UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Tejas Anil Shah

The Joy of EasyCrypt:

The Least Painful Way to Learn Formal Verification of Cryptography

Master's Thesis (30 ECTS)

Supervisor: Dominique Unruh, PhD

Tartu 2022

The Joy of EasyCrypt

Abstract:
The ideas of computer-assisted proofs and formal verification have been around for a considerable time, but the field has a high barrier to entry. When it comes to formally verified cryptography, the barrier is exceptionally higher since it requires familiarity with even more topics – translating to interesting challenges in the pedagogy of the field. Beginners learning to work with the tools of formal verification of cryptography face considerable difficulty due to the lack of resources to help with their learning journey.

In particular, when it comes to EasyCrypt, a toolset for reasoning about cryptographic proofs, an incomplete and sometimes obsolete reference manual, and a lack of learning material for beginners, together with the high barrier of entry, hinder the pace of learning. This work attempts to remedy this problem by creating formal verification learning material.

We present the first few chapters of an open-source textbook, The Joy of EasyCrypt, which establishes the context and develops the theory and practical exercises required to learn and work with EasyCrypt. This marks the beginning of an effort to make formal verification with EasyCrypt more accessible and easier to learn.

Keywords:
EasyCrypt, Computer-aided Cryptography, Formal verification

CERCS:
P170 - Computer Science, Numerical Analysis, Systems, Control

EasyCrypti rõõmud

Lühikokkuvõte:
Arvutipõhiste tõestuste ja formaalse verifitseerimise ideed on juba pikka aega olnud olemas, kuid selles valdkonnas on sisenemisbarjäär kõrge. Kui tegemist on formaalselt verifitseeritud krüptograafiaga, on see barjäär veelgi kõrgem. Krüptograafias nõuab see rohkemate teemade tundmist, mis toob valdkonna pedagoogikas esile huvitavaid väljakutseid. Algajad, kes õpivad krüptograafia formaalse verifitseerimise vahenditega töötama, seisavad õppematerjalide puudumise tõttu silmitsi märkimisväärsete raskustega.

Krüptograafiliste tõestuste tööriistakomplekti EasyCrypti puhul takistab õppimistempot puudulik ja mõnikord vananenud kasutusjuhend, algajatele mõeldud õppematerjalide puudumine ning kõrge sisenemisbarjäär. Käesolevas töös lahendati seda probleemi, luues formaalse verifitseerimise avatud lähtekoodiga õpiku "The Joy of EasyCrypt".

Esitleme töös valminud õpiku esimesi peatükke, milles luuakse formaalse verifitseerimise kontekst ning arendatakse EasyCrypti õppimiseks ja sellega töötamiseks vajalikku teooriat ja praktilisi harjutusi. Sellega muudame EasyCrypti formaalse verifitseerimise metodoloogiad kättesaadavamaks ja kergemini õpitavaks.

Võtmesõnad:
EasyCrypt, Arvutipõhine Krüptograafia, Formaalse verifitseerimise

CERCS:
P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Acknowledgements

I'd like to first thank my supervisor, Prof. Dominique Unruh for patiently answering all my questions related to EasyCrypt, and being the kind guiding force needed for the project. I'd also like to thank Ekaterina Zhuchko for proofreading an early draft of this project and encouraging me to trudge on.

I thank my parents, Anil and Kavita, and sister, Pooja, for being my pillars of support.

I thank my friends Stefano, Mitra, Thamara, Uku and Karan for keeping me company through thick and thin. I also thank my Philosophy study group (Sharmishtha and Neeraj), my TEDxBITSGoa group, and Paul for all the conversations and discussions we've had over the weekends. My studies at the University of Tartu wouldn't have been the same without them.

I also thank Paulina, Uku and Oliver for Estonian translations of the abstract.

Lastly, I'd like to thank Maria and the Industry Collaboration Team (Georg, Helen and Kristjan) at the Institute of Computer Science for allowing me access to the Sandbox space and helping me stay motivated over the duration of this project.

Contents

1 Introduction

The research community of formal verification of cryptography faces two interesting challenges. Firstly, it has a high barrier to entry since it requires familiarity with a fairly wide range of topics such as higher math, cryptography and more niche skills such as working with proof assistants and formal theorem proving. While math and cryptography courses are readily available at universities, lectures on logic, formal verification, and proof assistants are few and far between. Secondly, with security protocols getting more complex, the effort required to verify them formally increases substantially. This implies that the community's main focus ends up being research with little effort dedicated to teaching and training beginners.

These challenges are especially pronounced when it comes to EasyCrypt, a toolset for reasoning about cryptographic proofs. Additionally, the EasyCrypt user community remains relatively small with slow growth. This compounds the aforementioned problems since the researchers are hard-pressed on time to work on their research and continue the development of the tool.

For any open source project to thrive, it needs to be able to welcome beginners and bring them up to speed rapidly both in terms of its usage and development. At the very least, users expect a straightforward setup process and good documentation. For instance, let us look at the Coq ecosystem. Coq is a formal proof management system with a simple setup, great documentation, and excellent learning material in the form of the Software Foundations series [1] and other excellent textbooks and tutorials [2]. Additionally, there is also a straightforward way for developers to contribute to the ecosystem [3].

In comparison, the EasyCrypt project currently has a complicated setup process with various external dependencies, lacks a reference manual and meaningful documentation. Of course, it is an unfair comparison since Coq has been under development since 1989, while EasyCrypt has only been around since 2009. So Coq's community has had a lot more time to mature into what it is today.

