
begin
   bind_skip_const with x,
   simp [bind, pure],
end
```

Though this example is rather trivial since the rest of `p3` never references the bound variable x again, it is a common scenario that a distribution is unaffected by a sampled value even when that value is later referenced. Therein lies the purpose of the `bind_skip_const` tactic: if a sampling (in the form of a monadic bind) has no impact

on the resulting distribution, then we can safely ignore it. The `bind_skip_const` tactic greatly simplifies the proof of correctness for ElGamal in Sec. 4.2 and one of the key lemmas in the proof of semantic security in Sec. 4.4. The `bind_skip` tactic also proves critical within later proofs.

3.4 Negligibility

In cryptolib, we prove semantic security for ElGamal in Sec. 4.4 by proving for any efficient attacker A, the value SSG_Adv(A) from Eq. 3.2 is equal to a value which is negligible by hypothesis. Hence, we are able to show that SSG_Adv(A) is negligible without ever needing a formalization of the term negligible. This is exactly the proof strategy used in FCF [Pet15, Sec. 5.1.5], and it means we avoid having to specify the dense mathematical foundation of the computational model, which we briefly discuss.

Recall that we mentioned in Sec. 2.2 that a rigorous definition of efficient involves a security parameter $\eta \in \mathbb{Z}_{>0}$. Since we assume A is efficient, it turns out that the "value" SSG_Adv(A) is actually a function of η [BS15, Sec. 2.3.4]. It follows that negligibility is a property of a function: we say a function $f : \mathbb{Z}_{\geq 0} \to \mathbb{R}$ is **negligible** [BS15, Def. 2.5] if for all $c > 0$,

$$\lim_{\eta \to \infty} f(\eta) \cdot \eta^c = 0. \tag{3.3}$$

Though the slick proof from FCF hides the security parameter under the rug, we provide the file `negligible.lean` in cryptolib in the event that someone (else) wants to completely formalize the computational model in Lean in the future.

We first formalize the definition of negligible from Eq. (3.3):

```
def negligible (f : ℕ → ℝ) :=
  ∀ c > 0, ∃ n₀, ∀ n,
  n₀ ≤ n → abs (f n) < 1 / (n : ℝ)^c
```

Note that we use $\mathbb{N}$ as the domain for `f` since $\mathbb{N} = \mathbb{Z}_{\geq 0}$ in Lean, i.e. `0` $\in \mathbb{N}$. We also provide several lemmas, such as the sum of two negligible functions is negligible:

```
lemma negl_add_negl_negl {f g : ℕ → ℝ} :
  negligible f → negligible g → negligible (f + g)
```

We use `negl_add_negl_negl` to prove by induction that for any $m \in \mathbb{N}$, f negligible implies $m \cdot f$ is negligible:

```
lemma nat_mul_negl_negl {f : ℕ → ℝ} (m : ℕ):
negligible f → negligible (λ n, m * (f n))
```

We also prove that any function bounded by a negligible function is negligible:

```
lemma bounded_negl_negl {f g : ℕ → ℝ} (hg : negligible g):
   (∀ n, abs (f n) ≤ abs (g n)) → negligible f
```

Finally, we use `nat_mul_negl_negl` and `bounded_negl_negl` to prove that for $c \in \mathbb{R}$, f negligible implies $c \cdot f$ is also negligible:

```
lemma const_mul_negl_negl {f : ℕ → ℝ} (m : ℝ) :
   negligible f → negligible (λ n, m * (f n))
```

We also state, but do not prove, that $f : \mathbb{N} \to \mathbb{R}$ given by $f(n) = 2^{-n}$ is negligible:

```
theorem neg_exp_negl :
   negligible ((λ n, (1 : ℝ) / 2^n) : ℕ → ℝ) := by sorry
```

A mathematical proof of this fact proceeds by induction on c (from the definition of negligible) and uses L'Hôpital's rule to apply the inductive hypothesis. Mathlib has a formalization of L'Hôpital's rule,[2] but we were unable to parse the definitions to the point that we could apply them to our specific case in time for inclusion into the submitted version of cryptolib. A concerted effort over the course of a day or two (or delegating the task via Zulip) would suffice to close this goal. The above lemmas can be combined to prove that any function in $O(2^{-\eta})$ is also negligible, which is a useful fact for asymptotic proofs in the computational model, since $2^{-\eta}$ often appears as a bound.

[2] `https://leanprover-community.github.io/mathlib_docs/analysis/calculus/lhopital.html`

Chapter 4

ElGamal

In this chapter, we describe our formalization of the ElGamal public key encryption protocol and the proofs of correctness and semantic security. In Sec. 4.1, we define the ElGamal protocol using a finite cyclic group G, and show that the protocol satisfies correctness. In Sec. 4.2, we formalize the proof of correctness (with respect to the definition given in Sec. 3.2.1) leveraging the `bind_skip_const` tactic to prove an equivalence between `pmf`s. We also compare the formal proof of correctness with the computational proof of correctness given in Sec. 4.1. In Sec. 4.3 we formalize the DDH assumption before finally proving in Sec. 4.4 that ElGamal is semantically secure when the decisional Diffie-Hellman assumption holds for the group G on which the protocol is based. For the remainder of this chapter, we fix the following notation:

$$
\begin{aligned}
G &:= \text{a finite cyclic group,} \\
g &:= \text{a generator of } G, \text{ that is } G = \langle g \rangle, \\
q &:= \text{the order of } G, \text{ or equivalently the order of } g.
\end{aligned}
$$

These hypotheses are formalized at the top of `elgamal.lean` as follows:

