Formalizing Differential Privacy

Mads Buch, 20112625

Master's Thesis, Computer Science
August 2017
Supervisor: Aslan Askarov
Co-supervisor: Bas Spitters

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

On mathematical perception: "either you have no inkling of an idea or, once you have understood it, this very idea appears so embarrassingly obvious that you feel reluctant to say it aloud; moreover, once your mind switches from the state of darkness to the light, all memory of the dark state is erased and it becomes impossible to conceive the existence of another mind for which the idea appears nonobvious."

- Mikhael Gromov, *Source: M. Berger, Encounter with a geometer*

Abstract

Differential privacy's primary goal is to release data in a privacy-preserving manner. Differential privacy is the name of a framework that quantifies privacy-loss from one database to another. For a protocol to be differentially private, we prove an upper bound on the privacy loss between two adjacent databases. Databases are adjacent if they satisfy an adjacency relation. The technique promises that performing a single alternation to a database poses no statistical difference in the output of a differentially private protocol. This is promising for privacy, but implementing such techniques can be hard and raises several subtleties. To compensate we employ formal techniques to ensures that an implementation is correct.

Barthe et al. [A5] propose an approximate relational Hoare logic (apRHL). If we prove that the output of two instances of a protocol, each running on one of two adjacent databases, are approximately identical, then the protocol is differentially private.

The apRHL has been implemented in EasyCrypt in an experimental branch. It allows us to implement the proofs of differential privacy and have the proofs machine checked. We explore machine checked proofs for 4 case studies: the randomized response protocol, the Laplace mechanism, sums over streams, and the sparse vector technique.

The randomized response protocol allows to collect truth values from a large number of individuals in a way that ensures deniability. Because of fundamental features of the protocol, we can derive statistics with a limited privacy loss. This protocol has been implemented in EasyCrypt.

The next case study presents the Laplace mechanism. The mechanism is built into the logic. We see how to couple Laplace samplings and how we prove differential privacy for lists with Laplace noise.

As a more complicated case study, we present sums over streams. This displays a real application where differential privacy can be applied and where we use the apRHL to prove correctness.

The last case study is in the sparse vector technique. The central element for this case study is the proof of the above threshold protocol. It works as follows: given a list of queries, report the first query that yields a result greater than a given threshold. The surprising fact is that we can do it with a constant privacy loss indifferent to how long the list of queries is.

This work is of importance as we upload increasingly larger amounts of personal data and we need foundational techniques to process it in a privacy preserving manner

Acknowledgments

First, I would like to thank Bas and Aslan for their collaboration on this project. The project is based in an active area of research where navigating can be hard. Bas and Aslan have provided a crucial mix of literature, connections to the right people, and encouragement in hard times.

Next, I want to thank the EasyCrypt team who has been very helpful answering questions.

Lastly, thanks to everybody who have helped me keep my spirit high and supported me through the process: Sofie Eckmann, Mark Gottenborg, Jacob Buch, Mathias Pedersen, James Mashford, Joachim Harris, and Frederik Skytte.

Mads Buch,
Aarhus, August 1, 2017.

Resources

The four case studies each implement a proof using the EasyCrypt theorem prover. If the reader wants to check the proofs, following resources are available.

- The page `http://www.madsbuch.com/thesis` contains all relevant resources for this thesis.
- A virtual machine is available. It is configured with the correct version of EasyCrypt with integration to ProofGeneral. All the case studies are bundled with the machine.
- Installation instructions for EasyCrypt is available from their public repository: `https://github.com/EasyCrypt/easycrypt`. We note that **apRHL** support is needed for the proofs to pass.

Contents

Chapter 1

Introduction

A trusted party uses differential privacy to release data in a privacy-preserving manner. This happens through a number of differentially private protocols. However, assessing differential privacy for these protocols is hard by traditional software development techniques such as testing and review. The development in this area has a great impact on people and we need a rigorous formalism to discriminate protocols that are not differentially private.

This thesis looks at the current state of the art within formal methods for differential privacy. The goal is to understand differential privacy from a formal, relational point of view. This has two parts: the first part presents the foundations of the field, and the second part presents four case studies.

The first part elaborates on the foundations. We get acquainted with the problem, the methodology, and the tools we use. These elements are provided through three chapters.

- The first chapter introduces differential privacy. This introduction spans from the motivation, why we want to spend time doing it, to a more technical characterization of differential privacy and a presentation of the protocols in question.
- The next chapter develops a logic for reasoning about differential privacy known as approximate relational Hoare logic (apRHL). It starts by looking at the mathematical preliminaries to define distributions, lifting, approximate lifting, etc. Then we develop a syntax and semantics for a language to express probabilistic programs. Lastly, we delve into a number of proof rules for the formal reasoning.
- The last chapter concludes with a presentation of EasyCrypt. EasyCrypt is a theorem prover with support for probabilistic reasoning through a number of Hoare logics. It also includes the `aequiv`-judgment which implements apRHL.

The reader should at this point have an idea about the setting and be able to follow along with four case studies. These case studies attempt to justify the logical framework by presenting protocols with real world use.

- The first case study is the randomized response protocol. This protocol is often used to convey the fundamentals of differential privacy and fits the purpose of an introduction to the proof methodology. The case study presents a semantic proof for differential privacy implemented in the pRHL and pHL logics of EasyCrypt.

- The next case study presents the Laplace mechanism. The Laplace mechanism is widely considered a fundamental building block for differentially private protocols. Furthermore, it is also a basic proof principle in the apRHL logic.
- The third case study looks into sums over streams. The application is constrained to finite lists. This is primarily because of limitations on termination in the EasyCrypt theorem prover, but it is considered general enough. This case study shows some of the challenges associated with implementing proofs in a proof assistant.
- The last case study explains the sparse vector technique with some of its subtleties. This technique has been widely discussed in the community [A11, A1, A2]. The sparse vector technique spans a wide variety of protocols. The common denominator is the attempt to release the answers to a small subset of a large set of queries. The protocols vary in the release condition and whether it is offline or online. The primary proof of the case study is the Above Threshold protocol. It is implemented in an online and offline setting.

Furthermore, it should be noted that this field is cutting edge research. As such, information is mainly collected from newly released papers and manuals, from browsing the source code of EasyCrypt and from conversations, but also from experimentation and trial and error.

1.1 Prerequisites

This thesis assumes background knowledge in some well-defined scientific fields. These include following

- **Programming Languages:** Principles of programming languages including syntax, semantics, and general terminology.
- **Logic**: Principles of natural deduction and theorem proving.
- **Probability Theory:** Basic terminology and notions such as distributions, expectation, and more.

Part I

Foundations

Chapter 2

Differential Privacy

This chapter provides an overview of differential privacy. For this presentation, we have three components.

1. First, the chapter introduces differential privacy. This introduction includes what problem differential privacy solves and why we need a rigorous formalization.
2. Next, this chapter attempts to characterize differential privacy. This characterization includes some terms, some definitions, and the most important properties of differential privacy.
3. The last section introduces a number of protocols. These are the protocols that the rest of the thesis considers.

2.1 Introducing Differential Privacy

Before going too deep into terminology and mathematical definitions, we need to understand why it is worthwhile to consider differential privacy and, in particular, why we want to go to great lengths to construct a formal language and proof rules to reason about it.

2.1.1 Motivation

To motivate we look at an incident and an application of differential privacy. The most well-known incident seems to be the Netflix-incident, while an obvious application is in the health industry. However, there exists many more cases of de-anonymization.

Netflix Netflix was responsible for one of the most prevalent cases of a subtle privacy leak in 2006. They released a dataset containing over 100 million data points including *movie id*, *user id*, *date of rating*, and *rating*. This data was a part of a competition where everybody was free to develop a learning algorithm to predict the rating from the other fields.

The core argument for privacy was that the user was anonymized such that no information from a given row could identify a real person. Users are only identifiable by their *user id* which, in turn, provides a tool for collecting ratings from the same user.

The question is: how much auxiliary information do we need to correlate a real person to the *user id*? Narayanan and Shmatikov [A12] did a case study on this exact question using the Netflix dataset.

> With 8 movie ratings (of which 2 may be completely wrong) and dates that may have a 14-day error, 99% of records can be uniquely identified in the dataset.

Evidently, a lot of information is leaked. This information could populate databases for targeted marketing (commercial or political), it could be used to infer political standpoints, etc. While this might not be evil, it should at least be with the consent of the users.

Health Data Let us now consider all the health data collected by the Danish health institutions. Imagine coupling that with people's habits. The resulting database probably has answers on quite some of our health problems. At first, we might think that releasing such a database would be of great benefit to all of us. We could let all kinds of researchers, hackers, and enthusiasts try to come up with general algorithms for predicting diseases, completely like the Netflix competition and then use the result to improve everybody's health.

However, if we take a second glance, we see that this database, in the hands of an insurance company, could force people with high risks of diseases to pay more in premium, indifferent to how fair that is to the individual. It could be used by employers in the decision whether a person is eligible for a job or not. The misuses are numerous.

Differential privacy has already been deployed in health care [A7]. When new clinical trials are set up there is a need for participants satisfying certain conditions. The number of potential candidates are crucial, to decide if it is feasible to commence the trial. Hence, an investigator needs to query this database a large number of times when designing clinical trials. If the investigator were provided with unrestricted access to the database, it would yield a serious privacy loss. In order to solve that problem they employed a modified version of the exponential query that provides differential privacy.

The Solution: Differential Privacy Intuitively, we can ensure privacy by stripping the released data for any personal information such as names, addresses, and birthdays. While this might seem to be a solution to the problem, it is not enough. As the Netflix case showed, it is possible to identify records by quite a small amount of auxiliary data.

To develop a privacy-preserving system, we must understand what privacy is. A potential definition could be that we can not learn anything about the individuals in a dataset. However, this approach does not work. It is intrinsic to data analysis that we learn something about the population. E.g. if our preconception is that the half of all humans had only one leg, we would from a relevant dataset quickly be able to see that a person with a high probability has two legs. Hence this approach would also fail.

The privacy model we use relies on changes to the dataset. We want to disguise single operations to ensure that changes to the database can not be measured. The differentially private protocol introduce enough noise that we can not use the techniques proposed for de-anonymization.

Differential privacy is first and foremost the name of a formal framework for quantifying privacy loss. The overall goal of differential privacy is to be able to release data to the public without worrying about leaking personal information.

2.1.2 Formal Framework

Previously, we presented the reasons to consider differential privacy. This thesis presents a formal framework for differential privacy. As such, we want to justify the need for a rigorous formal framework for differential privacy.

We employ formal methods to ensure software safety. Software safety, in its most basic meaning, says that a program will not go wrong; it makes sure that compiled code will never run into an illegal state.

We use the same argument for differential privacy, with a twist. Here we say that a program goes wrong if it does not live up to its differential privacy promise. We assume a program that promises to be (ϵ, δ)-differentially private. Then we want to ship this program with a certificate, a proof that it is indeed (ϵ, δ)-differentially private.

We can let this certificate be a number of deduction steps in a certain logic that promises differential privacy for all programs inhabiting it. Barthe et al. developed such logic. They propose the approximate probabilistic relational Hoare logic (apRHL), which is the framework this thesis presents.

The need for formal methods scale with the impact of the application. We want our cars to drive safely and our space stations to stay in space. As the above motivational examples showed, differential privacy can have an enormous impact on each person.

2.2 Characterizing Differential Privacy

A part of the terminology is specific to a given application. As such, it is not strictly defined. Below we list those terms and attempt to explain their intuition.

- **Randomized Algorithm:** A randomized algorithm is an algorithm which in some way employs random data or computations. We will think of it as having access to probability distributions from which we can sample.
- **Database:** Usually, we think about it as a list of tuples (rows and columns). However, for example for the randomized response, the database is a single boolean value, where it for the Laplace mechanism is an integer value. Hence, we allows some freedom in the use of the term.

- **Norm:** The norm is the distance between two databases. In the case of collections and tuples, we say that the norm is the number of tuples modified, inserted, or deleted in the collection. For integers, it is the absolute value of their difference and for booleans, the norm can either be 1 or 0.
- **Mechanism:** A mechanism is a randomized algorithm that takes a database and produces a result that is differentially private under the assumption of a given norm on the databases.
- **Data Curator:** The trusted party who executes the differentially private algorithms.

Given above terms, we are now able to define differential privacy.

Definition 2.1: Differential Privacy

(Dwork and Roth [A8]). A randomized algorithm, $M : A \to B$, is said to be (ϵ, δ)-differentially private w.r.t. an adjacency relation Ψ (typically a norm of 1) if $\forall S \subseteq B$ and $\forall x\ y \in A,\ x\ \Psi\ y$ where following property holds.

$$Pr[M(y) \in S] \leq \exp \epsilon \times Pr[M(x) \in S] + \delta \tag{2.1}$$

As both the name and the definition suggests, differential privacy quantifies the difference between two databases and bounds it as given in definition 2.1.

For this definition, there are two components we need to understand: the epsilon and delta parameters.

- ϵ: Epsilon denotes the privacy loss. We can understand the parameter such that the higher value it has, the more information we leak. This interpretation is also consistent with the fact that we allow larger bound on adjacent databases.
- δ: Probability that we violate the privacy cost. This parameter is not always considered, but it provides a generalized understanding of differential privacy. We can understand such that the higher value it has, the greater is the change of releasing the result without any random noise added. Setting it to 1 removes all privacy guarantees.

Queries and Sensitivity It is very useful to allow the user of a database to specify her *queries*. However, this flexibility poses a problem: we can not look into the queries to tell if it is a part of an attack. To mitigate this, we introduce the notion of *sensitivity* for queries.

Definition 2.2: Sensitivity

We characterize queries as pure functions mapping a database to an integer. A function, $f : A \to \mathbb{Z}$, is said to be k-sensitive if

$$|f(x) - f(y)| \leq k$$

for $x \; \Psi \; y$ where Ψ is an adjacency relation as introduced in definition 2.1. This function is also known as the Lipschitz function.

As definition 2.2 formalizes, k-sensitivity refers to how different the result of a query can be on two adjacent databases.

Accuracy The accuracy is a measure of how accurate the result is. At an extreme one could make a differentially private mechanism by just outputting complete randomness. However, this would yield a bad accuracy.

It should not surprise that there is a trade-off between accuracy and privacy. The more noise we add to a given database, the more we protect its participants. On the other hand, the data will be less accurate.

2.2.1 Important Properties

We list a number of properties in this subsection that is considered fundamental to differential privacy.

Compositionality One of the main properties of differential privacy is compositionality. In many expositions, we use a number of proven differentially private mechanism and compose them into more specialized mechanism better suited for certain applications.

Theorem 2.1: Sequential Composition

Let $M : D \to R$ and $M' : D \to R \to R'$ be respectively (ϵ, δ) and (ϵ', δ') differentially private. We can sequence these two algorithms

$$\begin{aligned} r &= M \; d \; ; \\ result &= M' \; d \; r \end{aligned}$$

to obtain $result$ that is $(\epsilon + \epsilon', \delta + \delta')$ differentially private.

The composition principle in Theorem 2.1 composes differentially private protocols heterogeneously, i.e. each protocol does not have to spend the same amount of privacy.

There is also advanced composition which makes the trade-off of a lower ϵ for a higher δ. This principle can be applied on homogeneous sequencing and is better suited for loops where the same mechanism is applied a number of times.

Often, we will see the term *privacy budget* used. This refers to the accumulative property of the standard composition. I.e. if we answer multiple queries, in the simplest case, we add their privacy costs. In Theorem 2.1 we say that M spends (ϵ, δ) privacy and M' spends (ϵ', δ') privacy.