We also understand the fact that the developer team of EasyCrypt is hard-pressed for resources and time to write documentation and learning material. However, the point we stress is that this creates a vicious cycle of no documentation since the lack of documentation translates to lesser interest from beginners and slow growth. This implies that the community doesn't have resources to spend on documentation.

The lack of documentation and learning material also forces learners to rely on local experts or the community to answer their questions. This is often challenging for beginners and, at the same time, inefficient compared to good learning resources. Here, we'd like to point out that we became part of the Zulip community of EasyCrypt developers, but fear and doubt stopped us from asking questions there. We take part of the blame for not reaching out. However, it is essential to understand why this happened. When it comes to highly sophisticated topics and tools, learners hesitate to reach out and ask for help. This again goes to show the need for more documentation or a roadmap

to help beginners learn independently. Our work is a modest attempt at creating such learning material. In addition, the community can also benefit from a wiki with all the shared resources, new papers, and tutorials related to EasyCrypt.

1.1 Our work

We present a compilation of learning materials in the form of reading material and EasyCrypt theory (`.ec`) files for practice. The reading materials develop the theory and background required to work on (`.ec`) files which provide hands-on practice.

Since we are dealing with cryptography and formal verification, our work derives inspiration from two textbooks, Software Foundations that we mentioned earlier and Joy of Cryptography (JoC) by Mike Rosulek [4], an open-source undergrad textbook for cryptography.

Our goal for this thesis was to cover the first two chapters (One-Time Pad, The Basics of Provable Security) of JoC and provide hands-on exercises for the same. However, once we started working on the project, it became clear that there is a lot of ground to cover in terms of formal logic and EasyCrypt fundamentals. Hence the focus of our work changed to explaining these basics and covering as much as possible in the given time. In the end, we present the following chapters:

1. **Preface**: In this section, we provide the reader with a brief account of the history and evolution of the field. We convince the reader about the need for it, the criticism it faces, and the field's response to those challenges.

2. **Approaches to computer-aided cryptography**: We give the readers a high-level overview of the different approaches to formal verification in cryptography and explain the two approaches to verifying design-level security with examples.

3. **EasyCrypt**: In this section, we describe the EasyCrypt environment and then provide a motivating example for the readers to get a taste of working with EasyCrypt.

4. **Working with the different logics in EasyCrypt**: In this chapter, we explain how different logics work in EasyCrypt. This forms the bulk of our work since explaining the tactics, and the theory required an immense amount of time. In writing this chapter, we realised the effort it takes to document and provide examples for practice. We present three chapters: Ambient logic, Hoare logic, and Relational Hoare logic and the practice files needed to work with them.

Our work also includes updates to the lecture and homework files associated with the course *Verification of Cryptography with EasyCrypt* [5] that was offered by Prof. Dominique Unruh in Spring 2015 at the University of Tartu. We sincerely hope that courses related to EasyCrypt are offered again in the university since a lot of learning comes from having a peer group.

Additionally, we also created a few screen recordings with explanations of various things about EasyCrypt. We have shared them on YouTube for beginners to learn the basics [6].

Note: The material was developed on the EasyCrypt version with the git hash: ce56b105 [7] and has been tested to work on the stable release of EasyCrypt with the git hash: ea77e0ed [8] as well.

1.2 Organisation of this document

As mentioned above, our work is a compilation of reading material and (`.ec`) files. The ideal way to work through the material would be first to read the theoretical aspects that may be required and then work with the `.ec` files in EasyCrypt.

This document is structured such that reading it can give the reader a good understanding of how EasyCrypt works and a lot of the ideas required to work with it. However, there is no substitute for working with the tool and practising the exercises. We include the required exercise files in the **Extras** section.

All materials are hosted in an open source repository on GitHub called the-joy-of-easycrypt [9] with the reading material presented in the `README.md` files and the practice material in the `.ec` files for each topic.

2 Preface

"Program testing can be used to show the presence of bugs, but never to show their absence!"
– Edsger W. Dijkstra

Building reliable software is extremely tough. In order to guarantee the reliability of software, there are plenty of different approaches that can be employed, ranging from testing to various software building methodologies. However, these are among the weakest guarantees we can provide or are simply best practices that can ensure some level of reliability.

With the explosion of software applications being used, bad software has also been the source of literal explosions. For instance, NASA lost its Mars Climate Orbiter in 1998 [10] due to a bug in converting units from the Imperial system to the SI system. More recently, bad software led to the death of 346 people onboard two Boeing 737 MAX airplanes that crashed in 2018 and 2019 [11]. Bad software has proven to be extremely costly.

On the opposite end of the spectrum of software reliability guarantees lies the field of formal verification. The core idea behind formal verification of software is to provide mathematical guarantees of program behaviour based on specifications. It grew out of the field of formal methods in mathematics and later computer-aided mathematical proofs. So, formal verification rests on the firm foundations of formal logic and reasoning.

Cryptography also follows the same pattern when it comes to reliability guarantees. Testing and following best practices are still the go-to methods to ensure the reliability of most cryptographic protocols; these methods provide the weakest guarantees, as we discussed above. To provide stronger guarantees, the ideas of formal verification have been applied to cryptography, giving rise to the field of formally verified cryptography. As these fields advanced and became more and more complex, software tools that could aid us in this process were built. There are several approaches to formal verification of cryptography, as we will see. We will focus on one approach to the field (design-level security) and work with EasyCrypt, a toolset that allows us to verify game-based cryptographic proofs.