```
parameters (G : Type) [fintype G][comm_group G][decidable_eq G]
           (g : G)(g_gen_G : ∀ (x : G), x ∈ subgroup.gpowers g)
           (q : ℕ)[fact (0 < q)](G_card_q : fintype.card G = q)
include g_gen_G G_card_q
```

The `parameters` keyword has a different purpose than `variables`. If we declared `variables (G : Type)`, then each subsequent *definition* in `elgamal.lean` which referenced `G` would automatically be given an extra parameter to supply a `Type`. On the other hand, when we use `parameters (G : Type)`, any *proof context* of a theorem

(or lemma) which references `G` gains the hypothesis that `(G : Type)`. In essence, we are declaring some fixed `G` for the file `elgamal.lean`, so that any reference to `G` is to the same `G`. We use this method because `elgamal.lean` is a "concrete case" file, in the sense that its contents will be not be used by other files; we use the other files to prove the things in `elgamal.lean`. Similarly, the `include` command automatically adds the hypotheses `g_gen_G` and `G_card_q` to every proof context. Since `g_gen_G` and `G_card_q` are not referenced explicitly in theorem statements, `parameters` is not enough to ensure they are in the context.

4.1 The protocol

Taher El Gamal defined the eponymous public key encryption protocol in [Gam85]. In what follows, we only modify the notation used in the original description of the protocol. The key algorithm samples $x \in \mathbb{Z}_q$ uniformly at random. Then the public key is $pk := (G, g, q, g^x)$ and the private key is $sk := (G, g, q, x)$. We formalize this probabilistic algorithm as `keygen`.

```
def keygen : pmf (G × (zmod q)) :=
do
   x ← uniform_zmod q,
   pure (g^x.val, x)
```

Here `x.val` is the minimal element of $\mathbb{N}$ in the equivalence class $x \in \mathbb{Z}_q$. Note that declaring `G`, `g`, and `q` (along with their respective hypotheses) as `parameters` at the top of the files ensures that they are made implicit inputs for all other functions. In other words, it suffices to have `keygen` return `(g^x.val, x)` as the public and private key pair, instead of the more cumbersome `((G, g, q, g^x), (G, g, q, x))`.

The message space for ElGamal is G and the ciphertext space is $G \times G$. One might imagine in practice that if we wanted to communicate a sequence of English letters, we could use ASCII encoding to get a sequence of bytes, which in turns corresponds to some non-negative integer n. If we choose G such that the order q is greater than n, then g^n uniquely encodes our ASCII message. For a message $m \in G$, the encryption algorithm proceeds by sampling $y \in \mathbb{Z}_q$ uniformly at random and returning as the ciphertext

$$c := (g^y, (g^x)^y \cdot m) \in G \times G$$

This probabilistic algorithm is encoded as `encrypt` below.

```
def encrypt (pk m : G) : pmf (G × G) :=
do
   y ← uniform_zmod q,
   pure (g^y.val, pk^y.val * m)
```

Finally, the deterministic decryption algorithm takes as input the private key $x \in \mathbb{Z}_q$ and a ciphertext $(c.1,\ c.2) \in G \times G$ and returns $c.2/c.1^x \in G$.

```
def decrypt (x : zmod q) (c : G × G) : G :=
   (c.2 / (c.1^x.val))
```

The correctness of ElGamal is easy to verify by hand from this description. For $(g^x,\ x)$ returned from K and a message $m \in G$, we have

$$D\ (x,\ E\ (g^x,\ m)) = \frac{(g^x)^y \cdot m}{(g^y)^x} = \frac{g^{xy} \cdot m}{g^{yx}} = m. \tag{4.1}$$

However, formalizing this seemingly straightforward calculation in Lean is a little more involved and is the subject of Sec. 4.2.

4.2 Proof of correctness

The mathematical proof given in Eq. (4.1) in Sec. 4.1 consisted of unpacking the definition of E and D and then simplifying the resulting expression based on algebraic identities in the group G. The statement of the corresponding lemma for fixed x, y, and m in Lean is given below:

```
lemma decrypt_eq_m (m : G) (x y: zmod q) :
   decrypt x ((g^y.val), ((g^x.val)^y.val * m)) = m
```

However, this lemma does not immediately prove correctness in Lean, which we defined as `pke_correctness` in Sec. 3.2.1 as an equivalence between `pmf`s:

```
def pke_correctness : Prop :=
   ∀ (m : M), enc_dec keygen encrypt decrypt m = pure 1
```