Post Processing Differential private algorithms are resilient to post processing. This means that no amount of computational resources can make a differentially private protocol less private. Resilience to post processing is an important property, as it shows that we retain privacy after releasing data.

2.3 Protocols

Having introduced the basic notions for differential privacy, we move on to the applications. In this section, we will look into some protocols that are differentially private.

2.3.1 Randomized Response

The randomized response query is a per row or per participant protocol which ensures plausible deniability for the participant. It does this by randomizing the response such that individual participants do not necessarily apply to their true answer. Each participant is asked to answer based on following protocol.

1. Flip an honest coin.
2. If the coin shows tail, then tail answer the question truthfully.
3. If it shows a head flip a new honest coin. Answer yes if it shows tail and no for a head.

We use the coin as a metaphor for a distribution over a Boolean value with half probability for each outcome. We use this model in the proof.

2.3.2 Laplace Mechanism

The Laplace mechanism is considered one of the basic mechanism in differential privacy. It builds on the Laplace distribution which forms a top that decreases exponentially to both sides.

The Laplace mechanism draws noise from the Laplace distribution with ϵ spread and center at the input value.

Note: The notation varies in the literature. If we compare the work by Dwork and Roth [A8] with recent work from Barthe et al. [A1], we see that the numerator and denominator of ϵ have switched places, i.e. what we denote $\frac{2}{\epsilon}$ in first work is denoted $\frac{\epsilon}{2}$ in the latter. In this thesis, we follow Barthe et al.

2.3.3 Sums over Stream

Given a stream of data, some applications might need the aggregate result for every q'th element. This is the goal we attempt to solve by a number of protocols.

- **Partial Sum:** The partial sum algorithm aggregates q elements from the stream, then it adds Laplace noise and releases the answer. This technique offers good accuracy but a weak privacy guarantee.
- **Partial Sum':** The partial sum' algorithm adds Laplace noise to each element in the q-size substring. Then it adds them together and releases the answer. This technique offers a good privacy guarantee, but a bad accuracy as we add noise to each element.
- **Smart Sum:** The smart sum algorithm works as a combination of the two previously mentioned algorithms to achieve better accuracy with less privacy loss than the first algorithm.

These approaches are elaborated in the case study on sums over streams.

2.3.4 Sparse Vector Technique

The sparse vector technique is a widely considered set of algorithms in the community of differential privacy. As the starting point, we have a list of queries (a vector of queries). The list is large enough that it is implausible to answer all of them. We then ask if we can derive a *sparse* vector containing the c first queries we can answer. Surprisingly, it *is* possible.

For the Above Threshold protocol, the client supplies a threshold and gets the first query that yields an answer above provided threshold. We sequence that algorithm c times to get the c first queries above the threshold.

What is surprising is that we do not pay for all the negatives. We *only* pay to know the queries that are included in the result set.

We consider a number of variations:

- **Release Condition:** The algorithm outlined above used a threshold to define a release condition. We also look into an implementation that uses two thresholds where the query has to be between those. This algorithm goes by the name Between Threshold.
- **Query Passing:** The SVT comes in two variants w.r.t. how queries are passed: an adaptive (or online) edition, and the offline edition. The adaptive edition lets the client choose the next query based on the current result set.
- **Sequencing:** To return more than a single query satisfying the release condition we sequence the protocols.

We will look further into this algorithm in the associated case study.

Chapter 3

Probabilistic Framework

This chapter presents notions from probability theory and programming language. We do this from an information theoretic angle omitting concerns about complexity and availability of randomness.

The chapter will progress as follows:

1. The first section looks into the mathematical preliminaries. Firstly, it presents distributions, how they are defined, and how we treat them. The section continues with the tools needed to develop a framework for reasoning about differential privacy. These tools include coupling proofs, lifting, and approximate lifting.
2. Next, we define the syntax and semantics of the language we will use for the remainder of this chapter.
3. Lastly, the chapter presents the logics we use in EasyCrypt. This section will mainly consider apRHL. We will look into the proof rules and see how they support our goal.

3.1 Mathematical Preliminaries

In this section, we seek to provide a fundamental understanding of the tools we use to build our logical framework by presenting distributions and associated notions. Then it introduces ideas used to formalize reasoning about differential privacy. These ideas include couplings, lifting, and approximate lifting.

3.1.1 Distributions

Following the work of Barthe et al. [A1, A3] we consider distributions and sub-distributions over discrete sets. In this setting, a probability distribution is a function that maps an element $a \in A$ to a probability that the element is sampled from the distribution. We distinguish distributions from sub-distributions based on whether they span the full probability space, i.e. the sum of the probability of all elements is one. This gives rise to definition 3.1.

Definition 3.1: (Sub-)Distribution

A function $\mu : A \to \mathbb{R}^+$ is said to be a distribution if $\sum_{a \in A} \mu(a) = 1$ and a sub-distribution if $\sum_{a \in A} \mu(a) \leq 1$.

Some subtleties: The expectation function is defined *by* the distribution As such, we will not explicitly define an expectation function. Furthermore, the definition restricts the functions co-domain to positive real numbers.

Another definition we need for our further investigations is the support of probability distributions. The support, denoted *supp*(A), is simply the set of all the elements from the domain with a probability greater than 0. This gives rise to definition 3.2.

Definition 3.2: Support

The support of a distribution $\mu : A \to \mathbb{R}^+$ is all the elements from its domain with probability greater than 0. That is

$$\mathit{supp}(A) = \{a \in A \mid \mu(a) > 0\}.$$

Next, we define distributions over products. Products are intrinsic to relational reasoning, which is our main goal. The following two definitions allow us to construct product distributions and to destruct a product by projecting it.

Definition 3.3: Product Distribution

A product distribution is a distribution where the domain is a product type, i.e. a distribution

$$\mu : A \times B \to R^+.$$

The product construction, as expected, simply pairs up two distributions. The projection has a little more to it. Here we need to aggregate the projected sets. This is done by addition which preserves probabilities for the projected part.

Definition 3.4: Distribution Projection

Given at product distribution $\mu : A \times B \to \mathbb{R}^+$ the first projection, also called *first marginal*, is the sum of probabilities holding the first element of the product fixed. More formally, we write

$$\pi_1(\mu)(a) = \sum_{b \in B} \mu(a, b)$$

where $a \in A$. The definition works equivalently for the second marginal.

A probability distribution is constituted by a pure function mapping an element to a probability of that element is in the set. This characterization is slightly simpler than the measure theoretic approach, where we also define a measure function and pass to the expectation function. However, this model suffices for our needs.

Probability Monad Probability distributions have a monadic structure [A10]. By definition 3.1.1, this structure tightly resembles the continuation monad and composes the expectation functions. A monadic structure is important for following reasons.

- We will use it to define the semantics of the `pWhile` language.
- It provides a coherent way of talking about *all* distributions on a given set, i.e. we denote a distribution over A as $\mu \in Distr(A)$.

From now on, the set of distributions on some set A is denoted $Distr(A)$.

For an object to be a monad, it must support two operations: the **`bind`** operator and the **`unit`** operation. They should both obey the monad laws. The **`unit`** operation amounts to the Dirac distribution, i.e. the distribution over a single element. The **`bind`** operator takes a distribution and wraps it in a function to construct another distribution.

Figure 3.1: Monad rules for the *Distr* monad.

(Giry [A9]). Consider the two monad constructors: unit : $A \to Distr(A)$ and bind : $Distr(A) \to (A \to Distr(B)) \to Distr(B)$.
We can interpret the monad operations into a probability mass function.

$$\llbracket \text{unit}(v) \rrbracket_{\mathcal{D}}(x) = \begin{cases} 1 & \text{if } v = x \\ 0 & \text{if } v \neq x \end{cases}$$
$$\llbracket \text{bind}(d, h) \rrbracket_{\mathcal{D}}(x) = \sum_{v \in supp(d)} \llbracket d \rrbracket_{\mathcal{D}}(v) * \llbracket h(v) \rrbracket_{\mathcal{D}}(x).$$

Figure 3.1 explains the interpretation of the monad operations into a distribution applied to x.

In addition to the two operations discussed above, we also assume distributions to be in the probability monad. For the discussion of apRHL we only augment the monad operations by the Laplace distribution. We assume that the Laplace distribution is available as $lap : Real \to Int \to Distr(Int)$ where the first argument is the spread and the second is the center.

3.1.2 Coupling Proofs

Coupling proofs are a technique widely used in probability theory. Informally, we make a coupling between two probability distributions and use it to show some property about them.

As an example, we consider two coins: C_1 and C_2. We fix their probabilities such that $Pr[C_1 = head] = p$, $Pr[C_2 = head] = q$ and $p < q$. Now, in a stream of pairs of coin tosses over the coins, we want to show that C_2 will produce more heads than C_1. However, showing this can be challenging just by counting, as we might get more heads from C_1 in any arbitrary substring.

The solution is to employ a *coupling* between the two distributions. The process is as follows: We make two streams such that C_1 generates the first. The second is constructed to have the same probabilistic properties as C_2. We construct it such that it will always yield head when the sample from C_1 does. Otherwise, it yields head with the probability $(q - p)$. We now have that C_2 still emits head with the correct probability. Furthermore, we also *know* that C_2 samples more heads than C_1. This proposition is evident from the fact that we sample at least as many heads from the coupling. Finally, we still have a chance to sample a head when C_1 samples a tail.

Given that example, we can move on to the definition.

Definition 3.5: Coupling

A coupling of two distributions μ_1 and μ_2 is given by a *witness distribution* μ such that μ_1 and μ_2 are projections of μ. More formally we write

$$\mu \blacktriangleleft \langle \mu_1 \ \& \ \mu_2 \rangle \quad \text{if} \quad \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2.$$

The definition of couplings considers a *witness* distribution. The witness distribution was not explicit from the above example. We merely constructed a new stream of coin tosses. To remedy that, we will in the following lemma consider an explicit witness distribution.

Lemma 3.1: Biased coins

Given $p \in (0,1)$ and $q \in (0,1)$ such that $p < q$, we consider two distributions μ_1 and μ_2 with following properties:

$$\begin{aligned} Pr[\mu_1 = 1] &= p \qquad & Pr[\mu_1 = 0] &= 1 - p \\ Pr[\mu_2 = 1] &= q \qquad & Pr[\mu_2 = 0] &= 1 - q \end{aligned}$$

For an arbitrary sequence of k samples, there is a higher probability that μ_2 produces k 1's than for μ_1.

Proof. For this proof we will employ an explicit witness of a coupling between μ_1 and μ_2. the witness is defined as follows.

$$\begin{aligned} \mu = \{&(1,1) \mapsto p \\ &(1,0) \mapsto 0 \\ &(0,1) \mapsto q - p \\ &(0,0) \mapsto 1 - q\} \end{aligned}$$

To verify that it is indeed a witness we take the projections and compare them to μ_1 and μ_2.

$$\begin{aligned} \pi_1(\mu) = \{1 &\mapsto p && \text{(from } p + 0\text{)} \\ 0 &\mapsto 1 - p\} && \text{(from } q - p + 1 - q\text{)} \end{aligned}$$

and

$$\begin{aligned} \pi_2(\mu) = \{1 &\mapsto q && \text{(from } p + q - p\text{)} \\ 0 &\mapsto 1 - q\} && \text{(from } 1 - q + 0\text{)} \end{aligned}$$

It is straight forward to see that the projections satisfy distribution equivalence, and we have a witness in order.

Lastly, we justify that μ_2 has a higher probability of emitting a k-sized string of 1's than μ_1. This is directly evident from the witness distribution μ. If we get a 1 on the left hand, we necessarily also get it on the right hand, however, if we get a 1 on the right hand we might still get 0 on left hand.

This concludes this proof. ■

Lemma 3.1 states an explicit witness distribution. This is in contrary to the applications we consider later. In section 3.3.1, we consider the pRHL logic. In this logic, we only need to prove the *existence* of a witness distribution, not an actual implementation.

That said, for some applications, the explicit witness is necessary, e.g. the example presented in this chapter. The proof of Lemma 3.1 heavily relies on the actual witness distribution to justify its claim.

To cope with that requirement, we can use a probabilistic lifting. It allows us to restrict the coupling such that the mere existence of a coupling gives us information about the relation between two probability distributions.

3.1.3 Lifting

Lifting refers to a process that takes two probability distributions and lifts them into a relation. This relation defines a support for an arbitrary lifting operator for which we need to justify the existence of a witness distribution.

Definition 3.6: Lifting

Given a relation $\Psi \subseteq (A_1 \times A_2)$ and two distributions $\mu_1 \in \mathit{Distr}(A_1)$ and $\mu_2 \in \mathit{Distr}(A_2)$ we say that the distributions are lifted by the relation if there exists a coupling $\mu \blacktriangleleft \langle \mu_1 \ \& \ \mu_2 \rangle$ and $\mathit{supp}(\mu) \subseteq \Psi$. More formally we write

$$\mu_1 \Psi^{\#} \mu_2 \quad \text{if} \quad \exists \mu. \mu \blacktriangleleft \langle \mu_1 \ \& \ \mu_2 \rangle \wedge \mathit{supp}(\mu) \subseteq \Psi.$$

To put this definition into perspective, we consider the example 3.1 below. From the example, it is evident that the two probability distributions for coins used earlier are not equivalent. This is established solely from the fact that we can not find a witness probability distribution.

Example 3.1: Lifting restricts coupling

As an example to see how lifting works, we take the same probability distributions as presented in Lemma 3.1. We lift them using the equality operator to check if

$$\mu_1 (=)^{\#} \mu_1.$$

If we assume the same witness distribution as in the lemma, we need to only consider the subset under the relation (per second rule of lifting). This restriction yields the following witness.

$$\begin{aligned} \mu = \{&(1,1) \mapsto p \\ &(0,0) \mapsto 1-q\}. \end{aligned}$$

If we take into account that $p \neq q$, it is clear to see that there exist no projection to the original distributions. To support this we can consider the properties for projection $\pi_1(\mu)$.

$$\begin{aligned} Pr[\pi_1(\mu) = 1] &= p \\ Pr[\pi_1(\mu) = 0] &= 1-q. \end{aligned}$$

The properties above are not the same as those for μ_1 in Lemma 3.1. Further, We remark that if $p = q$ there is a coupling. This is in compliance with the relation we lifted the distributions over.

Lifting as defined above still only lets us reason about the exact relation between probability distributions. Hence we also call it an *exact lifting*. This does not fit the definition of differential privacy. This definition allows some wiggle room bound by ϵ and δ. To allow this, we employ an approximate lifting.

3.1.4 Approximate Lifting

The previous section presented the exact lifting. This notion does not suffice when we want to reason about the approximate relationship between probability distributions, which is intrinsic to differential privacy.

The approximation introduced by the definition of differential privacy closely resembles the approximation we introduce in the definition of approximate lifting. The definition bases approximation on a notion of the statistical distance we call ϵ-DP divergence which is defined as follows.

Definition 3.7: ϵ-DP Divergence

(Barthe and Olmedo [A6]). The ϵ-DP divergence of two distributions, $\mu_1, \mu_2 \in Distr(A)$ is denoted $\Delta_\epsilon(\mu_1, \mu_2)$ and is defined as

$$\sup_{S \subseteq A}(Pr_{x \leftarrow \mu_1}[x \in S] - exp(\epsilon) Pr_{x \leftarrow \mu_2}[x \in S]).$$

This resemblance to differential privacy should be apparent when we use the above definition in a relation such that $\Delta_\epsilon(\mu_1, \mu_2) \leq \delta$. By unfolding this definition, there is a relation to Definition 2.1. For the proof that differential privacy is a consequence of this type of lifting we refer to the work by Barthe et al. [A2].