2.1 Who is this text for?

This work consists of learning material for EasyCrypt. It is inspired by and written in the style of Software Foundations, a series of electronic textbooks that explain the mathematical underpinnings of reliable software. For concepts related to cryptography, we aim to track The Joy of Cryptography(JoC), an open-source undergrad textbook for cryptography.

This material is written for anyone curious about cryptography and wants to under-

stand formal verification and computer-aided cryptography in the computational approach to design-level security. In line with that, we assume that the reader is completely new to the field of cryptography, and for some concepts, the reader will be referred to JoC when needed. We will offer simplified explanations of concepts required for formal verification as we go. We expect familiarity with discrete mathematics, data structures and algorithms, a basic familiarity with the command line and the ability to work with the Emacs text editor. Essentially, if you have a little programming experience, then you should be able to work your way through the ideas presented in this work.

Although Emacs is pretty complex, we only expect the reader to know how to open files and navigate them to begin with. These skills can be learnt quickly with the tutorial that is presented upon a fresh install of Emacs. Additionally, the guided tour of Emacs [12] is helpful for those who want to know more about what Emacs is capable of and don't want to go through the entire tutorial.

As with the original text, **joy is not guaranteed**. Formal verification can be challenging, so we ask you to be patient and keep working with the material. A tip that we can provide is to start and learn as you go.

Let us now take a look at how the field of computer-aided math developed and trace the need for formal verification in cryptography. We will also look at the core ideas of computer-aided cryptography[1] and build up to working with EasyCrypt.

2.2 Formal verification: A brief history

The ideas of formal verification can be traced back to Leibniz's ideas about *characteristica universalis* back in the 17th century. He imagined it to be a universal conceptual language that could be used to solve logical problems. However, we do not need to go that far back; for our intents and purposes, formal methods and later computer-aided math came into the limelight with the famous four-colour theorem being proven with the help of computers by Appel and Haken [13, 14] in 1977.

The hypothesis of the four-colour theorem is quite simple and elegant. It says, "Any planar map can be coloured with only four colours". The computer-aided proof by Appel and Haken attracted plenty of criticism since it was done by performing an exhaustive case analysis of a billion cases. An exhaustive case analysis means that the computer went through every single case that could arise when trying to colour a map and came back saying that four colours are all it takes. For an elegant problem, the proposed computer-aided proof felt like using a sledgehammer to crack a seemingly innocent nut, and this style of theorem proving polarised the research community.

Nevertheless, the computer-aided proof of the four-colour theorem is a significant milestone, and it set off more work and efforts in the field of computer-aided mathematics.

[1] We will use the term *formally verified cryptography* interchangeably with *computer-aided cryptography*

The field has grown considerably since then, and there are a multitude of tools available for use. For instance, one can use theorem provers like Coq or Isabelle to formally prove general mathematical statements with the help of a computer. Or, one can use specialised tools like EasyCrypt or Tamarin to work with specific domains of formal verification, like cryptography in this case. As for the four-colour theorem, a fully computer-free proof is yet to be discovered.

2.3 Formal verification in cryptography

As the field of formal verification in math took off and got established over the years, there was a problem brewing in the academic circles dealing with cryptography. At its core, modern cryptography is based on math, and to demonstrate different security properties of protocols, we come up with mathematical proofs and guarantees about them. As the field advanced, these proofs started getting fairly complex and often too complicated for humans to go through. This problem can be best illustrated with this excerpt from a paper by Shai Halevi [15]:

"The problem is that as a community, we generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect). I became acutely aware of this when I wrote my EME* paper [16]. After spending a considerable effort trying to simplify the proof, I ended up with a 23-page proof of security for a — err — marginally useful mode-of-operation for block ciphers. Needless to say, I do not expect anyone in his right mind to even read the proof, let alone carefully verify it."

In the paper, Halevi highlights that cryptography as a field has advanced to the point where the proofs are so complex that the field could benefit from automation or, at the very least, some help from machines. In a way, automated and computer-assisted are the two approaches that are possible in the field of formal verification of cryptography.

2.4 What is computer-aided cryptography?

At this point, it is a good idea to read Sec 1.1 of JoC [17] to understand what cryptography is and is not. To summarise and add to the ideas from JoC, modern cryptography deals with three main problems related to the security of communication. They are the following:

1. **Privacy**: Protecting information from being accessed by unauthorised parties
2. **Integrity**: Protecting information from being tampered with or altered
3. **Authenticity**: Making sure the information is from an authentic source

Each of these properties can be defined as a mathematical property of information, and claiming that a cryptographic protocol protects a certain property is equivalent to

proving that the information possesses that mathematical property. Given that the proofs can be complicated and hard to follow, the idea of using computers to verify the proofs comes in here.

To illustrate this idea better, let us first meet Alice, Bob and Eve. We assume that Alice wants to send a message to Bob and wants the communication to be private. While Eve wants to eavesdrop on their communication. One way to prove that a system of communication is secure is to think of Alice, Bob and Eve playing a game where the objective of the game for Alice and Bob is to communicate in a way that Eve can't differentiate between the messages and a random string of characters. This would mean that Eve can extract just as much information from an intercepted message as they can extract from random noise, implying that the communication is private. This game can be modelled mathematically, and we can prove that the communication was private. Proofs written in this style are called game-based cryptographic proofs.

These ideas of representing information security properties as mathematical properties and the games that Alice, Bob and Eve play (game-based cryptographic proofs) are essentially the cornerstones of provable and verifiable security. Depending on the definition of the games and conditions related to them, the proofs can turn out to be fairly intricate, as we established earlier. In the worst case, handwritten proofs could even be wrong. Incorrect proofs translate to significant security risks and reaffirm the need for the field of formally verified cryptography. The complexity and difficulty of verifying these proofs translate to a need for more automation and computer-aided verification – tools like EasyCrypt are built to do just that.