```
theorem elgamal_correctness :
   pke_correctness keygen encrypt decrypt :=
begin
   simp [pke_correctness],
   intro m,
   simp [enc_dec, keygen, encrypt, bind],
   bind_skip_const with x,
   simp [pure],
   bind_skip_const with y,
   simp_rw decrypt_eq_m,
   simp,
end
```

Figure 4.1: Tactic mode proof of correctness for ElGamal.

Recall that `enc_dec` takes as input the `keygen`, `encrypt`, and `decrypt` functions which define a public key encryption protocol. Then, for a given message `m`, `enc_dec` runs the protocol and returns 1 if the decrypted message equals `m`.

Intuitively, the lemma `decrypt_eq_m` doesn't suffice on its own because assuming a fixed `x` implies we have implicitly "run" `keygen` (or sampled a key pair from its distribution, in the sense of a monadic bind). Similarly, assuming a fixed `y` implies we have "run" `encrypt`. But the key insight is that the actual values of `x` and `y` don't affect the resulting distribution `enc_dec`, precisely because of `lemma decrypt_eq_m`. Recall from Sec. 3.3 that the `bind_skip_const` tactic is written to simplify `pmf` proofs in situations where the values from a sampling don't affect the resulting distribution. (Our initial proof[1] without `bind_skip_const` was 139 lines of code versus 14 lines of code with `bind_skip_const` .)

After unpacking the various function definitions and the universal quantifier in `pke_correctness` with `simp` and `intro` respectively, we use

```
bind_skip_const with x
```

to peel off from the goal the outermost (superfluous) bind, which corresponds to `x ← uniform_zmod q` in the ElGamal `keygen` algorithm. At the same time, the tactic

[1] `https://github.com/Loops7/cryptolib/blob/0d710437f1129a296cc0756156f4b711eb80ad9d/src/elgamal.lean`, viewer discretion is advised.

introduces `(x : zmod q)` into the proof context. A similar use of `bind_skip_const` a couple lines later peels off the monadic bind corresponding to `y ← uniform_zmod q` from `encrypt`. The goal is then essentially reduced to `decrypt_eq_m` plus a trivial final deduction which `simp` solves.

Though the mathematical proof of correctness is conceptually very simple, our formalization of the proof involved iteratively unraveling a sequence of monadic binds via a custom-made tactic. This section illustrates some of the unique challenges of the formalization process.

4.3 The DDH assumption

Let $x, y, z \in \mathbb{Z}_q$ be sampled independently and uniformly at random. Informally, the decisional Diffie-Hellman (DDH) assumption on G states that an attacker cannot distinguish the two tuples

$$(g^x,\, g^y,\, g^{xy}), \quad \text{and} \quad (g^x,\, g^y,\, g^z)$$

with non-negligible probability. We make this notion of indistinguishability precise with an equivalent construction. Consider a probabilistic distinguisher $D : G^3 \to \mathbb{Z}_2$. Let $\mathrm{DDH0}(D)$ denote the event that D outputs 1 when it receives $(g^x,\, g^y,\, g^{xy})$ and let $\mathrm{DDH1}(D)$ denote the event that D outputs 1 when it receives $(g^x,\, g^y,\, g^z)$. Then we say that the **DDH assumption** holds for G if

$$|\Pr[\mathrm{DDH0}(D)] - \Pr[\mathrm{DDH1}(D)]| \quad \text{is negligible.} \tag{4.2}$$

All formalizations in this section are in `ddh.lean`. First, we model the distinguisher D as a `pmf`:

```
variables (D : G → G → G → pmf (zmod 2))
```

We model $\mathrm{DDH0}(D)$ as the event that `DDH0 = 1` for `DDH0` below:

```
def DDH0: pmf (zmod 2) :=
do
   x ← uniform_zmod q,
   y ← uniform_zmod q,
   b ← D (gˆx.val) (gˆy.val) (gˆ(x.val * y.val)),
```

```
  pure b
```

Similarly, DDH1(D) is the event that `DDH1 = 1`:

```
def DDH1: pmf (zmod 2) :=
do
  x ← uniform_zmod q,
  y ← uniform_zmod q,
  z ← uniform_zmod q,
  b ← D (gˆx.val) (gˆy.val) (gˆz.val),
  pure b
```

Finally we formalize the idea that these two probabilities are negligibly close (in terms of arbitrary ε, as we described in Sec. 3.2.2):

```
local notation `Pr[DDH0(D)]` :=
  (DDH0 G g g_gen_G q G_card_q D 1 : ℝ)
local notation `Pr[DDH1(D)]` :=
  (DDH1 G g g_gen_G q G_card_q D 1 : ℝ)
def DDH (ε : nnreal) : Prop :=
  abs (Pr[DDH0(D)] - Pr[DDH1(D)]) ≤ ε
```

The long list of terms passed to `DDH0` and `DDH1` correspond to `variables` declared at the top of `ddh.lean`, which we have omitted in our exposition. Their definitions correspond with the `parameters` declared on p. 25. The input 1 at the end of the long list of inputs to `DDH0` (resp. `DDH1`) corresponds to the event `DDH0 = 1` (resp. `DDH1 = 1`), which is what we need to model the DDH assumption as in Eq. (4.2).

4.4 Proof of semantic security

The formalized proof of semantic security for ElGamal is much more involved than the proof of correctness and spans several hundred lines of code. Thus, it is impossible to go into the level of detail we did with the proof of correctness, but we give an outline of the proof. Let A denote the attacker in the semantic security game. As in Sec. 3.2.2, let SSG(A) denote the event that A wins the semantic security game, and let SSG_Adv(A)

be the value $|\Pr[\mathrm{SSG}(A)] - 1/2|$. Let D be a probabilistic distinguisher which takes as input three elements of G and outputs an element of $\mathbb{Z}_2$. The distinguisher D uses A as a subroutine and will be formally specified below. Let $\mathrm{DDH0}(D)$ and $\mathrm{DDH1}(D)$ be defined as in Sec. 4.3. The proof of semantic security is as follows:

$$\begin{aligned}
\mathrm{SSG_Adv}(A) := |\Pr[\mathrm{SSG}(A)] - 1/2| &= |\Pr[\mathrm{DDH0}(D)] - 1/2| && (4.3)\\
&= |\Pr[\mathrm{DDH0}(D)] - \Pr[\mathrm{Game2}(A) = 1]| && (4.4)\\
&= |\Pr[\mathrm{DDH0}(D)] - \Pr[\mathrm{Game1}(A) = 1]| && (4.5)\\
&= |\Pr[\mathrm{DDH0}(D)] - \Pr[\mathrm{DDH1}(D)]| && (4.6)
\end{aligned}$$

where Game1 and Game2 are intermediate games which we will also define below. If the DDH assumption holds for the group G which underlies ElGamal, then by definition the value $|\Pr[\mathrm{DDH0}(A)] - \Pr[\mathrm{DDH1}(A)]|$ is negligible if D is an efficient algorithm. It follows that $\mathrm{SSG\_Adv}(A)$ is negligible, and so semantic security is proven.

As in Sec. 3.2.2, we model the attacker A as the composition of `A1` and `A2`:

```
(A1 : G → pmf (G × G × A_state))
(A2 : G → G → A_state → pmf (zmod 2))
```

Then the probabilistic distinguisher D is formalized as `D` below:

```
def D (gx gy gz : G) : pmf (zmod 2) :=
do
   m  ← A1 gx,
   b  ← uniform_2,
   mb  ← pure (if b = 0 then m.1 else m.2.1),
   b'  ← A2 gy (gz * mb) m.2.2,
   pure (1 + b + b')
```

`D` uses `A1` and `A2` as subroutines. We follow FCF [Pet15, Sec. 5.1.5] and do not formally prove that `D` is efficient assuming `A1` and `A2` are efficient, but this fact is obvious since `D` consists of a finite sequence of efficient subroutines.

The formal proof of semantic security consists of proving the four equalities (4.3)–(4.6). Eq. (4.3) follows from $\mathrm{SSG}(A) = \mathrm{DDH0}(D)$, which we prove by showing the distributions `SSG(A)` and `DDH0(D)` are equal:

```
theorem SSG_DDH0 :
   SSG keygen encrypt A1 A2' =  DDH0 G g g_gen_G q G_card_q D
```

Next, Eq. (4.4) follows from the fact that $1/2 = \Pr[\text{Game2}(A) = 1]$, where Game2 is defined in Lean as the probabilistic program `Game2` below:

```
def Game2 : pmf (zmod 2) :=
do
   x ← uniform_zmod q,
   y ← uniform_zmod q,
   m ← A1 (g^x.val),
   b ← uniform_2,
   ζ ← (do z ← uniform_zmod q,
         pure (g^z.val)),
   b' ← A2 (g^y.val) ζ m.2.2,
   pure (1 + b + b')
```

Notice in `Game2` that since the value of `b` is random, the value of `b'` does not affect the resulting distribution of `1 + b + b'`. Thus, by repeatedly applying the `bind_skip_const` tactic, we can easily prove the following theorem:

```
theorem Game2_uniform : Game2 = uniform_2
```

from which it follows that $\Pr[\text{Game2}(A) = 1] = 1/2$ by `lemma` `uniform_zmod_prob_2` from Sec. 3.1. Finally, we specify Game1 as `Game1` below:

```
def Game1 : pmf (zmod 2) :=
do
   x ← uniform_zmod q,
   y ← uniform_zmod q,
   m ← A1 (g^x.val),
   b ← uniform_2,
   ζ ← (do z ← uniform_zmod q,
         mb ← pure (if b = 0 then m.1 else m.2.1),
         pure (g^z.val * mb)),
```

```
  b' ← A2 (g^y.val) ζ m.2.2,
  pure (1 + b + b')
```

Eq. (4.5) follows from the fact that $\Pr[\text{Game1}(A) = 1] = \Pr[\text{Game2}(A) = 1]$, which is formalized by showing that `Game1` and `Game2` induce the same distribution:

```
theorem Game1_Game2 : Game1 = Game2
```

Notice `Game1` and `Game2` are exactly the same until the fifth line. Thus, repeated use of the `bind_skip` tactic simplifies the proof. However, the meat of the proof is a tedious verification that the distribution from which ζ samples in `Game1` is the same as the distribution from which ζ samples in `Game2`, using the base definition of equality of `pmf`s via functional extensionality from Sec. 3.3. That proof involves five lemmas and almost 150 lines of Lean code in `elgamal.lean`. A key lemma is `exp_bij`:

```
lemma exp_bij : function.bijective (λ (z : zmod q), g ^ z.val)
```

which formalizes the bijection $\mathbb{Z}_q \to G = \langle g \rangle$ given by $z \mapsto g^z$. We assume this fact has been proven somewhere in mathlib, but most likely at a very high level of generality, so we decided to just go ahead and prove it manually. The last equality in Eq. (4.6) follows from the fact that $\Pr[\text{Game1}(A) = 1] = \Pr[\text{DDH1}(D)]$, which is proven in `Game1_DDH1`:

```
theorem Game1_DDH1 : Game1 = DDH1 G g g_gen_G q G_card_q D
```

Unfolding the definition of `D` within `DDH1`, it is apparent by inspection of the goal state in Lean that the distributions are almost identical modulo applying `bind_bind`, `bind_comm`, and the `bind_skip` tactic. Finally, we state semantic security for ElGamal:

```
parameter (ε : ℝ)
theorem elgamal_semantic_security
   (DDH_G : DDH G g g_gen_G q G_card_q D ε) :
   pke_semantic_security keygen encrypt A1 A2'
```

where the parameter `(DDH_G : DDH G g g_gen_G q G_card_q D ε)` uses the definition of `DDH` from `ddh.lean` to formalize the assumption that G satisfies the deci-

sional Diffie-Hellman assumption. The proof is a straightforward sequence of rewrites according to Eqs. (4.3)–(4.6), as formalized in the four theorems above.

Chapter 5

Evaluation and Comparison

At a high level, the capabilities of the cryptolib library are comparable to Nowak's toolbox [Now07, Now08], but fall short of state of the art frameworks like EasyCrypt [BGHB11], CryptHOL [BLS20, Loc17], and the Foundational Cryptographic Framework (FCF) [PM15, Pet15]. However, these projects represent the work of several people over the course of several years. Due to the time constraints of this project, we couldn't hope to produce a framework with all of the features for automation or verified code extraction, etc., that these other projects provide. Instead, we chose to focus our efforts on formalizing the game-based proofs of correctness and semantic security for ElGamal encryption. The security proof for ElGamal is conceptually simple relative to other cryptographic proofs, but still yields a complete proof of concept that the underlying modeling of games in a theorem prover is valid. Each of EasyCrypt, CryptHOL, Nowak's toolbox, and FCF formalizes a similar proof for ElGamal as a proof of concept, making it easy to compare our work and the relative benefits of the underlying theorem prover, which are as follows: EasyCrypt is a deep embedding in Coq; CryptHOL is a shallow embedding in Isabelle/HOL; Nowak's toolbox is a shallow embedding in Coq; and FCF is a shallow embedding in Coq. In their paper presenting FCF, Petcher and Morrisett enumerate criteria which a framework for security proofs should satisfy [PM15, pp. 54–55]. In what follows, we use their criteria as a framework to guide our evaluation of cryptolib and Lean. All quoted material in (1)–(6) is from [PM15, pp. 54–55].

(1) "***Familiarity***. Security definitions and descriptions of cryptographic schemes should look similar to how they would appear in cryptography literature, and a cryptographer with no knowledge of programming language theory or proof assistants should be able to understand them." The `pmf` + `do` syntax used in cryptolib to model games is

an expressive construct that results in formalizations which closely match their mathematical counterparts. This is particularly clear in the case of the semantic security game `SSG`, and the decisional Diffie-Hellman "games" `DDH0` and `DDH1`. Furthermore, Lean's unicode support reduces the amount of code (e.g. compare ∀ and ∃ in Lean to writing `forall` and `exists` in Coq) and improves readability.

We predict that there are two primary barriers to comprehension of cryptolib definitions for someone not familiar with a theorem prover. The first is not understanding the effect of the `variables` keyword, which results in terms like

```
(DDH0 G g g_gen_G q G_card_q D : pmf (zmod 2))
```

from Sec. 4.3. However, this phenomenon is unavoidable in Lean, since `DDH0` has to know the assumptions on the group `G` if we are to claim that our specification accurately models the DDH assumption. There is no way to get around passing in these assumptions at some point in the code. The story is no different for Coq; Nowak's toolbox and FCF include similarly verbose input lists in their definitions of the DDH assumption and semantic security. In contrast, CryptHOL uses Isabelle's module system by declaring a `locale` to avoid having to pass in the long list of assumptions as explicit parameters in the definitions of DDH and semantic security [LSB17, p. 3, 13].

The second barrier is the use of the projection notation, as in