Next, we want to define approximate lifting. An approximate lifting is a lifting with a level of indirection. We introduce two witness distributions which each represent one of the distributions we are lifting. Both of these distributions need to be supported by the relation Ψ. Furthermore, the predicate, $\Delta_\epsilon(\mu_1, \mu_2)$, puts a bound on the distance between the distributions.

Definition 3.8: Approximate Lifting

(Barthe and Olmedo [A6]). Two distributions $\mu_1 \in Distr(A_1)$ and $\mu_2 \in Distr(A_2)$ are related by an (ϵ, δ)-lifting Ψ denoted $\mu_1 \; \Psi^{\#(\epsilon,\delta)} \; \mu_2$ if there exist two witness distributions μ_L and μ_R such that they each are a lifting on the relation Ψ and their ϵ-DP divergence is bound by δ. Formally the following three properties has to be satisfied

1. $\pi_1(\mu_L) = \mu_1$ and $\pi_2(\mu_R) = \mu_2$,
2. $supp(\mu_L) \subseteq \Psi$ and $supp(\mu_R) \subseteq \Psi$,
3. $\Delta_\epsilon(\mu_L, \mu_R) \leq \delta$.

To put the definition in perspective, we provide the following example. The example uses the coins we previously defined.

Example 3.2: Approximately related coins

To exemplify an approximate lifting, we construct before mentioned coins such that they are approximately related, i.e. we lift them into the equality relation.
To do this, we consider the distributions from lemma 3.1. We introduce the variables, ϵ and δ, to bound the difference between p and q. This bound is set up such that

$$p - \exp(\epsilon) * q \leq \delta$$

where $p - \exp(\epsilon) * q \geq (1 - p) - \exp(\epsilon) * (1 - q)$.
The relation, Ψ, is given by the set $\{(0, 0), (1, 1)\}$ when defined for the equality operator over the domain of the distributions.
From here we need to construct the witness distributions.

$$\begin{aligned}\mu_L = \{&(1,1) \mapsto p \\ &(0,0) \mapsto 1 - p\} \\ \mu_R = \{&(1,1) \mapsto q \\ &(0,0) \mapsto 1 - q\}.\end{aligned}$$

We now see that the properties for approximate lifting are satisfied. The two first clauses are given directly from the construction of the witness distributions, where the last clause is given by our assumption on the bound on p and q.
Evidently, the bound closely resembles the definition of the ϵ-DP divergence used in the definition of differential privacy. However, it supports the type of reasoning we need: the larger we make ϵ and δ, the larger we allow the difference of p and q.
For $\epsilon = 0$ and $\delta = 0$ we see that we would need to have $p = q$ which is consistent with the definition of exact lifting, and supports that and exact lifting is an approximate lifting with both parameters set to 0.

3.1.5 Final Remarks

The three proof techniques we have looked at nicely fits on top of each other.

- Firstly, the coupling allows us to reason about products of distributions. The reasoning is tightly bound to the constructed witness distribution. However, this is not desirable as we want to disregard semantic considerations and build a logical language.
- Secondly, we define a lifting for probability distributions. This lifting nicely extends couplings by a definable relation Ψ. We now have a foundation for a logic. We can state propositions properties about products of

distributions without explicitly reasoning witness distributions. However, we still reason about the relation in exact terms.
- Lastly, we define the approximate lifting. This definition introduces yet another witness distribution such that we have dedicated witness distributions for each probability distribution we lift. The witness distributions each provide an exact lifting under the same relation. Furthermore, they are statistically bound by $\Delta_\epsilon(\mu_L, \mu_R) \leq \delta$ providing the approximation property.

We now have the tools we need to build a framework to reason about differential privacy. However, before doing that we introduce a language to express probabilistic computations.

3.2 The pWhile Language

In the previous section, we established the mathematical tools we need for formal reasoning about differential privacy. To utilize the tools, we build a language which in we can express probabilistic computations. We interpret it into the probability monad to end up with a probability distribution. As the interesting component we consider the commands mostly. We let the details of expressions, distributions, identifiers and so forth be mostly implicit.

3.2.1 The Syntax

The syntax for the language is based on a standard imperative `While`-language augmented with random assignment. The BNF for the syntax is in Figure 3.2.

Figure 3.2: The syntax for the pWhile language

$$\begin{aligned}
\mathcal{E} \;&::= \mathcal{X} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E} * \mathcal{E} \\
&\quad\mid \mathcal{E} \wedge \mathcal{E} \mid \ldots \\
\mathcal{C} \;&::= \mathcal{X} \leftarrow \mathcal{E} \\
&\quad\mid \mathcal{X} \overset{\$}{\leftarrow} \mathcal{D}\mathcal{E} \\
&\quad\mid \text{while } \mathcal{E} \text{ do } \mathcal{C} \\
&\quad\mid \text{if } \mathcal{E} \text{ then } \mathcal{C} \text{ else } \mathcal{C} \\
&\quad\mid \text{skip} \\
&\quad\mid \mathcal{C};\mathcal{C}
\end{aligned}$$

The sets defined in figure 3.2 should be understood as follows.

- $\mathcal{E}$: The set of expressions. This includes type-specific operations like addition and subtraction. For integers, application, and abstraction for functions, and conjugation, and disjunction for booleans.

- $\mathcal{C}$: The set defined of commands. We consider assignment, random assignment, while loops, if-statements, skip, and sequencing. This is standard choice for a basic programming language.

As expected from an imperative language we draw a distinction between expressions and commands. All expressions are typed under τ. The types are not explicit and can be inferred.

Furthermore, there are a number of implicitly defined sets. They work as follows:

- $\mathcal{X}$: The set of variable names. An element is denoted by x.
- $\mathcal{D}$: The set of distributions. It varies a bit how this set is used. For the apRHL logic presented later, this set will only be inhabited by the Laplace mechanism. In other logics, we consider unbiased and biased coins. The exact choice of inhabitants is not important as we can usually implement distributions in terms of those given.

To simplify further reasoning, we introduce syntactic sugar for the `if-then` statement and the while loop.

Figure 3.3: Syntactic sugar for the pWhile language

$$\begin{aligned}\text{if } b \text{ then } c &= \text{if } b \text{ then } c \text{ else skip}\\ [\text{while } b \text{ do } c]_0 &= \text{if } b \text{ then null else skip}\\ [\text{while } b \text{ do } c]_n &= \text{if } b \text{ then } c; [\text{while } b \text{ do } c]_{n-1}\end{aligned}$$

Figure 3.3 introduces an alias for the `if-then-else` statement that does nothing on the else branch. It also expresses the while loop through nested if statements. The precise meaning of this is specified in the semantics section.

Lastly, we see that procedures are not included. The reason is that the formal logic we consider next has not been extended to include procedures.

3.2.2 Typing

For this language, we choose to omit explicit typing. For the majority of the literature, this is based on; the semantics has been omitted, though some expositions lay it out [A13].

The justification for not providing typing is that this language is used in juxtaposition with a number of Hoare logics. These Hoare logics provide implicit typing.

The next chapter introduces EasyCrypt which is a proof system that implements a variant of this probabilistic while language. This implementation employs types.

3.2.3 The Semantics

Lastly, this exposition of the language defines its semantics. The denotations of the sentences in the pWhile language are interpreted into the probability monad and we see an instance of the language as a probability distribution.

Figure 3.4: The denotational semantics for the pWhile language

We assume denotations for expressions $[\![-]\!]_{\mathcal{E}}m$ and for distribution expressions $[\![-]\!]_{\mathcal{DE}}m$. The denotations for the commands interprets a program into a probability distribution over memories m and look as follows.

$$\begin{aligned}
[\![\text{skip}]\!]m &= \text{unit } m \\
[\![c; c']\!]m &= \text{bind } \ ([\![c]\!]m) \ [\![c']\!]m \\
[\![x \leftarrow e]\!]m &= \text{unit } (m\{([\![e]\!]_{\mathcal{E}}m)/x\}) \\
[\![x \xleftarrow{\$} \mu]\!]m &= \text{bind } ([\![\mu]\!]_{\mathcal{DE}}m) \ (\lambda v.\ \text{unit } (m\{v/x\})) \\
[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!]m &= \begin{cases} [\![c_1]\!]m & \text{if } [\![e]\!]_{\mathcal{E}}m = true \\ [\![c_2]\!]m & \text{if } [\![e]\!]_{\mathcal{E}}m = false \end{cases} \\
[\![\text{while } e \text{ do } c]\!]m &= [\![\sup_{n \in \mathbb{N}}([\text{while } e \text{ do } c]_n)]\!]m
\end{aligned}$$

This language is set up entirely to reason about the apRHL as presented later. For our applications, we need adversaries to model interaction. However, we omit the discussion of those and note that we can implement the functionality in EasyCrypt by escaping to pRHL where procedure calls and abstract model types (presented later on) provides the full functionality.

3.3 Formal Reasoning

Previously, this chapter builds a mathematical framework for reasoning about probability distributions for differential privacy. Then, it presented a language for which we can express the probability distributions. In this section, we bind it together and build a number of proof rules to do formal reasoning.

As the base, we use Hoare logic. This choice makes it possible to reason about the state; we setup assertions on the memory before and after execution.

3.3.1 Probabilistic Relational Hoare Logic

The probabilistic relational Hoare Logic (pRHL), is a logic that takes two distributions and relates them based on a precondition and a postcondition. It builds on the exact lifting and proves an exact relationship.

The judgment looks as follows.

Definition 3.9: pRHL Judgments

Let c_1 and c_2 be two programs of the pWhile language and Φ and Ψ be propositions that may involve memory of the programs such that variable x is accessed in the left-hand memory m_1 as $x\langle 1\rangle$ and the right-hand memory m_2 as $x\langle 2\rangle$. Then a pRHL judgment has the form

$$\vdash c_1 \sim c_2 : \Phi \implies \Psi$$

such that $\Phi(m_1, m_2) \implies [\![c_1]\!]m_1 \ \Psi^{\#} \ [\![c_2]\!]m_2$.

Informally, the judgment reads as follows: The two programs, c_1 and c_2, are related under the exact lifting Ψ given an initial state conforming to Φ.

For this thesis, we omit an in-depth walk-through of the pRHL logic as the main formal framework is the apRHL which is presented next.

3.3.2 Approximate Probabilistic Hoare Logics

To support differential privacy we look at the approximate probabilistic Hoare logic (apRHL). While the pRHL allows to reason about the exact relation between two probability distributions, this logic is used to relate two distributions by a bound

The judgment and definition look as follows.

Definition 3.10: apRHL Judgments

Let c_1 and c_2 be two programs of the pWhile language, Φ and Ψ be propositions that may involve memory of the programs such that variable x is accessed in the left-hand memory m_1 as $x\langle 1\rangle$ and the right-hand memory m_2 as $x\langle 2\rangle$, and $\epsilon, \delta \geq 0$. Then an apRHL judgment has the form

$$\vdash c_1 \sim_{\langle \epsilon, \delta \rangle} c_2 : \Phi \implies \Psi$$

such that $\Phi(m_1, m_2) \implies [\![c_1]\!]m_1 \ \Psi^{\#(\epsilon,\delta)} \ [\![c_2]\!]m_2$.

From definition 3.10, we jump directly into the rules for the language constructions.

Figure 3.5: Basic apRHL proof rules.

Assn

$$\frac{}{\vdash x_1 \leftarrow e_1 \sim_{\langle 0,0\rangle} x_2 \leftarrow e_2 : \Psi\{e_1/x_1, e_2/x_2\} \implies \Psi}$$

Cond

$$\frac{\vdash c_1 \sim_{\langle \epsilon,\delta\rangle} c_2 : \Phi \wedge b_1\langle 1\rangle \implies \Psi \qquad \vdash c_1' \sim_{\langle \epsilon,\delta\rangle} c_2' : \Phi \wedge \neg b_1\langle 1\rangle \implies \Psi}{\vdash \text{if } b_1 \text{ then } c_1 \text{ else } c_1' \sim_{\langle \epsilon,\delta\rangle} \text{if } b_2 \text{ then } c_2 \text{ else } c_2' : \Phi \wedge b_1\langle 1\rangle = b_2\langle 2\rangle \implies \Psi}$$

Seq

$$\frac{\vdash c_1 \sim_{\langle \epsilon,\delta\rangle} c_2 : \Phi \implies \Psi' \qquad \vdash c_1' \sim_{\langle \epsilon',\delta'\rangle} c_2' : \Psi' \implies \Psi}{\vdash c_1; c_1' \sim_{\langle \epsilon+\epsilon',\delta+\delta'\rangle} c_2; c_2' : \Phi \implies \Psi}$$

While

$$\frac{\vdash c_1 \sim_{\langle \epsilon,\delta\rangle} c_2 : \Theta \wedge b_1\langle 1\rangle \wedge b_2\langle 2\rangle \wedge e\langle 1\rangle = k \implies \Theta \wedge b_1\langle 1\rangle = b_2\langle 2\rangle \wedge e\langle 1\rangle < k \qquad \vDash \Theta \wedge e\langle 1\rangle \leq 0 \rightarrow \neg b_1\langle 1\rangle}{\vdash \text{while } b_1 \text{ do } c_1 \sim_{\langle \sum_{k=1}^n \epsilon_k, \sum_{k=1}^n \delta_k\rangle} \text{while } b_2 \text{ do } c_2 : \Theta \wedge b_1\langle 1\rangle = b_2\langle 2\rangle \wedge e\langle 1\rangle \leq n \implies \Theta \wedge \neg b_1\langle 1\rangle \wedge \neg b_2\langle 2\rangle}$$

Figure 3.5 lists the rules that correspond directly to the pWhile language and should be understood as follows.

- **Assignment:** The `ASSN` rule is, unsurprisingly, not differentially private. In short, it says that we can eliminate an assignment by substituting variable names in the precondition by the respective values.
- **Conditionals:** The key idea with the `COND` rule is to split and `if-then-else`-statement into two new goals: one such that the condition holds and one such that it does not. We first ensure that the conditions are equal before the subgoals are constructed. In the subgoals, we reason about the left-hand condition. As the branches are mutually exclusive, we can reason about their differentially private properties in a disjoint manner.
- **Sequencing:** Using the `SEQ` rule, we introduce an intermediate proposition. We call the proposition the cut and it is denoted Ψ'. This rule also cuts ϵ and δ into two parts which are added in the conclusion.
- **While:** The `WHILE`-rule is the most laborious of the basic rules. Using it, we can reason about a while loop. It works as the while loop rules for pRHL augmented by ϵ and δ parameters. The non-obvious components of the rules are explained as follows.
  - $e\langle 1\rangle$ represents the loop variant. This is an expression which is evaluated in the right-hand sides expression. It has to be strictly decreasing after each loop. This is evident from the pre- and postcondition in the first component of the antecedent. The second component of the antecedent assures that the loop variant indeed is a stop condition.

- k is the initial value of the loop variant in a given iteration. At the end of the iteration, the loop variant should have decreased.
- n is the initial upper bound for the loop variant.

Above rules allow us to reason about the primary language constructions. Next, we look at some structural rules.

Figure 3.6: Structural apRHL proof rules.