2.5 Criticisms of computer-aided cryptography

Although the need for the field has been well established, it faces many challenges and has its own shortcomings.

Only a few important protocols, such as Bluetooth, and TLS, are formally verified, and it is less likely that verifying these protocols uncovers serious flaws. This is because they are subject to extensive testing and attacks. Whereas the protocols that aren't used to support major industrial applications generally aren't subject to the same level of scrutiny. They only receive limited manual testing, which leaves them prone to errors. This is where formal verification can actually be used to improve them significantly. The complexity of using the tools poses a significant hurdle, and these protocols are left unverified.

Additionally, a high barrier to entry to the field compounds the problem of protocols being left unverified. This is because teams developing protocols can't spend the time and effort required to go through the process of formally verifying them. [18]

In the end, the benefits of formal verification can seem marginal as the tools themselves can have flaws, leading to the philosophical conundrum of "Who will guard the guards themselves?"

As a response to these challenges, the field offers the following rebuttals:

1. The field is still new, and the tooling is undergoing active development. Once the tools mature and are easier to use, we can expect the industry to move in the direction of requiring formal verification for protocols to be put into use. For instance, the development of the 5G protocol happened in close collaboration with the formal verification community [19] and is an encouraging move in the right direction

2. Even though the tools might have flaws, the point to note here is that it would be exceptionally hard for a mathematical proof to be wrong and also getting past a formal verification check. So a flawed tool already provides better security compared to a human-verified proof. The standard to beat remains a highly specialised set of human eyes verifying a mathematical proof. The argument of who guards the guards themselves already applies to the current situation, and formal verification is, in fact, a significant improvement.

3. The high barrier to entry remains a problem with no simple solutions. However, this argument is akin to saying that nuclear physics has a high barrier to entry. Although true, this situation arises from the fact that these fields are highly specialised and require years of training to reach a certain level of competence.

All in all, there is plenty of work left to do in the field.

3 Approaches to computer-aided cryptography

The previous chapter outlined how the field of computer-aided cryptography came about, its core ideas and the challenges it faces. In this chapter, we will take a brief look at the different approaches to computer-aided cryptography, further develop the concepts of game-based proofs and understand where EasyCrypt fits into the picture.

The tools for computer-aided cryptography come in different flavours based on what is being dealt with. Broadly speaking, the tools are developed for the following:

1. **Design-level security**: To reason about the design of a protocol
2. **Functional correctness and efficiency**: To formalise that the implementations match the designs
3. **Implementation-level security**: To make sure that the implementations do not have other security risks

Different tools have been developed to work with verification at each level, and it is out of the scope of this work to go into further details. However, we point the interested readers to the paper *SoK: Computer-aided Cryptography* [20] which is the source of our summary. The Systematisation of Knowledge (SoK) provides good insights into the strengths and weaknesses of many available tools. We will, however, briefly touch upon design-level security as EasyCrypt deals with this level of security specifications.

3.1 Design-level security

The idea of tools that are developed for design-level security is to be able to reason about and formally prove security aspects of a protocol purely on the basis of its design.

Let's go back to Alice, Bob and Eve to understand this better. As usual, Alice and Bob want to communicate secretly, while Eve would like to eavesdrop. To keep their messages secure, Alice and Bob agree on the idea that they will jumble the words in a sentence in a specific order that they both know and then send the message across on a piece of paper. For instance, let us say that they agree that a message will be jumbled in the following way:

$$m = \text{“I think Eve can decode this easily.”}$$

$$\downarrow$$

$$c = \text{“Decode this easily I think Eve can.”}$$

Here m denotes the plaintext message and c denotes the ciphertext

However, as the message says, Eve can decode the “secure” message quite easily by simply taking a look at the ciphertext. This protocol is insecure by design.

Although the example is slightly contrived, it makes the ideas clear. These ideas aren't too far from reality either, as we have a class of ciphers called the Caesar ciphers, which work by shifting letters in the alphabet by a certain agreed-upon number to encrypt messages. So for instance if the agreed upon shift was 3 letters to the right, the cipher would replace "a" with "d", "b" with "e" and "z" with "c". These protocols were probably secure when we didn't have computers, but in the present times, we need better ways of protecting our data and proving that they are indeed secure.

3.2 Approaches to design-level security

Within design-level security, there are two approaches:

1. **Symbolic security approach**:
   In this approach, all the messages passed in the system are considered black boxes, and there are limits placed on the number of ways the eavesdropper can interact with the system.

   Together these assumptions allow us to automate the process of verifying some of the security properties and ruling out the possibilities of certain kinds of attacks.

2. **Computational security approach**:
   In this approach, the messages are considered to be bitstrings that the adversary can interact with in any way they please.

   This model is closer to reality as an adversary can indeed do whatever they please with intercepted communication.

To illustrate the ideas, let us go back to Alice, Bob and Eve. As before, Alice wants to send a message to Bob, with Eve trying to eavesdrop and take advantage of the communication. In the symbolic model, if Eve intercepted a passing message, they can only interact with the message in a limited number of ways as prescribed by the model. In the computational model, Eve can do whatever they pleased with the message, and it would be up to the protocol designers to prove that no matter what Eve did, they wouldn't be able to learn something that they wouldn't have learnt by chance. Of course, we place reasonable restrictions on Eve's computational power in both models since every protocol is insecure in the presence of an infinitely powerful adversary.