```
k ← keygen, c ← encrypt k.1 m
```

from the definition of `enc_dec` in Sec. 3.2.1. Though understanding the types of the sampled values makes the projection notation easy to parse, we suspect there are ways to improve the readability further. One way would be to include the samples `pk ← pure k.1`, and `sk ← pure k.2` in the specifications, though these inclusions negatively affect readability by increasing the length of the definition. Ultimately, we chose to use the projection values directly in the interest of concision. Nowak uses projections `Datatypes.fst` in his toolbox, whereas FCF uses the nicer pattern matching construct in game descriptions.

(2) "***Proof automation***. The system should use automation to reduce the effort required to develop a proof." The proofs in cryptolib for the most part rely on the four lemmas `pure_bind`, `bind_pure`, `bind_bind`, and `bind_comm`, as well as the custom tactics `bind_skip` and `bind_skip_const` to prove equivalences between `pmf`s. The first three lemmas (the monad laws) are marked with an `@[simp]` decorator, and so are automatically applied when we use `simp` in tactic mode. However, we still

have to apply `simp` and the `bind_skip` and `bind_skip_const` tactics manually to interactively solve the goal. Nowak provides tactics which automatically apply the monad laws, as well as a directed form of `bind_comm`. He also gives an analogue of the `bind_skip_const` tactic, which is especially useful in the proof of correctness [Now07, p. 332]. FCF includes similar tactics, as well as the `comp_skip` tactic which inspired the `bind_skip` tactic in cryptolib. Thus, the tactics in cryptolib, Nowak's toolbox, and FCF provide roughly the same amount of automation. CryptHOL supports a declarative proof style wherein proof steps are stated by the user and it tries to automatically fill in the details [BLS20, p. 496]. However, none of the above can approach the level of automation offered by EasyCrypt, which makes extensive use of satisfiability modulo theories (SMT) solvers to automatically dispense goals without user input [BGHB11, p. 73].

(3) "***Trustworthiness***. Proofs should be checked by a trustworthy procedure, and the core definitions (e.g. programming language semantics) that must be inspected in order to trust a proof should be relatively simple and easy to understand." The choice to use a shallow embedding of games means that cryptolib needs only the axioms on which Lean relies. Further, we involve no outside automatic provers (e.g. SMT solvers) as in EasyCrypt which might expand the base of trust. Thus cryptolib is at least as trustworthy as Lean itself. FCF and Nowak's toolbox provide a similar level of trust with respect to Coq, as does CryptHOL with respect to Isabelle/HOL.

(4) "***Extensibility***. It should be possible to directly incorporate any existing theory that has been developed for the proof assistant." The choice of a shallow embedding means that cryptolib has full access to the extensive mathlib library of theorems and tactics, a key advantage of Lean. Consider that we were almost able to use the `pmf` definition right out of the box to suit our needs. To use `pmf` in combination with the `do` syntax, we only had to provide a proof in `to_mathlib` that `pmf` is an `instance` of a `monad`, but this amounted to pointing Lean at the already defined `pure` and `bind` functions. We were further able to take advantage of the already defined structure of the `zmod` type, including several key lemmas. Nowak's toolbox and FCF include significant amounts of code to get analogues of the `pmf` and `zmod` definitions. They also define custom notation which mirrors the `do` syntax we get for free.

Similarly, much of the CryptHOL specification consists of defining a subprobability mass function `spmf` and proving literally hundreds of lemmas. CryptHOL uses the `spmf` in a similar way as cryptolib uses `pmf`, which works out of the box from mathlib. Furthermore, `pmf` has a much more understandable specification than CryptHOL. The

full specification for CryptHOL is a nearly impenetrable 316-page document [Loc17].

If we expand the *extensibility* criterion from "existing theory" to "existing ecosystem," then we can mention another important benefit of the cryptolib ecosystem, namely the very active Lean community on the Zulip chat platform.[1]. In the course of completing this project, we asked several questions on Zulip, each of which received a response in the span of ten minutes. In Sec. 3.2.1, we also mentioned how a Zulip user helped us sort out mysterious Lean behavior and continue our development. The real time help line on Zulip is a huge benefit to working in Lean.

(5) "***Concrete Security***. The security proof should provide concrete bounds on the probability that an adversary is able to defeat the scheme." Our proof of semantic security for ElGamal is a concrete result in that we directly equated the advantage SSG_Adv(A) of the attacker in the semantic security game with the "advantage" $|\Pr[\text{DDH0}(D)] - \Pr[\text{DDH1}(D)]|$ of a distinguisher D in the decisional Diffie-Hellman problem. However, cryptolib is limited to proving concrete security results which are equalities, as in the previous sentence. In general, concrete security results bound the advantage of an attacker in relation to a hardness assumption and are not restricted to equalities. We discuss this distinction in more depth below.

Cryptolib is able to model games using the `pmf` + `do` syntax as in Sec. 3.1 and prove game hops where the distribution induced by G_i is equal to the distribution induced by G_{i+1} using the lemmas and tactics described in Sec. 3.3. This framework is roughly equivalent to the capabilities of Nowak's framework and is sufficient to prove semantic security for ElGamal, as well as security for certain other cryptographic protocols, such as those formalized in [Now08]. However, in the general game-based methodology for security proofs, much less restrictive game hops are permitted. In particular, a game hop is permissible if we can prove

$$|\Pr[G_i(A) = 1] - \Pr[G_{i+1}(A) = 1]| \quad \text{is negligible,} \tag{5.1}$$

[Sho04, p. 2]. FCF is able to formalize such game hops [PM15, Sec. 3.2], as is EasyCrypt [BGHB11, Lem. 1], and CryptHOL [LSB17, Sec. 5.5]. Each of these projects implements the so-called "fundamental lemma of game-playing" in [BR04, Lem. 2] which bounds the value of the term in Eq. (5.1) under certain condition. The result is that these frameworks are able to prove concrete security for a significantly larger class of protocols.

(6) "***Code Generation***. The system should be able to generate code containing

[1] `https://leanprover.zulipchat.com/`

the procedures of the cryptographic scheme that was proven secure." There is often a gap between the formal specification of a protocol in a theorem prover and an actual implementation of the protocol in software and hardware. Verifying protocol implementations is a current direction of security research, see e.g. [APS14] for a survey. *Code generation* is even more ambitious; it refers to the ability to extract a provably secure implementation of a protocol from a provably secure specification. Among EasyCrypt, CryptHOL, FCF, and Nowak's toolbox, FCF is the only project which includes functionality for code extraction [PM15, Sec. 3.6],[Pet15, Ch. 7].