Conseq

$$\frac{\begin{array}{c}\vdash c_1 \sim_{\langle \epsilon', \delta' \rangle} c_2 : \Phi' \implies \Psi' \\ \vDash \Phi \to \Phi' \qquad \vDash \Psi \to \Psi' \qquad \epsilon' \leq \epsilon \qquad \delta' \leq \delta\end{array}}{\vdash c_1 \sim_{\langle \epsilon, \delta \rangle} c_2 : \Phi \implies \Psi}$$

Frame

$$\frac{\vdash c_1 \sim_{\langle \epsilon, \delta \rangle} c_2 : \Phi \implies \Psi \qquad FV(\Theta) \cap MV(c_1, c_2) = \emptyset}{\vdash c_1 \sim_{\langle \epsilon, \delta \rangle} c_2 : \Phi \wedge \Theta \implies \Psi \wedge \Theta}$$

Pw-Eq

$$\frac{\forall i. \vdash c_1 \sim_{\langle \epsilon, \delta_i \rangle} c_2 : \Phi \implies x\langle 1 \rangle = i \to x\langle 2 \rangle = i}{\vdash c_1 \sim_{\langle \epsilon, \sum_{i \in I} \delta \rangle} c_2 : \Phi \implies x\langle 1 \rangle = x\langle 2 \rangle}$$

The structural rules do not touch the term. They solely change the precondition and postcondition of a given statement. ApRHL has two structural rule, the rule of consequence and the frame rule, which are both considered normal to include.

- **Rule of consequence:** The rule of consequence allows the proof author to rewrite the precondition and postcondition along with the ϵ and δ properties. This has to be in such a way that the new conditions are implied by the original conditions and that the new ϵ and δ are less than or equal to the original ones.
- **Frame rule:** The frame rule allows adding arbitrary propositions by conjugation to the precondition and postcondition. It has to be added in such a way that they do not alter the free variables (FV) or modified variables MV.
- **Pointwise equality:** The last structural rule is pointwise equality. The rule is used to lift program variables that are pointwise equal into logical variables through i. We note that the domain of i is identical to the domain of x.

Until now we have only touched the deterministic parts of the probabilistic while language. However, the language also supports random assignment. It is with those assignments we introduce differential privacy and, as such, they are central to the apRHL proof system.

Figure 3.7: ApRHL proof rules for reasoning about the Laplace distribution.

LapNull

$$\frac{y_1 \notin FV(e_1) \qquad y_2 \notin FV(e_2)}{\vdash y_1 \xleftarrow{\$} \mathcal{L}_\epsilon(e_1) \sim_{\langle 0,0\rangle} y_2 \xleftarrow{\$} \mathcal{L}_\epsilon(e_2) : \top \implies y_1\langle 1\rangle - y_2\langle 2\rangle = e_1\langle 1\rangle - e_2\langle 2\rangle}$$

LapGen

$$\frac{}{\vdash y_1 \xleftarrow{\$} \mathcal{L}_\epsilon(e_1) \sim_{\langle k' * \epsilon,0\rangle} y_2 \xleftarrow{\$} \mathcal{L}_\epsilon(e_2) : |k + e_1\langle 1\rangle - e_2\langle 2\rangle| \leq k' \implies y_1\langle 1\rangle + k = y_2\langle 2\rangle}$$

LapInt

$$\frac{\epsilon' \triangleq \ln \dfrac{\exp(\nu\epsilon)}{1 - \exp(-\sigma\epsilon/2)} \qquad \Phi \triangleq |e\langle 1\rangle - e\langle 2\rangle| \leq k \wedge (p + k \leq r < s \leq q - k) \wedge ((q-p) - (s-r)) \leq \nu \wedge 0 < \sigma \leq (s-r) + 2}{\vdash y_1 \xleftarrow{\$} \mathcal{L}_\epsilon(e) \sim_{\langle \epsilon',0\rangle} y_2 \xleftarrow{\$} \mathcal{L}_\epsilon(e) : \Phi \implies y_1\langle 1\rangle \in [p,q] \leftrightarrow y_2\langle 2\rangle \in [r,s]}$$

Figure 3.7 lists three rules for reasoning about the Laplace assignment in apRHL. They work as follows.

- **Laplace Null:** This proof rule employs a coupling with a constant offset. Effectively it means that the assignments work identically on both sides.
- **Laplace Generate:** The `LapGen` rule is the primary rule for adding Laplace noise to an expression, e, and save the result in y. The rule takes two parameters: k and k'. These parameters can be interpreted as follows.
  - k: Coupling parameter. This parameter decides how y_1 and y_2 are coupled.
  - k': The amount of noise we use in the rule.
- **Laplace Interval:** This rule applies subset coupling between two Laplace samples

The `LapGen` and `LapNull` rules are tightly related. If we set $k = e_2\langle 2\rangle - e_1\langle 1\rangle$ and $k' = 0$ in the `LapGen` rule, we will see that we can derive the `LapNull` rule. This derivation, however, only works if $y_i \notin FV(e_i), i \in \{1,2\}$.

We now have the machinery for simple proofs in order. To support more complex proofs, a collection of proof rules has been developed for supporting various other constructions. We note that the `UTB-R` rule has been omitted as `UTB-L` is sufficiently general.

Figure 3.8: Extended apRHL proof rules.

Utb-L

$$\frac{\vDash \Phi \to \Phi_0\langle 1\rangle \qquad \vdash c \sim_{\langle \epsilon,\delta\rangle} c' : \Phi \implies \Theta\langle 1\rangle \to e\langle 1\rangle = e\langle 2\rangle \qquad \vdash_{\delta'} c : \Phi_0 \implies \Theta}{\vdash c \sim_{\langle \epsilon,\delta+\delta'\rangle} c' : \Phi \implies e\langle 1\rangle = e\langle 2\rangle}$$

Ac-While

$$\frac{\begin{array}{c}\vdash \Theta \wedge e\langle 1\rangle \leq 0 \to \neg b_1\langle 1\rangle \qquad \epsilon^* \triangleq (\sqrt{2n\ln(1/\omega)})\epsilon + n\epsilon(e^{\epsilon}-1)\\ \delta^* \triangleq n\delta + \omega \qquad \omega > 0 \qquad \vdash c_1 \sim_{\langle \epsilon,\delta\rangle} c_2 : \Theta \wedge b_1\langle 1\rangle \wedge b_2\langle 2\rangle \wedge e\langle 1\rangle = k \implies \Theta \wedge b_1\langle 1\rangle = b_2\langle 2\rangle \wedge e\langle 1\rangle < k\end{array}}{\vdash \text{while } b_1 \text{ do } c_1 \sim_{\langle \epsilon^*,\delta^*\rangle} \text{while } b_2 \text{ do } c_2 : \Theta \wedge b_1\langle 1\rangle = b_2\langle 2\rangle \wedge e\langle 1\rangle \leq n \implies \Theta \wedge \neg b_1\langle 1\rangle \wedge \neg b_2\langle 2\rangle}$$

Figure 3.8 lists a collection of proof rules for extended reasoning. Following is the description of those rules.

- **Up to bad reasoning:** This is used to employ accuracy bounds on the protocols. This can be understood such that we add the probability of a privacy violation to δ. This is in accordance with the interpretation of the δ quantity.
- **Advanced composition for while:** The `AC-WHILE` rule allows to compose a sequence of (ϵ, δ)-differentially private terms using the advanced compositions theorem. As for the `WHILE` rule, we have the loop variant in the left-hand memory denoted by $e\langle 1\rangle$. One should note that this requires the body of the loop to be homogeneously differentially private where the `WHILE` rule allows the bodies to have varying (ϵ, δ)-properties.

3.4 Concluding Remarks

Differential privacy is inherently probabilistic. The key definition considers probabilities. In this light, it is important to have a strong framework to build the algorithms that employ differential privacy. In this chapter we setup the means for rigorous formalizations of differential privacy including the following.

1. A mathematical foundation. This includes a precise definition of distributions and the machinery to reason about these distributions in an approximate relational fashion.
2. A language. The `pWhile` language is introduced to have a concrete language to express the computations we reason about. The language is an imperative language augmented by random assignment.
3. Judgments. That last part of the chapter is the development of judgments. These makes it possible to state general propositions about the algorithms in view.

The material presented in this chapter is drawn from several sources. The mathematical preliminaries on the apRHL originate from Barthe and Olmedo's work on composition theorems and relational logic for f-divergences between probabilistic programs [A6] where the pRHL judgment was developed in Barthe et al.'s work on formal certification of code-based cryptographic proofs [A4].

The language we use is the one presented by Barthe et al. [A5].

Chapter 4

EasyCrypt

EasyCrypt is a theorem prover aimed at cryptographic proofs. It supports traditional Hoare logic, three probabilistic Hoare logics, and an ambient logic to combine everything. These are classical logics to reason about higher order deterministic objects, and probabilistic programs.

The chapter progresses in several steps.

1. Firstly, the chapter introduces the EasyCrypt language. This presentation includes types, operators, and modules for expressing and organizing computations. It also includes axioms, lemmas, and proof procedures which we use to reason about properties of procedures.
2. Then, the chapter looks at the included judgments. The official EasyCrypt distribution includes four logics: the ambient logic, HL, pHL, and pRHL which are all introduced through an example. Furthermore, this thesis also includes an experimental implementation of the apRHL logic which is a part of the main distribution of EasyCrypt.
3. Finally, the chapter elaborates on proof authoring. We discuss how to develop proofs and how to implement them in EasyCrypt.

4.1 The Language

EasyCrypt provides an ecosystem for expressing probabilistic procedures and proving properties about them. This ecosystem is constituted by a fully fledged programming language and constructions for developing proofs. The first part we look into is the type system.

4.1.1 Types

EasyCrypt supports the type construction one expects from a modern strictly typed programming language. It has algebraic data types (ADTs), as known from all major functional programming languages. It also has function types, sum types, and product types (including records).

Listing 4.1: Example of types.

```
 1  (* Type aliasing *)
 2  type probability = real.
 3  (* Polymorphic types *)
 4  type 'a Maybe = [
 5      Something of 'a
 6    | Nothing].
 7  type 'a List = [
 8      (* Note, the difference between "&" and "*" *)
 9      Cons of 'a & 'a List
10    | Nil ].
```

The example in Listing 4.1 shows three cases.

1. Line 2 shows a type alias. These are useful for documentation purposes.
2. Line 4 shows a polymorphic maybe type. This case utilizes a type variable, denoted `'a` and a tagged sum type of `Something` or `Nothing`.
3. Line 7 shows the definition of a list type. As with the maybe type, we have two constructors. The `Cons` constructor takes a value of `'a` and an element of the list type. We note that using `&` is the syntax for currying the constructor. For the product type, we would have used `*`.

For further documentation on the type system, we refer to the documentation [A14].

4.1.2 Operator

Operators in EasyCrypt provides the value level expressions. They are what we traditionally see as top-level expressions in mainstream functional programming languages. Operators span pure functions and are used as literals in the procedures.

Listing 4.2: Example of operators.

```
(* Operator *)
op num : int.
(* Bound operator *)
op fortytwo : int = 42.
(* Unbound function *)
op filter : ('a -> bool) -> ('a List) -> ('a List).
(* Bound function *)
op head ['a] (xs : 'a List) : 'a Maybe =
  with xs = (Nil)        => Nothing
  with xs = (Cons y ys) => Something y.
(* Restricted operator *)
op prob : {probability | 0%r <= prob}  as prob_gt0.
```

From Listing 4.2, we see that we can have operators without accompanying binding. For many general purpose languages, this is not allowed. However, for EasyCrypt our main goal is to state and prove properties. Hence, working solely on the type of an operator and its associated axioms suffices.

The definition of `prob`, at line 12, is restricted to real numbers greater than or equal to 0. The syntax used is very akin to the syntax used by mathematicians. The construction is syntactic sugar for defining prob to be the type `probability` and defining an axiom named `prob_gt0` such that `0%r <= prob`.

4.1.3 Modules

The EasyCrypt modules consist global variables and procedures. They enclose a protocol coherently. They are later used in lemmas and axioms.

We present modules with an offset in the example in Listing 4.3 below. We describe it starting with the concrete module `M`. For that module we look at global variables, procedures, and the language which constitutes procedure, namely the probabilistic while language. We conclude this subsection with a discussion about the abstract module, or module type, which are used to model adversaries.

Listing 4.3: A simple example of an EasyCrypt module.

```
 1 module type A = {
 2   proc consume(x : int) : unit
 3 }.
 4 
 5 module M (A' : A) = {
 6   var x : int
 7   var n : int
 8 
 9   proc step() : unit = {
10     var b;
11     b <$ {0,1};
12     n <- n+1;
13     if(b){
14       x <- x+1;
15     } else {
16       x <- x-1;
17     }
18   }
19 
20   proc nSteps() : int = {
21     return n;
22   }
23 
24   proc walk() : int = {
25     var t, ret;
26     t <$ {0,1};
27     while(t){
28       t <$ {0,1};
29       step();
30       A'.consume(x);
31       ret <@ nSteps();
32     }
33     return ret;
34   }
35 }.
```

The procedure, `step`, is declared on line 9 in Listing 4.3. The declaration resembles that of most imperative languages like Java and C. As input, the procedure takes a tuple of arbitrary size, and as output, it returns a value of any type. The `step` procedure returns `unit` and, as such, it does not need any return statement.

Above, the `step` procedure two global variables are declared. A global variable defines a state that is mutable from within all procedures of the module. This resembles a field variable in object oriented languages.

The implementation of the probabilistic while language is also shown in Listing 4.3. There are three variations of assignment: deterministic assignment (denoted `<-` with an example on line 12), random assignment (denoted `<$` with an example on line 11), and assignment from procedure calls (denoted `<@` with an example on line 31).

Furthermore, there the example also shows conditional (the `if`-statement), iteration (the `while`-statement) as imperatives. The last two constructions are **`var`** and **`return`**. **`var`** is used to declare variables with either explicit type or inferred type. The **`return`** statement discriminates the result of the procedure and has to be the last statement.

The last construction presented in the example is the **`module type`**. A module type declares a module without any implementation. In EasyCrypt this construction is used for both adversaries and oracles. In the above example, we declare that the module `M` can access `A'` which is an instance of `A` on line 5. Lastly, it is possible to provide an implementation to a module type by the construction `module MA : A = {...}`, which is not a part of the example.

4.1.4 Axioms and Lemmas

Axioms and lemmas are stated equivalently. The only difference is lemmas are shipped with a proof procedure where the axioms are not.

Listing 4.4: Examples of stating axioms.

```
axiom a1 :
  forall (a b : bool), a = b => b = a.
axiom a2 (a b : bool) :
  a = b => b = a.
axiom a3 :
  forall (a : bool) (b : bool), a = b => b = a.
axiom a4 (a : bool) (b : bool) :
  a = b => b = a.
axiom a5 :
  forall (a b : bool), a = b =>
    forall (c : bool), b = c => a = c.
```

Listing 4.4 shows five different definitions of axioms. The difference is in the syntax for variable quantification. Quantification is done in two ways: as parameters of the axiom or using the `forall` keyword in the body of the axiom.

The lemmas from Listing 4.5 are solved using only the **`auto`** tactics. The tactics are introduced below and further discussed in section 4.3.

Listing 4.5: Examples of stating lemmas.

```
lemma l1 :
  forall (a b : bool), a = b => b = a.
proof.
  auto.
qed.
lemma l2 :
  forall (a b : bool), a = b =>
    forall (c : bool), b = c => a = c by auto.
```

Listing 4.5 shows two examples of lemmas. They differ in the way a proof procedure is supplied. In the first example, the proof is explicitly outlined between the `proof` keyword and the `qed` keyword. The next example has its procedure in-lined by using the `by ...` construction. This construction is suitable for very small proofs, as it takes up less space.