Each of these methods has its advantages and disadvantages. The symbolic model works on the assumption that although Eve can practically do infinitely many different things, a lot of those actions will not help them in decoding the message. By limiting the interactions of Eve, the model can leverage automation. The main advantage of this model of formalising design-level security is that it can rule out different classes of attacks. Many robust tools, such as ProVerif, Tamarin etc., have been developed for proof verification in the symbolic model. Large-scale protocols like TLS 1.3 [21] and 5G

authentication key exchange protocol [19] have been formally verified using these tools. However, this model provides security guarantees more limited than the computational model.

The computational model generally involves translating rigorous pen-and-paper proofs to objects that the computer can verify. The main advantage of this approach is that it can provide stricter security guarantees at the cost of losing the automation that the symbolic model enjoys. The proofs under this model require more manual effort in general. The tooling in the computational model is also being actively developed with tools like CryptoVerif and EasyCrypt available for use. These tools have been used to verify TLS 1.3 [21] and the Amazon Web Service (AWS) key management system [22] and the SHA-3 standard [23].

After that brief introduction about the field of formal verification in general and design-level security, we are now ready to learn about EasyCrypt.

4 Introduction to EasyCrypt

As we saw earlier, the security properties of cryptosystems can be modelled as games being played by challengers (Alice, Bob) and adversaries (Eve). EasyCrypt is a proof assistant that allows us to model and verify these game-based cryptographic proofs.

4.1 EasyCrypt environment

EasyCrypt is written in OCaml and uses many external tools and libraries. However, the components that we will be interacting with the most are the following:

1. **Emacs and Proof General**:
   Emacs is an open-source text editor, and Proof General is a generic interface for proof assistants based on Emacs. Together these form the front-end for working with EasyCrypt. When working with EasyCrypt, we generally prove statements interactively using the Emacs + Proof General environment.

   Note: EasyCrypt also has a batch processing mode which doesn't need the Emacs + Proof General front-end and only requires a terminal. However, this mode is used to check multiple files and not to write proofs.

2. **External Provers and Why3**:
   Often called SMT (Satisfiability Modulo Theory) provers or SMT solvers are powerful tools that try to solve the problem of satisfiability of a formula given a set of conditions. With EasyCrypt, there is a possibility to use multiple external provers like Alt-Ergo [24], Z3 [25] and CVC4 [26], and Why3 [27] is used as the platform for all the provers.

   These provers can take care of a lot of the low-level and often gruelling work of proving some mathematical results, as we will see.

Note: Since there are a lot of external dependencies setting up the environment can be quite time-consuming. The best instructions for setting up EasyCrypt aren't on the official repository but can be found on Alley Stoughton's website [28]. We also provide a screen recording: EasyCrypt Installation Guide [6] in which we go through the instructions step-by-step and also troubleshoot some of the problems that can arise. We omit these instructions here since these can change. There are also plans to simplify the installation process with a portable application for EasyCrypt. However, until these plans materialise, we have to rely on the current method of installation.

Now without delving too much into the theory, let us start with a quick motivating example.

4.2 Abstract example: IND-RoR game

Let us model a game to prove that a cryptographic protocol possesses the property of ciphertexts being indistinguishable from random. It is often abbreviated as IND-RoR (Indistinguishability – Real or Random). The notion of indistinguishability is considered to be one of the most basic security assurances that are expected from cryptosystems.

Intuitively, the idea is that if an adversary cannot tell apart an encrypted message from encrypted randomness, then they can't obtain any information from intercepted ciphertexts that they wouldn't be able to obtain purely by chance. Hence, the system can be thought to be secure.

To model it mathematically, let us assume we have a cryptographic protocol, CP, that consists of an encryption function, Enc, and a decryption function, Dec, known to the challengers. The adversary gets access to the output of Enc. In this simplified example, we don't worry about the specifics of these functions.

So the game for IND-RoR under the protocol, CP, would proceed as follows:

1. Pick $b \in \{0, 1\}$, uniformly at random.
2. If $b = 0$, the challengers encrypt a real message using Enc.
   Let us call it $c_0 = Enc(m_{real})$
3. If $b = 1$, the challengers encrypt a random bitstring using Enc.
   Let us call it $c_1 = Enc(m_{random})$
4. If $b = 0$, send c_0 to the adversary, else send c_1
5. Adversary is allowed to perform its computations on the provided ciphertext and is expected to output b_{adv}
6. Now, the adversary is said to win the game if they have a non-negligible advantage in correctly guessing which ciphertext (real or random) they were supplied with. Mathematically, the adversary wins the game if:

$$\mathbb{P}[b_{adv} == b] = \frac{1}{2} + \epsilon, \text{ where } \epsilon \text{ is non-negligible}$$

Now, if we come up with a proof that $P[b_{adv} == b] = \frac{1}{2} + \epsilon$, where ϵ is negligible, regardless of what the adversary does, then we can claim that the protocol is secure under the IND-RoR paradigm.

Most cryptographic protocols rely on game-based proofs like these to prove the protocol's security properties. Our game, although simplified to a large extent, can already be modelled in EasyCrypt.

4.3 Modelling the IND-RoR game with EasyCrypt

Let us develop an EasyCrypt proof sketch for the game that we defined above.
Every EasyCrypt proof consists of the following major steps:

1. **Defining the objects**:
   We first need to establish the context we would like to work in. EasyCrypt comes with several predefined types, operators and functions, such as integers, real numbers, etc. Hence we begin by loading (`require`) and importing (`import`) the theories into the current environment (files that contain these definitions are called theory files or just theories) that we need.