Cryptolib does not have the capability to extract verified implementations, not because we didn't code it, but because our framework is built on classical reasoning. In mathlib, the definition of `pmf` is marked as `noncomputable`, which in Lean means we are operating within classical logic, i.e. we assume the axiom of choice. Any code in cryptolib which uses a `pmf` is also marked `noncomputable`. While we lose the ability to extract code, we simplify many aspects of the resulting code. From the axiom choice we can derive the law of the excluded middle, which in turn we can use to show that every proposition is decidable [AdMK21, Sec. 11.6]. In particular, equality is automatically decidable for any type. Recall in Sec. 3.1 how the type `G` was given the type class `[decidable_eq G]`. The use of a type class requires that we provide a term (or instance proof, more precisely) of type `decidable_eq G` if want to work with terms of type `pmf G`. But since we are working within classical logic via the `noncomputable` keyword, Lean automatically dismisses all requirements to provide proofs of `decidable_eq`. Due to mathlib, the makeup of the Lean community skews towards mathematicians, 99.9 percent of whom conduct their research within classical reasoning. Thus the community tends to produce proofs as a mathematician would, i.e. without a care for constructive logic.

This acceptance of classical reasoning in Lean is in stark contrast to the staunchly constructivist Coq community. This constructivist bent is reflected in both Nowak's toolbox and FCF. In FCF, using the type `Comp A` in FCF (which is analogous to a `pmf A` in cryptolib) requires passing in as a parameter a term of type `eq_dec A`. Nowak's toolbox includes a similar term for decidable equality in its definition of a distribution. For a cryptographer or mathematician looking to use a framework to formalize security proofs, it makes life much easier to not have to think about foundational issues such as decidable equality. Furthermore, the presence of all the decidable equality terms makes the code harder to read.

Chapter 6

Conclusion and Future Work

The use of theorem provers offers one solution to the "crisis of rigor" [BR04, p. 3] in security proofs for cryptographic protocols. Consequently, researchers have spent considerable effort in recent years to develop tools in theorem provers which aid in the formalization of security proofs. In this paper, we contributed to this effort by presenting cryptolib, the first such framework in Lean. The novel use of mathlib's `pmf` monad with Lean's `do` syntax gave us the ability to model games as probability distributions, which in turn allowed us to formalize game-based security proofs. As a proof of concept, we formalized proofs of correctness and semantic security for the ElGamal encryption protocol.

There are a number of ways to expand cryptolib. The easiest is to use the definitions and formalizations already in place. For example, we could formalize several security properties for public key encryption protocols which are defined via games, such as indistinguishability under chosen-ciphertext attack (IND-CCA1), and indistinguishability under adaptive chosen-ciphertext attack (IND-CCA2). We could also define correctness and security for a symmetric encryption scheme. We might also define the random oracle model to facilitate security proofs given in that model, such as the proof of semantic security for the hashed version of ElGamal [Sho04, Sec. 8]. The two cryptographic protocols formalized in [Now08] are also attainable within the cryptolib framework. More ambitiously, we might look to incorporate the fundamental lemma of game-playing from [BR04] to be able to formalize game hops which are not equalities of the underlying distributions.

Bibliography

[AdMK21] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem Proving in Lean, 2021. Available at `https://leanprover.github.io/theorem_proving_in_lean/index.html`. Accessed May 26, 2021.

[ANY12] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of BBS. *Sci. Comput. Program.*, 77(10-11):1058–1074, 2012.

[APS14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects Comput.*, 26(1):99–123, 2014.

[BAG17] David Butler, David Aspinall, and Adrià Gascón. How to simulate it in Isabelle: Towards formal proof for secure multi-party computation. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2017.

[BAG20] David Butler, David Aspinall, and Adrià Gascón. Formalising oblivious transfer in the semi-honest and malicious model in CryptHOL. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 229–243. ACM, 2020.

[BCM20] K Buzzard, J Commelin, and P Massot. Formalising perfectoid spaces. pages 299–312. Association for Computing Machinery, 2020.

[BDG+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In

Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[BGHB11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[BHL+21] Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernndez Mir, and Scott Morrison. Schemes in Lean, 2021. Available at `https://arxiv.org/abs/2101.02602`. Accessed August 7, 2021.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 82–96. IEEE Computer Society, 2001.

[Bla12] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2012.

[BLAG21] David Butler, Andreas Lochbihler, David Aspinall, and Adrià Gascón. Formalising Σ-protocols and commitment schemes using CryptHOL. *J. Autom. Reason.*, 65(4):521–567, 2021.

[BLS20] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2):494–566, 2020.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 62–73. ACM, 1993.

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptol. ePrint Arch.*, 2004:331, 2004.

[BS15] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography, version 0.5*. 2015. Available at `https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_5.pdf`.

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer, 2013.

[CR88] Benny Chor and Ronald L. Rivest. A knapsack-type public key cryptosystem based on arithmetic in finite fields. *IEEE Trans. Inf. Theory*, 34(5):901–909, 1988.

[dt21] The Coq development team. The Coq proof assistant, version 8.13.2, 2021. http://coq.inria.fr.

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

[Gir82] Michèle Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, pages 68–85, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[KMM13] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-aided reasoning: ACL2 case studies*, volume 4. Springer Science & Business Media, 2013.

[Loc17] Andreas Lochbihler. CryptHOL. *Arch. Formal Proofs*, 2017. Available at `https://www.isa-afp.org/entries/CryptHOL.html`.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Softw. Concepts Tools*, 17(3):93–102, 1996.

[LSB17] Andreas Lochbihler, S. Reza Sefidgar, and Bhargav Bhatt. Game-based cryptography in HOL. *Arch. Formal Proofs*, 2017.

[Now07] David Nowak. A framework for game-based security proofs. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2007.

[Now08] David Nowak. On formal verification of arithmetic-based cryptographic primitives. In Pil Joong Lee and Jung Hee Cheon, editors, *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*, volume 5461 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2008.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[Pet15] Adam Petcher. A Foundational Proof Framework for Cryptography. PhD thesis, Harvard University, 2015. Available at https://dash.harvard.edu/bitstream/handle/1/17463136/PETCHER-DISSERTATION-2015.pdf.

[PM15] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *International Conference on Principles of Security and Trust*, pages 53–72. Springer, 2015.

[Poi05] David Pointcheval. Provable security for public key schemes. In *Contemporary cryptology*, pages 133–190. Springer, 2005.

[Rab79] M. O. Rabin. Digitized Signatures and Public Key Functions as Intractable as Factorization. Technical report, USA, 1979. Available at http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-212.pdf. Accessed August 3, 2021.

[Sch21] Peter Scholze. Half a year of the Liquid Tensor Experiment: Amazing developments. https://xenaproject.wordpress.com/2021/06/05/half-a-year-of-the-liquid-tensor-experiment-amazing-developments/, 2021. Accessed July 20, 2021.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, 2004:332, 2004.

[Vau98] Serge Vaudenay. Cryptanalysis of the chor-rivest cryptosystem. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 1998.

Appendix A

A Step-by-Step Proof in Lean

In this section, we complete a step-by-step proof in Lean to better illustrate the interactive tactic mode. Recall the definition of an even integer from Sec. 2.1:

```
def is_even (n : ℤ) : Prop := ∃ m, n = 2 * m
```

The formal proof which we investigate is given below:

```
theorem even_add_two_even (n : ℤ) :
   is_even n → is_even (n + 2) :=
   begin -- (1)
   intro ha, -- (2)
   cases ha with ka hka, -- (3)
   use (ka + 1), -- (4)
   rw hka, -- (5)
   ring, -- (6)
end
```

In words, `even_add_two_even` states that adding two to any even integer yields an even integer. The state of Lean at each line in the proof of `even_add_two_even` is shown in Fig. A.1. Anything written after double dashes `--` is ignored by Lean, so users can write comments or documentation.

Step (1) is the state when the cursor is placed after the `begin` which opens tactic mode. Our only assumption so far is that `n` is a term of type ℤ. Step (2) is after we use the `intro` tactic to introduce the assumption `is_even n` into the local context with the

Figure A.1: State of Lean at each step in the proof of `even_add_two_even`.

identifier `ha`. It is common to prefix identifiers for hypotheses with an `h`, though it is important to remember a hypothesis is just a term of some type in Lean.

In step (3), the `cases` tactic splits the assumption `ha` into two assumptions: first, `ka` is a term of type $\mathbb{Z}$, and second that `ka` satisfies the relationship `n = 2 * ka`. The latter hypothesis is given the identifier `hka`. As in `is_even 2`, the `use` tactic in step (4) makes the claim that `(ka + 1)` is the integer `m` such that `(n + 2) = 2 * m`. The last step is to prove the claim of the previous sentence. The `rw` (rewrite) tactic in step (4) tries to match the left side of the hypothesis `hka` with the current goal. Since `n` appears in the goal, it is rewritten by the right side of `hka`. As in the Sec. 2.1, we use the `ring` tactic automatically close the goal. in frame (6).

A next step might be to generalize further and prove `even_add_even_even`:

```
theorem even_add_even_even {a b : ℤ} :
   is_even a → is_even b → is_even (a + b) := by sorry
```

In this same way, one can build up an entire theory of even numbers, and ultimately all of mathematics (exercise left to the reader).

Appendix B

Summary of cryptolib

- `ddh.lean` - Provides a formal specification of the decisional Diffie-Hellman assumption on a finite cyclic group. Includes 35 loc, 3 definitions, 0 theorems, 0 lemmas.
- `elgamal.lean` - Contains the formal specification of the ElGamal public key encryption protocol, and the formal proofs of correctness and semantic security. Includes 372 loc, 7 definitions, 6 theorems, 7 lemmas.
- `negligible.lean` - Defines negligible functions and provides several useful lemmas regarding negligible functions. Includes 227 loc, 2 definitions, 1 theorem (unproven), 6 lemmas.
- `pke.lean` - Provides formal definitions for correctness and semantic security of a public key encryption scheme. Includes 50 loc, 4 definitions, 0 theorems, 0 lemmas.
- `tactics.lean` - Provides the `bind_skip` and `bind_skip_const` tactics to help prove equivalences between `pmf`s. Includes 28 loc, 2 (tactic) definitions, 0 theorems, 2 lemmas.
- `to_mathlib.lean` - Contains general lemmas written in the course of the cryptolib development for inclusion into mathlib. Includes 124 loc, 1 definition, 5 theorems (instance proofs, e.g. that `pmf` is an `instance` of `monad`), 8 lemmas.
- `uniform.lean` - Defines the uniform distribution over a finite group as a `pmf`, including the special case of $\mathbb{Z}_q$, the integers modulo q, and provides two useful lemmas regarding uniform probabilities. Includes 47 loc, 4 definitions, 0 theorems, 2 lemmas.