Proof Procedure

The proof procedure is a list of tactics and tacticals denoted τ. Together they create a deduction tree proving a statement by the core rules of the theorem prover. Before introducing tactics and tacticals, we need to understand a subtlety about contexts.

Evidently, EasyCrypt works on several different logics. Each of these logics employs tactics that overlap with the other logics. For this, it is important to note the *context* of the current goal. For some of the tactics, the context decides how the tactics behave.

In the presentation of the lemmas, the proofs were closed using the `auto` tactic. In the next section, we are using a number of tactics. Following is a short explanation of the tactics we use.

- `auto.` Attempts to reduce the goal to a simpler one using various rules.
- `proc.` Unfolds a procedure call substituting the call for the procedures content.
- `progress.` Attempt to make the goal simpler possibly creating more goals.
- `rnd.` Reduces random assignments. The concrete behavior depends on the context.
- `skip.` Reduces an empty Hoare logic statement to an ambient statement.
- `smt.` Solves current goal using an SMT solver.

We will also see a number of tacticals in the coming sections and chapters. Initially, we mostly used is the sequencing tactical. However, there are also a number of tacticals we need to understand.

- τ_1 ;τ_2. Apply τ_1 to current goal and apply τ_2 to *all* generated subgoals.
- `by` τ. Applies τ to current goal and fails if not all new subgoals are closed by trivial.
- `try` τ. makes τ fail silently.

With this short presentation, it should be possible to understand the examples in the next section.

4.2 The Logics

EasyCrypt supports a number of judgments for probabilistic reasoning. In this section, we will walk though the judgments, discuss their syntax, and attempt to get acquainted with them.

To present the judgment, we take offset in an example. Listing 4.6 contains a module, `Coins`, with three procedures. They constitute an unbiased coin, a biased coin, and a function that simply flips a boolean value.

Listing 4.6: Module for further examples

```
module Coins = {
  proc unbiased() : bool = {
    var sample;
    sample <$ {0,1};
    return sample;
  }

  proc biased(p : real) : bool = {
    var sample;
    sample <$ Biased.dbiased p;
    return sample;
  }

  proc flip(b : bool) : bool = {
    return !b;
  }

  proc flipNT(b : bool) : bool = {
    while(true){}
    return !b;
  }
}.
```

The example will be used for the probabilistic Hoare logics. Before we get to them, we present the ambient logic. It is used to bind probabilistic statements together, and it is used within those logics as preconditions and postconditions.

4.2.1 Ambient Logic

EasyCrypt uses an ambient logic to connect the other judgment introduced later on. It does this through the usual logical constructors such as existential and universal quantification. We have types such as booleans, integers, and reals. Lastly, EasyCrypt also supports the usual operations such as *and* (/\\), *or* (\\/), *equality* (=).

Listing 4.7: Module for further examples.

```
lemma ambientDemo (a : bool) (b : bool) :
    (a /\ b) => (b /\ a).
proof.
  smt.
qed.
```

The example in Listing 4.7 states commutativity of the boolean *and* operator. EasyCrypt proves it through the integrated SMT solver.

4.2.2 Hoare

EasyCrypt supports Hoare Logic judgments. They are expressed in the form `hoare[M.p :Ψ ==>  Φ]` where Ψ is the precondition and Φ the postcondition. `M.p` denotes the procedure `p` from the module `M`.

As usual, a Hoare logic judgment indicates that if a procedure terminates on a memory that satisfied the precondition, then the procedure terminates with a value satisfying the postcondition.

Listing 4.8: Two Hoare judgments

```
lemma hlDemo (a : bool):
    hoare[Coins.flip : b = a ==> res = !a]
by proc; auto.

lemma hlDemoNT (a : bool):
    hoare[Coins.flipNT : b = a ==> res = !a].
proof.
  proc; auto.
  while (true); skip => //.
qed.
```

Listing 4.8 contains two lemmas. The first is for a terminating statement while the second is for an non-terminating statement. We emphasize following two points:

1. In the first lemma, we do not write `hoare[Coins.flip(a) : true ==> res = !a]`. Instead, we refer the argument to the procedure in the precondition.
2. The variables `a` and `b` are bound in two different contexts: `a` is bound to the logical expression while `b` is a part of the procedure. We call a an *ambient variable* and b a *program variable*.

Both of these points are carried on to all further judgments. Further judgments use the same structure as they are all variations of Hoare logics.

The next judgment we look at is the Hoare logic augmented by probabilistic bounds to reason about probabilistic properties of procedures.

4.2.3 PHoare

The probabilistic Hoare logic is akin to the HL judgment previously discussed. The main difference is that we quantify the change of termination. Previously we stated that *if* a procedure terminated it would satisfy the postcondition. This judgment states the same with the addition of a bound or a value for the probability of termination.

Listing 4.9: Three examples of the pHL judgment

```
lemma phlDemoEq :
    phoare[Coins.biased :
      p = (1%r/2%r) ==> res] = (1%r/2%r)
by proc;rnd;skip;smt.

lemma phlDemoUb :
    phoare[Coins.biased :
      p = (1%r/3%r) ==> res] <= (1%r/2%r)
by proc;rnd;skip;smt.

lemma phlDemoLb :
    phoare[Coins.biased :
      p = (2%r/3%r) ==> res] >= (1%r/2%r)
by proc;rnd;skip;smt.
```

Listing 4.9 shows the three variations for quantifying probability for termination. They are read as follows.

1. Given the procedure of a biased coin from Listing 4.6 with the parameter p instantiated to $\frac{1}{2}$, the output of the procedure is *true* with the probability $\frac{1}{2}$.
2. Given the procedure of a biased coin from Listing 4.6 with the parameter p instantiated to $\frac{1}{3}$, the output of the procedure is *true* with an upper bound on the probability of $\frac{1}{2}$.
3. Given the procedure of a biased coin from Listing 4.6 with the parameter p instantiated to $\frac{2}{3}$, the output of the procedure is *true* with a lower bound on the probability of $\frac{1}{2}$.

EasyCrypt provides a syntax for writing probabilistic expressions on the form `Pr`$[event] = p$. This syntax is coupled such that we can rewrite it into a `phoare` judgment by the **`byphoare.`** tactic.

As a final remark, it should be noted that the Boolean type used in the procedure is the same as the boolean type used in the logic. All three examples from Listing 4.9 reason directly about the return value of the procedure rather than comparing it to a value. I.e. we use `phoare[··· ==> res] ···` rather than `phoare[··· ==> res = true] ···`.

This judgment considers a single procedure. For our purposes, we want to be able to relate procedures. In chapter 3 we discussed the pRHL judgment. Next, we discuss the corresponding judgment in EasyCrypt.

4.2.4 Equiv

To reason about the exact relation of procedures, we use the `equiv` judgment. The judgment is based on the pRHL logic.

Listing 4.10: The pRHL judgment

```
lemma prhlDemo :
    equiv[Coins.unbiased ~ Coins.unbiased :
      true ==> ={res}].
proof.
  proc; rnd; progress.
qed.
```

Listing 4.10 shows an example of the judgment. We show that the procedure of an unbiased coin has the same distribution as itself.

We aim for differential privacy as the main goal for this thesis. To get there we need another ingredient: a notion of statistical distance conforming to the definition of differential privacy (definition 2.1). This is what the approximate pRHL solves.

4.2.5 Aequiv

The approximate probabilistic relational Hoare logic (apRHL) is a logic originally proposed for differential privacy, why it also resembles its definition closely. EasyCrypt uses the keyword **aequiv** (as in approximately equivalent), and judgments have the following form.

Listing 4.11: The apRHL judgment

```
op eps  : {real | 0%r < eps}  as eps_gt0.
op delt : {real | 0%r < delt} as delt_gt0.

lemma eps_ge0 : 0%r <= eps by smt.
lemma delt_ge0 : 0%r <= delt by smt.

lemma aprhlDemo :
  aequiv[[eps & delt]
    Coins.unbiased ~ Coins.unbiased :
      true ==> true].
proof.
  toequiv.
  smt (eps_ge0).
  smt (delt_ge0).
  proc; rnd; progress.
qed.
```

At the beginning of the example in Listing 4.11, we introduce two operators (or variables): **eps** and **delt**. They have their domains restricted to only includes positive reals. Next, we derive that they are greater than or equal to 0. This is more general than their axioms. These lemmas are simply derived using the **smt** tactic.

The main judgment is presented in the lemma. The Judgment is closely related to the `equiv` construction with the difference that we provide an ϵ and a δ through a product (`[eps & delt]`) in the beginning of the judgment.

In the example, we proof the same as in Listing 4.10. as the `aequiv` construction is more general than the `equiv` construction, we simply turn it into the `equiv` construction using the `toequiv` tactics.

This judgment is not in the main distribution of EasyCrypt. The tactics for the judgment is mostly documented in the papers on apRHL, in examples using the `aequiv` judgment and in the EasyCrypt source code. The used tactics are documented in the appendix. Furthermore, we present a list of tactics and proof rules along with their relation to the apRHL rules.

Tactics and Proof Rules

To make an overview, we here present a list of `aequiv` tactics compared to their corresponding proof rules. The proof rules can be found in figure 3.5 and figure 3.7.

- **Sequencing - `seq c1 c2 :`Φ `<[eps1 & delt1]>`:** The proof rule `SEQ` is implemented in EasyCrypt by the same identifier as for other logics. Along with the left and right cuts, the tactic also takes a tuple, `<[eps1 & delt1]>`, deciding how much privacy budget goes to the first expression of the cut. These values are subtracted from the second expression of the cut. It corresponds to the proof rule such that, given `eps0` and `delt0` to be respectively ϵ and δ for the current goal
- **While - `awhile [ef & df] n [v] inv`:** The `awhile` tactic corresponds to the `WHILE` proof rule in figure 3.5. It is used for sequentially composing the bodies of the while loop. The components should be interpreted as follows
  - **ef:** A function, int $\rightarrow$ real that returns ϵ for the k's index of the while loop. This corresponds to ϵ_k in the summation of ϵ's in the conclusion of the rule.
  - **df:** A function, int $\rightarrow$ real that returns δ for the k's index of the while loop. This corresponds to δ_k in the summation of δ's in the conclusion of the rule.
  - **n:** Maximum number of iterations. This component corresponding to n from the rule.
  - **v:** The loop variant expression. This corresponds to $e\langle 1\rangle$.
  - **inv:** The invariant. This corresponds to Θ in the rule.
- **Laplace Generation - `lap k k'`:** This rule corresponds to `LAPGEN`, such that k' and k corresponds to the variables of the same name in the rule. The generated goals correspond to proving that the precondition indeed implies to the postcondition in the rule.

The structural and extended `aequiv` tactics from figure 3.8 and figure 3.6.

- **Pointwise Equality - `pw-eq`:** The pointwise equality
- **While - `awhile`:** The while rule

Furthermore, there are a couple of tactics which are not reflected in a proof rule.

- `toequiv`: Translates current `aequiv` goal into an `equiv` goal. This is done by proving that `eps` and `delt` parameters are 0 and then remove them.
- `ofequiv`: Translates an `equiv` goal into an `aequiv` goal by introducing `eps` and `delt` set to 0.

In the appendix, there is a full reference on the `aequiv` tactics we use.

4.3 Proof Authoring

In the previous section, we present the syntax of EasyCrypt. Next, we delve into the proof authoring process.

4.3.1 Proof Development

Before implementing the proof in a theorem prover, we need to have a plausible idea about the proof. There are several ways to get ahead from a given point int authoring. However, going forward the wrong way can lead to a dead end.

The following is a structure which has showed to work in practice.

1. **Rough Paper Proof:** Develop the proof on paper using the logical framework available in the theorem prover. In our case, we developed the rough paper proofs using the apRHL rules outlined in the papers.
2. **Procedure implementation:** Before we can prove anything we need the procedure in question. We allow ourself to rewrite the procedure such that it is easier to prove a property. To ensure equivalence to the original procedure, we often implement the original procedure also, and state equivalence. This reinterpretation of the proof often leads to new assumptions, so we need to consult the paper proof again.
3. **Invariants:** The invariants is constituted by a number of predicates and operators. To make sure these predicates are correct, one can implement a number of sanity lemmas and auxiliary lemmas. In the sanity lemmas, one simply instantiates the predicates with values that should conform to it, and prove that it does. The auxiliary lemmas are more important later in the proof development: they help make it easier to prove later lemmas.
4. **The Proof:** It this stage the author should attempt to implement the proof. When stuck, it is helpful to go back in the process and consult the paper proofs, the procedure, and the invariants.

It is important to understand that above structure is not a step-by-step procedure. When the author gets to write the invariants of the proof, she might learn something new and need to go back to the paper proof with the updated assumptions.

4.3.2 Conventions

Previously we discussed the overall proof development and implementation in EasyCrypt. However, programming and proving are by large an exercise of communication. To do this effectively, we conform to conventions.

File Layout EasyCrypt source files typically include four key chunks of code. These includes types, operations, modules, and the proofs/axioms. To make it easy to navigate in the source files, the authors of the code should organize the code chunks in the following way.

1. Types
2. Operations
3. Modules
4. Axioms and lemmas

Tactic Scoping When the proof author uses a tactic, this will often result in a number of subgoals that. To organize the tacticals used for the subgoals of a proof we can list them in such a way that auxiliary-like subgoals get a bullet. This organization indicates the origin of subgoals.

4.3.3 Tactics

Previously in this chapter, we laid out the tactics: `auto`, `proc`, `progress`, `rnd`, `skip`, `smt` along with the tactics discussed in the apRHL subsection. Following extends this list.

- `admit.` admits current goal.
- `wp.` Weakest precondition
- `sp.` Strongest postcondition
- `move.` Works as the identity tactics. It is used in conjunction with an introduction pattern such as `-> H`, `-> //` that respectively introduces the hypothesis H and applies trivial.

For a complete reference for the EasyCrypt theorem prover, we refer to the manual [A14].

4.4 Concluding Remarks

A large part of this chapter is based on the EasyCrypt reference manual [A14]. The part that was not documented was obtained in other ways including following.

- Answers to questions The EasyCrypt team has kindly answered questions.
- The source code to EasyCrypt.
- Examples from the paper on the between thresholds protocol [A1].

Part II

Case Studies

Chapter 5

Randomized Response

As a warm up, we formalize the proof that the randomized response protocol is $\lg 3$ differentially private. The proof is simple but allows us to look into proving theorems in EasyCrypt.

The randomized response protocol works on a per participant basis. For the participant, it provides plausible deniability. This property is important in many applications. One such application could be an authority that wants to gather statistics about embarrassing or criminal activities. For this database, we cannot allow the analyzing authority to list incriminated participants. Therefore participants follow this procedure.

1. Flip an honest coin.
2. If the coin shows tail answer the question truthfully.
3. If it shows head flip a new honest coin. Answer yes if it shows tail and no for head.

If a large set of participants follow this protocol, we have a set of answers which is a product of the real answers and a uniform distribution over coin flips. From the knowledge about the coupling, we can derive the distribution of the truthful answers.