   In this example, we will only need the `Real Bool DBool` theory files. `Real` and `Bool` are needed to work with real and boolean types, and we need the `DBool` (boolean distribution) to pick a boolean value at random. Importing these theories works like so:

   ```
   require import Real Bool DBool.
   ```

   Code block 1. Importing theory files

   Notice how the statement ends with a "."; this is how statements are terminated in EasyCrypt.

   In addition to the built-in types, we might need custom data types and operations. EasyCrypt allows us to do so using the keywords `type` and `op`.

   Let us define a `msg` type for messages, and `cip` type for ciphertexts. Then let us define the operation `enc: msg -> cip` which defines a function called `enc` mapping `msg` to `cip`, and a function called `dec` to go back from `cip` to `msg`. Additionally, let us also define a function called `comp` to model the adversary performing some computation upon receiving a `cip` and returning a `bool`.

   ```
   type msg.
   type cip.

   (* Encrypt and decrypt operations *)
   op enc: msg -> cip.
   op dec: cip -> msg.

   (* Compute operations for the adversary *)
   op comp: cip -> bool.
   ```

   Code block 2. Defining types and ops

Note that these are only abstract definitions, and we haven't specified the details of these types or functions. The interesting thing about EasyCrypt is that we can already go pretty far with the abstract types and operations.

Let us keep going and define custom module types and modules. Module types can be thought of as blueprints, while modules can be thought of as concrete instances of module types. For those familiar with object-oriented programming, this is similar to interfaces for classes and the classes that implement those interfaces. Module types are like interfaces, and since modules are also instantiations, they are closest to singleton classes.

In our example, the challengers have the ability to encrypt and decrypt messages. We can model this by creating a module type called `Challenger`, which needs to have two procedures called `encrypt` and `decrypt`. As we said, a module type is simply a blueprint. To work with the module types, we could create a concrete instance of the `Challenger` type and fill out the procedures. In our example, we create a module, `C`, of type `Challenger`. `C` has to implement the procedures `encrypt` and `decrypt`. This can be achieved like so:

```
module type Challenger = {
  proc encrypt(m:msg): cip
  proc decrypt(c:cip): msg
}.

module C:Challenger = {
 proc encrypt(m:msg): cip = {
    return enc(m);
 }
 proc decrypt(c:cip): msg = {
   return dec(c);
 }
}.

(* Similarly, we define an adversary *)
module type Adversary = {
  proc guess(c:cip): bool
}.
(* and a concrete instance of an adversary *)
module Adv:Adversary = {
  proc guess(c:cip): bool = {
    return comp(c);
  }
}.
```

Code block 3. Defining module types and modules

Note: Module types and modules need to begin with a capital letter.

We now have all the ingredients required to model the IND-RoR game outlined above. A game can be defined as a module in EasyCrypt like so:

```
module Game(C:Challenger, Adv:Adversary) = {
  proc ind_ror(): bool = {
      var m:msg;
      var c:cip;
      var b,b_adv:bool;
      b <$ {0,1}; (* Sample b uniformly at random *)
      if(b=true){
        (* Set m to be an authentic message *)
      } else {
        (* Set m to be a random string *)
      }
      c <@ C.encrypt(m);
      b_adv <@ Adv.guess(c);
      return (b_adv=b);
  }
}.
```

Code block 4. Defining a game

Here, we leave the `if` and `else` blocks empty since we don't want to introduce too much complexity in this motivating example.

2. **Making claims**:

   Once we have the objects defined, we can make claims related to these objects. We can either state these claims as axioms with the `axiom` keyword, in which case EasyCrypt will not expect a proof or a lemma with the `lemma` keyword, and EasyCrypt will expect a proof for the statement.

   For our game, a claim that we can state is that the probability of `(b_adv=b)` or the result, `res`, holding is certainly less than or equal to 1. This statement about the probability of an event is universally true. Hence we can state it as an axiom like so:

```
axiom ind_ror_pr_le1:
phoare [Game(C,Adv).ind_ror: true ==> res] <= 1%r.
```

Code block 5. Stating an axiom

This code can be read in the following way: We state the axiom, `ind_ror_pr_le1`, which says that the probability (`phoare`) of the result (`res`) holding upon running

the `ind_ror` game with `C` and `Adv` is less than or equal to 1. (The trailing `%r` after `1` is how we cast an integer to a real number.)

We will look at the components in more detail as we go along, and it should also be noted that the above result can be stated as a lemma and proved. However, the key takeaway here is that axioms don't require proofs.

Next, let us claim that the cryptosystem is indeed IND-RoR secure. Statements like these are what we intend to prove and are the main reason we go through the whole setup.

Note: We skip the detail about the negligible advantage and ϵ since this is only an illustrative example. We'd pay more attention to the details when we work with an actual protocol.

```
lemma ind_ror_secure:
phoare [Game(C,Adv).ind_ror: true ==> res] <= (1%r/2%r).
```

Code block 6. Stating a lemma

3. **Proofs**:

   Once we state a lemma, EasyCrypt expects a proof for the same. It is good practice to start a proof script with the **`proof`** keyword, but it is not strictly necessary. A proof script is a sequence of tactics that transform the goal into zero or more sub-goals. A proof is said to be complete or discharged once we get to zero sub-goals. Upon completing a proof, we end the script with **`qed`**, and EasyCrypt adds the lemma to the environment.

   Since we haven't filled out the details in our running example, we can't really make progress with the proof of **`lemma`** `ind_ror_secure`, so for this example, we will **`admit`** it. Admitting a result is akin to axiomatizing it. Ideally, we wouldn't want to axiomatize a lemma. However, we use this example to illustrate the structure of EasyCrypt proofs in general.

```
lemma ind_ror_secure:
phoare [Game(C,Adv).ind_ror: true ==> res] <= (1%r/2%r).

proof.
    admit.
qed.
```

Code block 7. Proof script

The complete code of the example can be found in `abstract-ind-ror.ec` file. Most proof scripts are developed interactively. To get a taste of how this works, you can open

`abstract-ind-ror.ec` in Emacs. This should happen automatically if you open any `.ec` file. Once you open the file, you can step through the proof line by line using the following keystrokes.

1. Ctrl + c and then Ctrl + n to evaluate one line/block of code
2. Ctrl + c and then Ctrl + u to go back one line/block of code
3. Ctrl + x Ctrl + s to save the file
4. Ctrl + x Ctrl + c to exit Emacs[2]

These four keybindings should be enough to get you through this file. We will return to more navigation and other important keybindings in the next chapter. We include these instructions in the file as well, so you don't have to keep switching back and forth. Upon evaluating the first instructions, the screen should look like so:

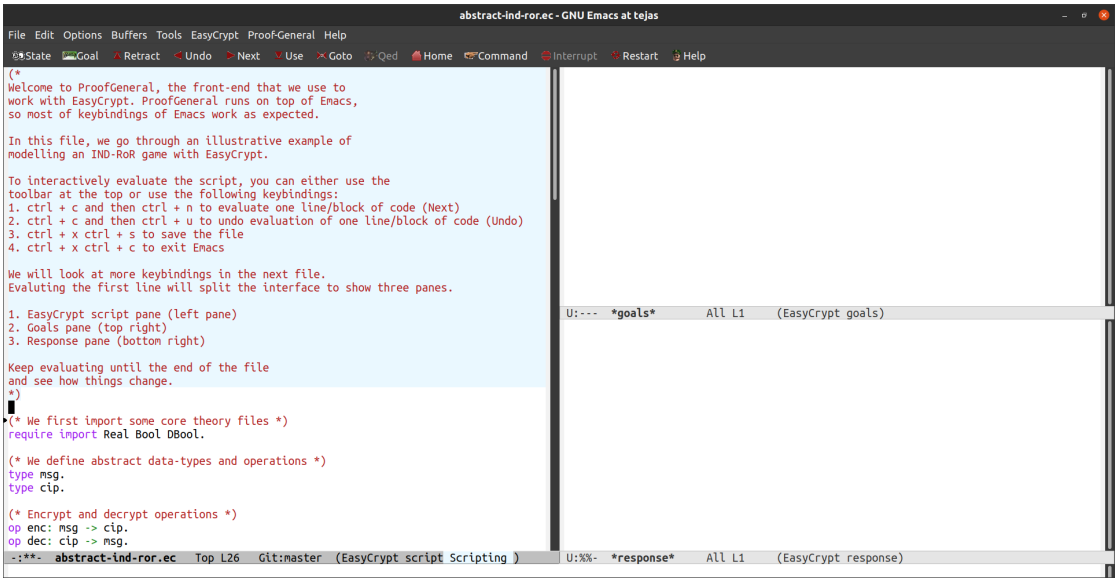

Figure 1. Working with `abstract-ind-ror.ec`

We will develop the concepts required to work with it in the following chapters. However, we encourage you to get started working with the tool and try it out right away.

[2]Emacs will prompt you to save the file if you modified it and remind you that there is an active easycrypt process running. So please pay attention to the prompts. You need to respond with a "yes" to the prompt about killing the easycrypt process to exit Emacs.

4.4 Different logics in EasyCrypt

Now that we understand that overarching structure of what we can achieve with EasyCrypt let us get down to the details.

EasyCrypt allows us to work with mathematical objects and results of different types. To work with these different results, EasyCrypt has the following logics:

1. **Ambient logic**:
   This is the higher-order logic that allows us to reason with the proof objects and terms.

2. **Hoare logic and its variants**:

   (a) **Hoare Logic (HL)**: Allows us to reason about a set of instructions or a single program.

   (b) **Relational Hoare Logic (RHL)**: Allows us to reason about a pair of programs.

   (c) **Probabilistic Hoare Logic (pHL)**: When we have a program with elements of probabilistic behaviour, we need to modify Hoare Logic to work with the elements of probability. EasyCrypt supports working with these kinds of programs as well.

   (d) **Probabilistic Relational Hoare Logic (pRHL)**: Similarly, pRHL allows us to reason with pairs of programs with probabilistic behaviour.

With these logics, EasyCrypt allows us to verify the security properties of cryptographic protocols. Most cryptographic proofs are game-based, implying that we need the ability to work with pairs of programs, as we saw earlier in the IND-RoR example. This is essentially why we need RHL and pRHL.

Apart from these logics, EasyCrypt relies on external SMT solvers to provide a fair degree of automation. SMT solvers are tools that help to determine whether a mathematical formula is satisfiable or not. We will learn to work with these logics and also how to use the external solvers in the following chapters.

5 Ambient Logic

The ambient logic of EasyCrypt is what drives all proof scripts. We will work through some examples to get a grasp of what it is. All the examples listed here can be found in the `ambient-logic.ec` file as well. It is highly recommended to work through the file using EasyCrypt in the Proof General + Emacs environment. However, simply reading through this chapter will also provide you with a working knowledge of ambient logic and the tactics that we use in this chapter.