Lemma 5.1: Randomized response projection

Let C_1 and C_2 be two honest coins, $result$ is participant answers and $truth$ is the true participant answer. $Pr[C_i]$ is shorthand for $Pr[C_i = tail]$, $Pr[result]$ for $Pr[result = yes]$ and $Pr[truth]$ for $Pr[truth = yes]$.

$$\begin{aligned} Pr[result] &= Pr[C_1] * Pr[truth] + (1 - Pr[C_1]) * Pr[C_2] && (5.1) \\ &= \frac{1}{2} * Pr[truth] + \frac{1}{4} && (5.2) \end{aligned}$$

From here we can derive the probability of the *truth* being "yes".

Before delving into the actual formalization, we take a look at the problem in terms of differential privacy. Differential privacy uses the notion of a database that is usually known as a set or a table of tuples. For the randomized response protocol, however, the database is simply the *single* boolean value provided to the protocol. A database as simple as that greatly simplifies the whole notion of distance between databases as the universe of databases only span two elements (*true* and *false*). Thus, defining the adjacency relation to include values that differ by one includes all possible pairs of *true*/*false* values.

5.1 A Proof

To get acquainted with proofs of differential privacy, we look at a simple semantic proof.

Theorem 5.1: Randomized Response is $\ln(3) - DP$

The randomized response query is $(\ln 3, 0)$-differentially private.

Proof. The proof builds on following truth table.

Secret	C1	C2	Answer(Secret)
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	T
F	F	F	F

Note that $Pr[Answer(T) = T] = \frac{3}{4}$ and $Pr[Answer(F) = T] = \frac{1}{4}$.

Fix the protocol output to T (the proof is the same when we fix it to F) and recall the definition of differential privacy. Following are the four cases we consider.

$$Pr[Answer(T) = T] \leq \exp(\epsilon) Pr[Answer(T) = T] + \delta \tag{5.3}$$
$$Pr[Answer(T) = T] \leq \exp(\epsilon) Pr[Answer(F) = T] + \delta \tag{5.4}$$
$$Pr[Answer(F) = T] \leq \exp(\epsilon) Pr[Answer(T) = T] + \delta \tag{5.5}$$
$$Pr[Answer(F) = T] \leq \exp(\epsilon) Pr[Answer(F) = T] + \delta \tag{5.6}$$
$$\tag{5.7}$$

Disregarding the $\exp(\epsilon)$ and δ components, the only equation that violates the inequality is the the second as it states $\frac{3}{4} \leq \frac{1}{4}$. To solve this we set $\epsilon = \ln(3)$ and $\delta = 0$. This yields $\frac{3}{4} \leq 3 * \frac{1}{4} + 0$ which is true. We also note that this has no impact on the other inequalities.

■

The rest of the chapter considers the implementation of this proof in EasyCrypt.

5.2 Formalizing the Procedure

To be able to prove anything about the randomized response protocol, we need an implementation. The properties are then shown for this instance of the protocol. For that reason, it is also important that the implementation is indeed the one we want. For this example, it is straightforward to verify that it does what we want. The whole algorithm is fairly simple and, as such, we believe its correctness. In other circumstances, we might want to implement a suite of tests to convince us that the implementation performs as expected.

In EasyCrypt, we use the module system and make a `sample` procedure that takes the truth as input and emits either true or false. The following snippet shows our implementation.

Listing 5.1: The randomized response protocol.

```
module RR = {
  proc sample (sec : bool) : bool = {
    var t, t', returnValue : bool;
    t <$ {0,1};
    if (t) {
      returnValue = sec;
    }
    else {
      t' <$ {0,1};
      returnValue = t';
    }
    return returnValue;
  }
}.
```

As mentioned in the introduction to this chapter, the database for this protocol is simply a single value of either *true* or *false*. Hence the input to the sampling procedure is a boolean value. It is now also evident that we cannot possibly do more than a single operation on the database: we can either do nothing or flip the boolean value.

Having defined the procedure the next step is to start the actual proof procedure.

5.3 Proof Preliminaries

Before doing the actual proof of differential privacy, we need some preliminaries to implement the final proof.

The first task is to show that our procedure is lossless. The lossless property simply states that the procedure always terminates. The termination property is necessary because our final proof spans the complete domain and co-domain of the procedure.

For this program, the lossless property does not come as a surprise. It is straightforward to verify termination for all inputs and random assignments for the procedure.

Next, we want to make it easier to prove properties about the main procedure in Listing 5.1. The rewritten procedure, in Listing 5.2, collects all random assignments in the top part followed by all deterministic operations. This construction is structurally simpler and allows for easily splitting up the proof in a probabilistic part and a deterministic part. Later, we will see how this helps to prove properties.

Listing 5.2: A semantically equivalent procedure to RR.sample

```
module RR' = {
  proc sample (sec : bool) : bool = {
    var t, t', returnValue : bool;
    t  <$ {0,1};
    t' <$ {0,1};

    if (t) {
      returnValue = sec;
    }
    else {
      returnValue = t';
    }
    return returnValue;
  }
}.
```

In the end, properties proven about the procedure in Listing 5.2 should also apply to the procedure in Listing 5.1. This is, however, not immediately possible even though intuitively it is easy to see that the procedures are equivalent. We know that the `{0,1}` distributions are independent and, as such, it does not make any difference to draw from it twice.

We need to justify for EasyCrypt the equality of the two distributions. This justification is done by a proof of equivalence of `RR.sample` and `RR'.sample` as shown in Listing 5.3.

Listing 5.3: *RR.sample* **and** *RR′.sample* **are equivalent.**

```
lemma rr_and_rr'_are_equiv :
  (* res{1} = res{2} is the same as ={res} *)
  equiv[RR.sample ~ RR'.sample : ={sec} ==> ={res}].
proof.
  proc.
  seq 1 1 : (={sec, t}) => //.
    + by rnd; skip; smt.
  if{1}.
  + by auto; smt.
  + seq 1 1 : (={sec, t, t'} /\ t{2} = false) => //.
    * auto; smt().
    * auto.
qed.
```

The machinery for the proof that the procedure from Listing 5.1 is lg 3-differentially private is now in order. The next section will present the proof.

5.4 Proving Differential Privacy

The last thing we need to do is the actual proof that the randomized response protocol is $(\lg(3), 0)$-differentially private. Given the work we already did, the task is quite simple. The following code snippet states the proof and includes the proof procedure.

Listing 5.4: The proof that the randomized response protocol is $\lg 3 - DP$

```
lemma RRIsEpsilonLg3DP &m (x, y : bool):
  Pr[RR.sample(x) @ &m: res]
    <= 3%r * Pr[RR.sample(y) @ &m: res] by smt.
```

First, note that we multiply the right-hand probability by 3, which does not exactly match the definition of differential privacy. This is because ϵ is $\lg 3$. the lg function and the exp function are the inverse of each others and are canceled out.

Next, the proof is constituted only by a `smt.` tactic. We should take into account that the tactic uses everything in the current scope. This also includes all previous and included lemmas which extends further than the ones included in this chapter. In particular, the source file holds a number of proofs on `RR.sample` alone. These proofs corresponds to the properties we read of the truth table in the initial proof of this chapter. Because of that the proof is quickly proven and it can simply be handed over to the SMT solver. But this is only due to the prior work in the proof procedure and is not self-contained.

5.5 Concluding Remarks

We have walked through much of the workflow on implementing a random procedure and showing that it is differentially private. This includes

1. Formalizing the procedure from an informal description of the protocol.
2. Proving the lossless property.
3. Implementing auxiliary procedures simplifying further proofs.
4. Showing equivalence to the initial procedure.
5. Proving the differential privacy property.

From here the procedure is ready to be used in an actual application.

The proof presented in the case study relied on the pRHL and yields a semantic proof. The next three case studies are implemented in the apRHL and yields logical proofs.

Chapter 6

Laplace Mechanism

This chapter presents the Laplace mechanism through simple examples. We will show that numeric values and lists over numeric values can be made differentially private by adding Laplace noise. As the Laplace mechanism is a built-in construction in apRHl, the work is not overwhelming. However, this case study provides the foundations for the last two case studies, and we will build upon the understanding developed in this chapter.

The chapter progresses as follows.

- First, we elaborate on preliminaries. This presentation includes the database shape and assumptions we use for the Laplace mechanism.
- Then, we present the Laplace mechanism in terms of numeric values.
- Lastly, we juxtapose the Laplace mechanism with the while construction to prove differential privacy for lists with added Laplace noise.

6.1 Preliminaries

This section introduces the assumptions we use to reason about differential privacy using the Laplace rule.

6.1.1 Databases

The two applications we consider in this chapter use two different notions of databases. For the Laplace mechanism, we only consider integers. In the list example, we use a list of integers.

6.1.2 Ambient Parameters

We choose to define some of the parameters as operators. This choice greatly simplifies the proof development as we need them in the ambient logic.

Listing 6.1: Ambient parameters.

```
op N : { int | 0 < N } as gt0_N.
op eps : {real | 0%r < eps} as gt0_eps.
op eps_i : real = (eps/(N+1)%r).
```

Listing 6.1 shows the parameters defined as operators. The `N` operator is the size of the list. As we will see later on, we constrain it to the exact size of the lists we consider.

Next, we define `eps` and `eps_i`. They define the ϵ parameter for the proof respectively for the whole protocol (for both proofs) and each iteration of the loop in the proof considering lists.

6.1.3 Adjacency

In the earlier case study, there was no need to formalize distance between databases. This was a result of a database being a boolean value.

For this case study, we use two notions of databases: one for a single numeric value and one for lists of numeric values.

Listing 6.2: The adjacency relations.

```
pred adjV (v1 v2 : int) = `|v1 - v2| <= 1.

pred adjL (l1 l2 : int list) =
        size l1 = size l2
    /\  forall i, 0 <= i < size l1 =>
          `|(nth 0 l1 i) - (nth 0 l2 i)| <= 1.
```

Listing 6.2 shows two predicates that constitute adjacency relations. The first, `adjV`, is for values. Evidently, we allow them to differ by a maximum of 1. The second, `adjL`, is for lists. Using this predicate, we require that two adjacent lists have the same length and that all values with corresponding indices differ by at most one. As we will see in the next chapter, this is not the only adjacency relation for lists in use.

6.2 Laplace and Numeric Values

The simplest case of an algorithm we can imagine in the apRHL logic considers a single numeric value and applies Laplace noise to it. Then it is released in a differentially private manner.

Listing 6.3: Laplace for numeric values.

```
1 proc val(x : int) : int = {
2   var s;
3   s <$ lap eps x;
4   return s;
5 }
```

Listing 6.3 shows the protocol. Four out of the five lines are necessary to define a procedure considering a single variable. The important stuff happens on line 3, where we sample an integer value from the Laplace distribution centered at `x` with `eps` spread.

Listing 6.4: Proof of differential privacy for Laplace on numeric values.

```
lemma lap_value_dp :
  aequiv[[eps & 0%r]
    Lap.val ~ Lap.val : adjV x{1} x{2} ==> ={res}]
by proc; lap 0 1.
```

To prove differential privacy we couple the distribution such that they have no difference for $1 * \epsilon$. Listing 6.4 shows the implementation of the proof. Beside unfolding the procedure, it only uses the `lap` tactic.

To understand the application of the `lap` tactic we try to unfold the proof rule from Figure 3.7. In the application $k = 0$ and $k' = 1$. If we substitute the variables in the precondition and postcondition of the rule we have $|0 + x\langle 1\rangle - x\langle 2\rangle| \leq 1 \implies s\langle 1\rangle + 0 = s\langle 2\rangle$. Per assumption (the adjacency relation) we know that the difference on x is no greater than 1 and this trivially holds.

6.3 Laplace and Sequential Composition

Before going to Laplace on lists, we consider a single composition of two Laplace samplings. The procedure is straightforward; we sample twice and return both samples in a tuple.

Listing 6.5: Procedure using sequential composition.

```
proc seq_comp(x : int, y : int) : (int * int) = {
  var s1, s2;
  s1 <$ lap (eps/2%r) x;
  s2 <$ lap (eps/2%r) y;
  return (s1, s2);
}
```

The procedure in Listing 6.5 is all we need. We see that each Laplace sampling only receives half `eps`. This is important for the proof as we divide the privacy cost.

The proof is neither complicated. We use the sequencing rule and remember to save half the amount of privacy through the cut.

Listing 6.6: Proof of for sequential composition.

```
lemma lap_seq_comp :
  aequiv[[eps & 0%r]
    Lap.seq_comp ~ Lap.seq_comp :
      adjV x{1} x{2} /\ adjV y{1} y{2}  ==> ={res}].
proof.
  proc => //=.
  seq 1 1 : (adjV y{1} y{2} /\ ={s1})
    <[(eps/2%r) & 0%r]>.
  * lap 0 1.
  * lap 0 1.
  smt.
qed.
```

In the proof in Listing 6.6, we note the construction at the end of the `seq` tactic. `<[(eps/2%r) & 0%r]>` denotes how much privacy we allow before the cut. These amounts are subtracted when entering after the cut.

Each use of the `lap` tactic corresponds to before and after the cut.

6.4 Laplace and Lists

The last protocol applies Laplace noise to a list of numeric values.

Listing 6.7: Laplace for lists over numeric values.

```
proc list(a : int list) : int list = {
  var rs, i, x, s;
  i <- 0;rs <- [];
  while(i < N){
    x <- nth 0 a i;
    s <$ lap eps_i x;
    rs <- rs ++ [s];
    i <- i+1;
  }
  return rs;
}
```

Listing 6.7 contains the implementation of the protocol. It is a while loop iterating a list sampling the Laplace distribution as in above protocol and appending the result to an output list.

The proof conceptually has six parts: initialization, rewriting the ϵ parameter, the while loop, initialization of an iteration, the Laplace sampling in an iteration, and the last deterministic part of the iteration. However, all parts are not equally interesting, so we only focus on the rewriting of ϵ and the while loop.

Listing 6.8: While principle for apRHL.

```
conseq <[(((N+1)%r)*eps_i) & 0%r]>.
+ smt.
```

Listing 6.8 uses the rule of consequence to rewrite the `eps` parameter into `(N+1)*eps_i`. The truth is a direct consequence of the definition of `eps_i`. This rewrite prepares for proving the goal considering the while loop.

Listing 6.9: While principle for apRHL.

```
awhile      [(fun _ => eps_i) & (fun _ => 0%r)]
            (N+1)
            [N - i]
            (    ={i, rs}
              /\ 0 <= i{1}
              /\  adjL a{1} a{2});
  1..4: smt (gt0_eps gt0_N).
+ by rewrite sumr_const; smt.
+ by smt.
move => k0.
```

The last goal we consider for this proof is the while loop. Listing 6.9 shows the employed tactical to reduce the goal. The subgoals we consider are the following.

1. `0 <= N+1` The number of iterations are greater than or equal to 0.
2. `0 <= eps_i`. The function defining ϵ for loop-variant-index is greater than or equal to 0. This tactic will also appear for `delt` if the case is not trivial.
3. Stop condition: If the invariant is true and the loop variant is less than 0, then the loop condition is false.
4. The precondition before the loop implies the invariant, that the loop variant behaves identically on both sides, and that the initial loop variant is less than the upper bound.
5. The sum over all `eps_i` is `eps`.
6. The sum over all `delt_i` is `delt`.

Lastly, we note the `move => k0` tactical in the bottom. The `awhile` tactic provides a parameter `k` that denotes the loop variant for the current iteration. This is a unique value (as a loop variant is strictly decreasing) and can be used to identify a given iteration if we reason about heterogeneous privacy cost. However, in this example, we use the same amount of privacy in each iteration, and we disregard the `k` parameter.

Chapter 7

Sums Over Streams

This chapter looks into the sums over a streams application [A5]. When taking the sums over streams, we want to be able to release the sum at any given point in the stream. To do this, the protocol aggregates a sum from the stream. The goal is to make sure that the released answers are differentially private.

The chapter progresses as follows.

- First, the chapter introduces the protocols and the approach for the proof.
- Secondly, the chapter elaborates on the preliminaries of the proof. This includes handling variables, auxiliary operations, and the adjacency relation we proof differential privacy under. This section also elaborates on auxiliary lemmas
- Lastly, the chapter elaborates on the proofs of differential privacy.

7.1 Sum over Streams

We first look into three ways to solve the problem. All the protocols seek to release a sum of the i'th element.

The first technique sums the i'th first elements and then adds Laplace noise to the sum. The privacy cost for each query is ϵ. Hence, the cost of releasing n elements is $n * \epsilon$. This approach delivers a quite good accuracy as the result only get noise once, but the privacy cost is large.

The second technique adds Laplace noise to all elements first. Then it sums the i'th first elements elements. This yields a much better privacy cost as we get n results for ϵ cost. However, the accuracy is also worse as we add noise to all elements.

The last technique is to employ previously mentioned techniques such that we split a list into q sublists. For each of these sublists, we add Laplace noise according to the second technique. Then this list is summed.

Before going in depth with the implementation, it should be mentioned that the implementation solely works on finite lists. Currently, we can only reason about terminating structures. For this, lists are sufficiently general.

7.2 Proof Preliminaries

For the actual proofs, we lay out some preliminaries.

7.2.1 Ambient Properties

We need to reason about some of the variables of the procedure in the ambient logic. If a variable is not modified by the procedure, then we define them in terms of operators. For this case study we define the lengths of lists and the ϵ-parameters as operators.

7.2.2 Auxiliary Operators

Besides the operators we use to define constants, we also introduce some auxiliary functionality. This is done to limit the code in the procedure. In turn, this makes it easier to reason about the properties as there are fewer constructions to prove.

Furthermore, we seek to implement these operators in terms of existing, library-supported operators. This approach makes is considerable easier to prove properties about them.

Lastly, each operator has sanity checks which resemble simple software test. They also have auxiliary lemmas to ease further reasoning.

Remv To reason about the difference of a single elements with same index in two lists, we assert that the lists should be identical, if we remove the exact element that differs.

To ease that we implemented the `remv` operator.

Listing 7.1: The remove element operator

```
op remv ['a] (n : int) (xs : 'a list) =
  take n xs ++ drop (n+1) xs.
```

Sum The `sum` identifier is an abbreviation for the `big` operator of the `BigZModule` module.

Prior to this solution we implemented the sum-functionality both through a recursive operator and in terms of the `foldl` operator. However, we need to use it in a context, and in particular we need to prove properties depending on the `sum` operator. The most predominant property, `sum l = (sum (remv t l)) + (nth 0 l t)`, explicitly needs a lemma from the BigZModule library.

Listing 7.2: Lemma: remove and sum

```
lemma remv_sum (l : int list) (t : int) :
  0 <= t < size l =>
    sum l = (sum (remv t l)) + (nth 0 l t)
      by smt (remv_concat BigInt.big_cat sum_add).
```

Listing 7.2 shows the lemma in question. The SMT-solver receives three lemmas: `remv_concat`, `BigInt.big_cat`, and `sum_add`. Removing any of these lemmas result in a timeout from the solver.

Offset Copy The offset copy operator is needed for the smart sum protocol.

Listing 7.3: The offset copy operator

```
op offsetCopy (s : int list, x : int list
  , c : int, i : int, q : int) : int list =
     take i s
  ++ map ( (+)  c) (take q x)
  ++ drop (i+q) s.
```

The implementation of the offset copy operator in Listing 7.3 replaces q elements from index i in the s list, with the first q elements from the x list and adds a constant c to each replaced element.

7.2.3 Database Adjacency

In the spirit of differential privacy, we assume that the databases are adjacent over a relation. The relation is setup such that the input lists differ on at most 1 element. In EasyCrypt this precondition is formulated as the Listing below.

Listing 7.4: Precondition for sums over stream.

```
pred adj (l1 l2 : int list)  =
       (size l1 = size l2
    /\ (exists t, 0 <= t < size l1 /\
           remv t l1 = remv t l2)
    /\ (forall i, 0 <= i < size l1 =>
          `|(nth 0 l1 i) - (nth 0 l2 i)| <= 1)).
```

This predicate in Listing 7.4 defines an adjacency relation by following 3 rules:

1. The lists should have equivalent length
2. There exists a single element we can remove to make the lists completely identical.
3. Corresponding values (The value from each list with same index) is less than or equal to 1.

These three rules formalize the initial requirement.

We note that there are multiple ways to define the adjacency relation and that more ways have been tried. The final construction, however, showed to be best suited for the proofs.

7.3 The Proofs

Lastly, this chapter presents the procedures and proof for differential privacy for the respective approaches. Everything is implemented in EasyCrypt.

7.3.1 The Partial Sum

The partial sum procedure implements the first approach. It returns the sum with Laplace noise added and is $(\epsilon, 0)$-differentially private.

Listing 7.5: Laplace of sum implementation.

```
proc partialsum_sum(a : int list) : int = {
  var s, out;
  s <- sum a;
  out <$ lap eps s;
  return out;
}
```

The implementation in Listing 7.7 differs a bit from the explained version, as we sum the entire list, add noise, and return this value. However, it is sufficient enough, as we can imagine a curator making this lists for desired points in the stream.

The proof

The proof that this implementation is ϵ-DP is carried out in two parts. Firstly, we prove that the entire sums of the lists do not differ by more than 1. Secondly, we prove that the Laplace noise disguises that difference. In EasyCrypt it looks as in Listing 7.6.

Listing 7.6: Proof for the partial sum procedure

```
lemma partialsum_sum_dp :
    aequiv[[eps & 0%r]
      Sums.partialsum_sum ~ Sums.partialsum_sum :
      adj a{1} a{2}  ==> ={res}].
proof.
  proc => /=.
  seq 1 1 : (`|s{1} - s{2}| <= 1).
    * toequiv; auto; smt.
  lap 0 1.
qed.
```

In that proof, the first part is constituted by the sequencing where we, in the cut, assert that the difference on the sums are no larger than 1. We note that the `smt` tactic uses all previous lemmas about the adjacency predicate and its associated operators. In particular it also uses lemmas from the imported BigZModule. Ideally, we would have specified that exact lemmas the SMT solver should rely on. However, this is more crucial when the SMT solver can not prove the passes expression within its time limit. In this case, it responds in a timely manner. Furthermore, the `smt` tactic is the reason the proof can be expressed so concisely.

The last part is the application of the `lap 0 1`. This sets up a coupling between `s{1}` and `s{2}` such that they yield the same result.

7.3.2 The Partial Sum'

The next implemented procedure added Laplace noise to each element of a list. This is proven to be $(\epsilon, 0)$ differentially private.

Listing 7.7: Sum of Laplace implementation.

```
 1 proc partialsum'(a : int list) : int list = {
 2   var rs, i, x, s;
 3   i <- 1;rs <- [];
 4   while(i < N){
 5     x <- nth 0 a i;
 6     s <$ lap eps_i x;
 7     rs <- s :: rs;
 8     i <- i+1;
 9   }
10   return aggr rs 0;
11 }
```

First, we note the `aggr` operator which is called on line 10. The operator takes a list and returns a new list where each element is to sum of all previous elements in the list.

This protocol resembles the second approach discussed above. In this setting the protocol is provided a list. Then it adds noise and returns that list back. This is sufficient to show that the second approach is differentially private.

We omit the proof for this procedure as it completely resembles that of Laplace list in section 6.4 with the only difference that this implementation uses an operator that returns the cumulated sum.

7.3.3 The Smart Sum

The last procedure is the smart sum protocol.

Listing 7.8: Smart sum implementation.

```
 1 proc smartsum(a : int list) : int list = {
 2   var i, c, b, s, x, qList;
 3
 4   i <- 0; c <- 0;
 5   b <- witness; x <- witness;
 6   s <- witness;
 7
 8   while(i < Ms){
 9     (* q-list *)
10     qList <- take Qs a ;
11     a <- drop Qs a;
12
13     b <@ partialsum_sum(qList);
14     x <@ partialsum'(qList);
15
16     s <- offsetCopy s x c (i*Qs) Qs;
17
18     c <- c+b;
19     i <- i+1;
20   }
21   return s;
22 }
```

Listing 7.8 is the implementation in question. We note line 4 to line 6. In the listing it takes up only 3 lines, but actually it counts 5 lines. To save space, we have placed multiple assignments on the same lines.

The loop in the procedure run from 0 to `Ms`. `Ms` is the number of `Qs`-sized list the entire `a`-list can hold. Hence we iterate over sublists in `a`, not elements.

Lastly, note that we employ previously implemented procedure on line 13 and line 14. While this is a part of EasyCrypt's procedures, we still can not reason about them in apRHL. In the proofs, we have omitted goals concerning those statements.

The Proof

First, we list the proof statement.

Listing 7.9: Smart sum in differentially private

```
lemma smartsum_dp :
    aequiv[[eps_s & 0%r]
      Sums.smartsum ~ Sums.smartsum :
        adj a{1} a{2} /\ (size a{1}) = Ns ==>
          ={res}].
```

We note that the precondition clause `(size a1) = Ns` is here because we choose to declare the length of the list as an operator. As previously mentioned this makes the proof development easier as we consider the length of the list in the ambient logic.

Initialization The first five statements in the procedure is solely initialization. We prove this by sequencing and stating that the variables are identical after the cut. Furthermore, we note the values of i and propagate the other properties. The ϵ value is also rewritten to suit the loop.

Listing 7.10: Proof of initialization

```
seq 5 5 : (={i, c, b, x, s} /\ adj a{1} a{2}
  /\ size a{1} = Ns /\ i{1} = 0).
+ by toequiv; auto.
conseq <[(((Ms+1)%r)*eps_i_s) & 0%r]>.
+ smt.
```

From Listing 7.10 we see that `conseq` is used as a structural rule to rewrite ϵ.

The While Loop Next, we prove the while loop. For this proof we apply an equal amount of ϵ and zero δ to each loop. Furthermore we use some equivalences in the loop as the loop invariant.

Listing 7.11: Proof of the while loop

```
awhile      [(fun k => (eps_i_s)) & (fun _ => 0%r)]
            (Ms+1)
            [Ms - i]
            (    ={b, x, i, s}
              /\ 0 <= i{1}
              /\  adj a{1} a{2});
  1..4: smt (gt0_eps_s gt0_Ns ax_Qs ax_Ms).
+ by rewrite sumr_const; smt.
+ by smt.
move => k0.
```

Listing 7.11 shows the EasyCrypt proof. Most notable is that `s` is equivalent by invariance. This is because we couple each value put in s through the `lap` tactic. Furthermore, `s` is the return value of the procedure, so the invariant is necessary. The critical part of the proof is the coupling by the `lap` tactic.

Calling Procedures The `call` tactic for the aequiv judgment has not yet been implemented in EasyCrypt. We have circumvented that by admitting the goals. However, the current proof should be sufficient when the actual tactic becomes available.

7.4 Concluding Remarks

This chapter is based on the implementation of the proof of differential privacy for sums over stream from the work by Barthe et al [A5].

Chapter 8

Sparse Vector Techniques

The sparse vector technique is motivated by the need to ask a large number of queries, where only the ones that satisfy certain properties are interesting for the analyst.

A concrete protocol is the Sparse protocol for Above Threshold. The Above Threshold protocol works such that an analyst can ask as many queries as she wants but she only pays for those that result in a result above a certain threshold.

We can generalize the sparse vector technique to consider other metrics for query release. In particular, Barthe et al. [A1] consider a variant called Between Thresholds. It uses the sparse vector technique, but instead of looking at queries that are above a threshold, it looks at queries in a given interval.

8.1 Sparse Vector Technique

We characterize the protocols of the sparse vector technique based on the following parameters.

- Release condition. This can be the above threshold condition, which we look at. It can also be the between thresholds condition.
- Adaptivity. We can choose to get our queries from an adaptive adversary or we assume that we have a list of all queries the adversary wants to execute. This is considered an offline protocol.
- Outer loop. We can sequence the inner loop either by normal sequential composition or make the trade-off between ϵ and δ and use the advanced composition principle.
- Numeric. We consider numeric variants as those who return the result rather than identifying the queries that satisfy the release condition.

As the primary example, we implement the off-line version of the SVT using the above threshold condition as release condition and sequential composition. After that, we show that we can easily extend the protocol to consider an adaptive variant and the numeric variant.

Proof-technically we consider two primary parts: The outer loop and the inner loop. The inner loop is the protocol that returns the next interesting query. The outer loop then composes the inner loop. Typically the inner loop is the interesting part, and we use sequential or advanced composition for the outer loop.

8.2 Preliminaries

Before delving into the proofs we, as usual, elaborate on the preliminaries.

8.2.1 Queries and Databases

A query and the database are abstract entities. In EasyCrypt we define them by a type.

Listing 8.1: Query and database for SVT.

```
type query.
type db.
op evalQ : db -> query -> int.
op nullQ : query.
```

Listing 8.1 shows the implementation. Along with the type of the query and the database, there are two operators: `evalQ` and `nullQ`. They act as the value level interaction we have with a database and a query. These functions suffice to implement the adjacency relation and build the protocol.

8.2.2 Database Adjacency

This application considers a collection of queries. This reflects in the definition of database adjacency. In this application we consider two databases to be adjacent if the same query executed on both returns a value that differs by at most one.

In EasyCrypt it is formalized as in Listing 8.2.

Listing 8.2: Adjacency of databases.

```
pred adj : db & db.
axiom one_sens d1 d2 q: adj d1 d2 =>
  `|evalQ d1 q - evalQ d2 q| <= 1.
```

We see that the definition of the property is split from the declaration of the predicate. In the previous chapter, we gathered the declaration and definition in the same statement. The difference here is that we need to quantify all queries. However, The `q` parameter should be a part of the predicate parameters. Alternatively we could have used `forall` quantification inside the predicate. This technique is parameterized over the queries.

8.2.3 Ambient Properties

As in the previous two case studies, we define a number of properties in the ambient logic through operators. In particular, the properties include the length of lists and the ϵ-parameters. As before, the justification is that they are not modified by the program, and they are needed in the ambient logic.

8.3 Proof of Above Threshold

The above threshold protocol takes a list of queries and returns the index of the first query that yields a result more than a given threshold. Surprisingly, there is no privacy cost of all the queries below the given threshold. This section elaborates on the proof for privacy implemented in EasyCrypt

Listing 8.3: Procedure for the Above Threshold protocol.

```
1  proc above_t (d : db, qs : query list
2    , tHold : int) : int = {
3    var i, q, s, r, t;
4    i <- 0; r <- r_init; t <$ lap eps_t tHold;
5    while (i < N){
6      if (r = r_init){
7        q <- nth nullQ qs i;
8        s <$ lap eps_i (evalQ d q);
9        if(t <= s){
10         r <- i;
11       }
12     }
13     i <- i+1;
14   }
15   return r;
16 }
```

First, the implementation of the protocol. Listing 8.3 shows the protocol we will prove differential privacy for. There are a couple of things done differently compares to other implementations. We decided to split a single `if`-statement with two conjugate clauses into two; an outer statement (on line 6) and an inner statement (in line 9). The outer statement enters the `then`-branch if we did not previously see a result. The inner statement enters its `then`-branch if the evaluated query is above the threshold. We did not implement a formal proof of equivalence to a protocol enjoying broad consensus and the equivalence to such boils down to our convictions. However, the most important thing about the protocol is that it returns the right value.

Listing 8.4: Lemma for the Above Threshold protocol.

```
lemma above_t_dp :
  aequiv[[eps & 0%r ]
    AboveT.above_t ~ AboveT.above_t :
        (adj d{1} d{2} /\ ={qs, tHold} /\
            N = size qs{1}) ==> ={res}].
```

Having implemented the protocol, we start the proof-process by stating the lemma. Listing 8.4 shows the lemma statement and is fairly standard. We assume adjacent databases, identical lists of queries, identical thresholds, and that `N` amounts to the size of the query list. With this in order, we can begin the proof.

For the proof, we present the most important parts. For the full proof, we refer to the source files. In this exposition, we will cover the application of pointwise equality, the approximate while rule, handling the critical iteration, and the coupling in the critical iteration.

Before going into the parts of the proof, we make a note on the **`eps`** value and threshold coupling. For this proof **`eps`** is split into 2 quantities: $\frac{\texttt{eps}}{2}$ that goes under the name **`eps_t`** and $\frac{\texttt{eps}}{4}$ that goes under the name **`eps_i`**. **`eps_t`** is used for threshold coupling, and twice the amount of **`eps_i`** is used to couple s in the critical iteration.

From now on we consider t coupled such that `t{1}+1 = t{2}`.

Listing 8.5: Application of pointwise equality.

```
1  pweq (r, r).
2  + while true (N - i); ···.
3  + while true (N - i); ···.
4  + move => &1 &2 /(_ (r){1}); smt().
5  move => R.
```

Listing 8.5 shows the application of pointwise equality. After this, we have an ambient variable `R` that corresponds to r in the procedure. Furthermore, the postcondition is changed from `={r}` to `R = r{1} => R = r{2}`. This change is critical for our coupling as we now show that the right-hand execution is a consequence of the left-hand execution.

The three subgoals which are proven immediately are respectively that both sides satisfy the lossless property (i.e. they terminate) and that `R = r{1} => R = r{2}` indeed implies the equivalence of **`r{1}`** and **`r{2}`**.

The last statement on line 5 moves the newly created universal quantification of **`r`** to the assumptions.

Listing 8.6: Application of the awhile tactic.

```
1  awhile [ (fun k => if k=N-R then 2%r*eps_i else 0%r)
2         & (fun _ => 0%r)]
3         (N+1)
4         [N-i]
5         (={i, qs}
6          /\ adj d{1} d{2}
7          /\ N = size qs{1}
8          /\ 0 <= i{1}
9          /\ t{1}+1 = t{2}
10         /\ (r{1}= r_init => r{2}= r_init)
11         /\ (r= r_init \/ r < i){1}
12         /\ (r= r_init \/ r < i){2}
13         /\ (r{1} = R => r{2} = R)
14         ); 1..5: smt (eps_gt0 N_gt0).
15 + by smt.
16 move => k0.
```

Next, we prove the while loop. This is slightly more complicated than the previous proofs using the while tactic: we spend all the privacy budget in a single iteration rather than using it equally in all iterations. By doing this, we need to consider the `k` parameter passed to the function that decides the privacy use. If we look at Listing 8.6, we see that the body of the function branches on `k=N-R`. This branching makes sense recalling that `k` denotes the loop variant. The loop variant is set to `N-i` and is a strictly decreasing value. As we want to identify the `k`'th iteration, we branch on `k=N-R`.

Line 2 in the Listing is the function deciding δ for a given iteration. We do not use any δ and omit it. Then we have the upper limit of the loop variant in line 3 and the actual loop variant in line 4. Line 5 to line 13 contain the invariant. The non self-explanatory parts are those from line 10 and down: The clause on line 10 says that if we did not see the critical iteration on the left-hand side, then we did not see it on the right-hand side. The next two lines state that we either saw the critical iteration, or we did not. Finally, we make the bond between invariant and the ambient variable `R`. This is critical in the proof when we case on the critical iteration.

Lastly, we see that we have introduced a universal quantifier for `k`. We move that to the assumptions by the statement in line 16.

Listing 8.7: Casing for threshold condition.

```
if{1}.
+ rcondt{2} 1;1:by auto; smt().
case @[ambient] (k0 = N-R) => Hk0. wp.
```

When in the loop, we need to consider two cases: the case where we should disregard the result of the query, and the case where we should act on it. To do this casing, we need to look two places. We need to make sure that we enter the then branch of in the procedure, and we need to make sure that `k0` in the ambient logic is equal to `N-R`. To do this, we use the `if{1}` and the `case` tactic as listed in Listing 8.7.

Listing 8.8: Couplings of the s variables.

```
* exists * t{1}, t{2}; elim * => t1 t2.
  conseq (_ : _ ==> (t1 <= s){1} = (t2 <= s){2});
    first smt ().
  lap 1 2; smt (one_sens).
```

```
* exists* (evalQ d q){1}, (evalQ d q){2};
    elim* => e1 e2. wp.
  lap (e2-e1) 0; smt (one_sens).
```

Lastly, we do the couplings. We have a coupling for each of the cases generated from above part. These two couplings are displayed in Listing 8.8. This first is the coupling in the critical iteration. We couple `s{1}` and `s{2}` such that if `s{1}` enters the critical branch then `s{2}` does also. This requires the coupling parameter of the `lap` rule to be two because of the threshold coupling and the difference of up to one in the query results (one sensitivity queries).

The other coupling is a null coupling. Is uses no privacy, but does not employ any coupling, so the two Laplace distributions act freely. However, this is not a problem as we know we are not in the critical branch.

8.4 Extensions

We augmented the Above Threshold protocol in three ways: The Sparse protocol to return the c first queries above a threshold, the adaptive variant to allows online, adaptive queries, and the numeric variant that returns the numeric value of the query.

8.4.1 Proof of Sparse

The Sparse protocol takes a list of queries and returns a list of maximum C elements that all satisfies the above threshold constraint.

Listing 8.9: Procedure for the Sparse protocol.

```
proc sparse(d : db, qs : query list
  , tHold : int) : int list = {
  var rs, i, r;
  rs <- []; i <- 0;
  while(i < C){
    r <@ above_t(d, qs, tHold);
    qs <- drop r qs;
    rs <- rs ++ [r];
    i <- i+1;
  }
  return rs;
}
```

Listing 8.9 shows the implementation of the protocol. It is a straightforward implementation of sequential composition over C homogeneous protocols. In this case, the inner protocol is simply the `above_t` as presented above.

Listing 8.10: Lemma for the Sparse protocol.

```
lemma sparse_dp :
  aequiv[[(eps_o_i*(C+1)%r) & 0%r]
    AboveT.sparse ~ AboveT.sparse :
     (adj d{1} d{2} /\ ={qs, tHold}) ==> ={res}].
```

Listing 8.10 shows the statement of the lemma. Here we note that we use the ϵ-parameter from the `above_t` protocol $C + 1$ times. The $C + 1$ is chosen to accommodate the the `awhile` tactic.

The proof closely resembles those done for lists previously, and we omit the details.

8.4.2 Proof of Adaptive Above Threshold

By a minor change, we can transform the Above Threshold protocol into the adaptive counterpart. We omit the procedure from this exposition as its changes are primarily og a technical character. However, the new proof statement has some interesting details.

Listing 8.11: Lemma for Adaptive Above Threshold.

```
lemma above_t_adv_dp :
  aequiv[[eps & 0%r ]
    Svt_a(A).adaptive_above_t
  ~ Svt_a(A).adaptive_above_t :
      (adj d{1} d{2} /\ ={glob A, tHold}) ==>
        ={res, glob A}].
```

In Listing 8.11 we see that the equivalence of the `qs` variables is exchanged for equivalence of `glob A` (The global state of the `A` module) in the precondition. In the postcondition, we also promise equivalence of `glob A`. This strengthening of the postcondition is crucial if we want to integrate the protocol in the Sparse protocol as we need to know that the adversaries are equivalent for the next run.

The proof progresses mostly like the proof for the offline variation. The primary difference is that we apply pointwise equality to the `glob A` as well as the `r` variable.

8.4.3 Proof of Numeric Above Threshold

As a consequence of the coupling was set for the proof of the above threshold protocol, it is not immediately possible to return the value. However, to get the answer for the query the protocol releases, we can apply a sequential composition: we first run the above threshold protocol. Then we run the query at the index return by the protocol and return that answer.

This technique yields 2ϵ-differential privacy as above threshold costs ϵ and the Laplace mechanism on the query result costs ϵ.

The proof and procedure are omitted as the interesting things are procedure calls and sequencing respectively presented in Chapter 7 and Chapter 6.

8.5 Concluding Remarks

The proofs presented in the work by Dwork et al. [A8] is a semantical proof; it works on the probability density function for the Laplace distribution.

The proof for the offline above threshold protocol has before been implemented in EasyCrypt [A2]. However, it was not possible to obtain a copy of the implementation. As such, it was reimplemented for this exposition.

Part III

Conclusion

Chapter 9

Related and Future Work

CertiCrypt was developed as a precursor to EasyCrypt [A4]. It is integrated into the Coq theorem prover and it used slightly different formalisms. The choice of implementing EasyCrypt as its own theorem prover allowed for the integration to an SMT solver which greatly improves the proof authoring experience.

In this work, we primarily look at the protocols from the first chapters from Dwork and Roth's book. There is much more material than we could hope to look at within the given time frame. Going through the book to formalize more protocols and concepts would be a next step for work focused towards differentially privacy.

The implementation of the proofs in a theorem prover is work that ensures the correctness of the proofs. The version of EasyCrypt we used was highly experimental and is also a good objective for further work. First and foremost a full implementation of the `aequiv` judgment along with documentation will help the implementation process. In particular, we needed the `sp` and the `call` tactics.

As a last remark, this thesis contributed by presenting a view on formalizing differential privacy. Dwork and Roth wrote an important book on the subject [A8], but their proofs are hard to transfer into various formalisms as they are of a semantical nature. A contribution in this line could be to contribute EasyCrypt tutorials with offset in differential privacy.

Chapter 10

Conclusion

Differential privacy allows the data curator to release data in a privacy-preserving manner through a number of protocols. However, the correctness of these protocols is hard to asses by traditional software development techniques such as testing and review. Therefore, we need a rigorous formal framework to effectively discriminate protocols that seem differentially private but are not. This thesis presented the foundations for differential privacy including the following

- A presentation of differential privacy.
- The mathematical tools we need to build the logical framework for differential privacy.
- The apRHL logic that allows for reasoning about differential privacy
- The EasyCrypt theorem prover that implements the `aequiv` judgment; an implementation of the apRHL.

This presentation was followed up by four case studies that show the applications of differential privacy along with the implementation of the proofs for their differential privacy property in EasyCrypt. These case studies were the following

1. The randomized response query to allow participants to answer a question in a differentially private way.
2. A number of simple use-cases of the Laplace mechanism, its generalization to two elements, and its generalization to a list of elements.
3. As an application of real-world we consider the sums over streams. The implementation process showed a number of subtleties a protocol author would not need to consider. In particular the exact definition of the adjacency relation.
4. The last case study was a well discussed protocol. The Above Threshold protocol was shown to be notoriously tricky [A11]. However, it worked out to make a satisfying procedure and prove it to be differentially private.

As argued in the introduction, implementing these protocols is important. It is what ensures our privacy when large amounts of private data becomes easily accessible to researchers and interested people. With these formal proofs we can be completely certain that the procedures are correct.

Lists

List of Definitions

List of Lemmas

List of Theorems

List of Listings

List of Figures

List of Examples

Bibliography

[A1] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. *CoRR*, abs/1606.07143, 2016.

[A2] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. *CoRR*, abs/1601.05047, 2016.

[A3] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Coupling proofs are probabilistic product programs. *CoRR*, abs/1607.03455, 2016.

[A4] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009.

[A5] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *SIGPLAN Not.*, 47(1):97–110, January 2012.

[A6] Gilles Barthe and Federico Olmedo. *Beyond Differential Privacy: Composition Theorems and Relational Logic for f-divergences between Probabilistic Programs*, pages 49–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[A7] Fida K. Dankar and Khaled El Emam. Practicing differential privacy in health care: A review. *Trans. Data Privacy*, 6(1):35–67, April 2013.

[A8] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3&#8211;4):211–407, August 2014.

[A9] Michèle Giry. A categorical approach to probability theory. pages 68–85. 1982.

[A10] C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, Piscataway, NJ, USA, 1989. IEEE Press.

[A11] Min Lyu, Dong Su, and Ninghui Li. Understanding the sparse vector technique for differential privacy. *CoRR*, abs/1603.01699, 2016.

[A12] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 111–125, Washington, DC, USA, 2008. IEEE Computer Society.

[A13] Tetsuya Sato. Approximate relational hoare logic for continuous random samplings. *CoRR*, abs/1603.01445, 2016.

[A14] EasyCrypt Team. *EasyCrypt Reference Manual.* https://www.easycrypt.info/documentation/refman.pdf, version 1.x edition, feb 2017. https://www.easycrypt.info/documentation/refman.pdf.

Part IV

Appendix

EasyCrypt Documentation

The implementation is still under heavy development and is therefore not a part of the main distribution of EasyCrypt. This section seeks to provide documentation for the rules used to carry out the proofs.

◎ `awhile`

Syntax: `awhile [`f_ϵ`,`f_δ`] n [v] I`. Reasoning about while loops in an apRHL judgment using sequential composition.
The arguments to the tactic are as follows:

1. f_ϵ has the type int $\to$ real. It takes a parameter, k, and returns the additive amount of ϵ when the loop variant equals to k.
2. f_δ has the type int $\to$ real. It takes a parameter, k, and returns the additive amount of δ when the loop variant equals to k.
3. `n` is integer denoting the upper bound of the number of iterations.
4. `v` is the loop variant.
5. `I` is the invariant.

Following is the list of generated subgoals. Note that not all might be included when actually applying the tactic.

- $n > 0$
- $\forall k. f_\epsilon > 0$
- $\forall k. f_\epsilon > 0$
- If the invariant is true and the loop variant is less than 0, then the loop condition is false.
- The precondition before the loop implies the invariant, that the loop variant behaves identically on both sides, and that the initial loop variant is less than the upper bound.
- The sum over all f_ϵ amounts to available ϵ
- The sum over all f_δ amounts to available δ

◎ `toequiv`

Syntax: `toequiv`. Transforms an `aequiv` goal into an `equiv` goal. The two judgments relate in such that `equiv[M.a ~ M.b : pre ==> post]` is equivalent to `aequiv[[0%r & 0%r] M.a ~ M.b : pre ==> post]`.
Applying this rule to a goal like `aequiv[[eps & delt] M.a ~ M.b : pre ==> post]` yields three subgoals:

1. `eps <= 0`.

2. `delt <= 0`.
3. `equiv[ M.a ~ M.b : pre ==> post]`.

For example, if the current goal is

```
Type variables: <none>

────────────────────────────────────────────────────────
e = eps
d = delt

pre = true

    M.b ~ M.b

post = true
```

Then we run `toequiv.` to get

```
Type variables: <none>

────────────────────────────────────────────────────────
0%r <= eps
```

and

```
Type variables: <none>

────────────────────────────────────────────────────────
0%r <= delt
```

and

```
Type variables: <none>

────────────────────────────────────────────────────────
pre = true

    M.b ~ M.b

post = true
```