As we saw in the motivating example earlier, formal proofs are a sequence of proof tactics, and so far, we've only seen the **`admit`** tactic. In this chapter, we will learn some more basic tactics and work with simple mathematical properties of integers.

5.1 Navigation

As noted earlier, the official Emacs tutorial covers the basics of working with Emacs. The tutorial is presented as an option on the splash screen on fresh installs of Emacs. It can also be accessed by starting Emacs and then pressing (Ctrl + h), followed by t.

All the keybindings begin either with Ctrl key, denoted by "C", or the META or Alt key, denoted by "M".

So, "C-c C-n" simply means: Ctrl + c and then Ctrl + n.

Apart from all the basic keybindings of Emacs, we have a few more bindings that are used specifically for interactive theorem proving in EasyCrypt. We list some of the most common commands here:

#	Keystroke	Command
1.	C-c C-n	Evaluate next line or block of code
2.	C-c C-u	Go back one line or block of code
3.	C-c C-ENTER	Evaluate until the cursor position
4.	C-c C-l	To reset the Proof-General layout
5.	C-c C-r	Begin evaluating from the start (Reset)
6.	C-c C-b	Evaluate until the end of the file
7.	C-x C-s	Save file
8.	C-x C-c	Exit Emacs (Pay attention to the prompts)

Table 1. Frequently used keystrokes

Most formal proofs are written interactively, and the proof-assistant, EasyCrypt in our case, will keep track of the goals (context and conclusions) for us. The front-end, Proof-

General + Emacs in our case, will show us the goals and messages from the assistant in the goals pane and response pane, respectively. Our objective is to use different tactics to prove or "discharge" the goal.

Figure 2. The different panes in EasyCrypt

5.2 Basic tactics and theorem proving

EasyCrypt has a typed expression language, so everything we declare should either explicitly have a type or be inferable from the context. As we saw earlier, EasyCrypt comes with basic built-in data types. These can be accessed by importing them into the current environment. In this file, we will be working with the `Int` theory file, and we also want EasyCrypt to print out all the goals. For this, we provide a directive to EasyCrypt using a `pragma`.

```
require import Int.
pragma Goals: printall.
```

Code block 8. Imports and pragma

Generally, the first steps in our files will be importing theories and setting the pragma.

Before we dive into cryptography, we need to understand how to direct EasyCrypt to modify the goals and make progress. As we mentioned, this is achieved by using tactics. In general, the proofs for lemmas take the following form:

```
lemma name (. . .) : (. . .) .
proof.
    tactic_1 .
```

```
    . . .
    tactic_n.
qed.
```

Code block 9. Proof script form

5.2.1 Tactic: `trivial`

To begin with, we will look at a few properties of integers and introduce how some of the tactics work.

Reflexivity is simply the property that an integer is equal to itself. Mathematically, we would write it like so:

$$\forall x \in \mathbb{Z}, x = x$$

We can express this in EasyCrypt with a lemma and prove it like so:

```
lemma int_refl: forall (x: int), x = x.
proof.
    trivial.
qed.
```

Code block 10. Using the `trivial` tactic

Once we state our lemma and evaluate it, EasyCrypt populates the goal pane with what needs to be proved. In our case, it presents the following in the goal.

```
Current goal

Type variables: <none>
----------------------------
forall (x : int), x = x
```

Code block 11. Goal upon evaluating `int_refl`

We start the proof script with the `proof`, and then EasyCrypt expects tactics to close the goal. In this case, we use `trivial`, to prove our `int_refl` lemma. Upon evaluating `trivial`, the goal pane is cleared since using this tactic closes the goal. Once there are no more goals, we can end the proof with `qed`, and EasyCrypt saves the lemma for further use and sends us this message in the response pane. Like so:

```
+ added lemma: `int_refl'
```

`trivial` tries to solve the goal using various tactics. So it can be hard to understand when to apply it, but the good news is that `trivial` never fails. It either solves the goals or leaves the goal unchanged. So you can always try it without harm.

5.2.2 Tactic: `apply`

Now EasyCrypt knows the lemma `int_refl` and allows us to use it to prove other lemmas. This can be done using the **`apply`** tactic. For instance:

```
lemma forty_two_equal: 42 = 42.
proof.
   apply int_refl.
qed.
```

Code block 12. Using the **`apply`** tactic

`apply` tries to match the conclusion of what we are applying (the proof term) with the goal's conclusion. If there is a match, it replaces the goal with the subgoals of the proof term.

In our case, EasyCrypt matches `int_refl` to the goal's conclusion, sees that it matches, and replaces the goal with what needs to be proven for `int_refl`, which is nothing, and it concludes the proof.

EasyCrypt comes with many predefined lemmas and axioms that we can use. For instance, `addzC`, and `addzA` are axioms related to commutativity and associativity for integers, respectively.

We can ask EasyCrypt to print using: **`print`** `addzC`.

EasyCrypt responds with:

```
axiom nosmt addzC: forall (x y : int), x + y = y + x.
```

5.2.3 Tactic: `simplify`

In the proofs, sometimes tactics yield something that can be simplified. We use the tactic **`simplify`** to carry out the simplification. **`simplify`** reduces the goal to a normal form using lambda calculus. We don't need to worry about the specifics of how, but it is important to understand that EasyCrypt can simplify the goals given it knows how to. It will leave the goal unchanged if the goal is already in a normal form.

For instance, here is an example that illustrates the idea.

```
lemma x_plus_comm (x: int): x + 2*3 = 6 + x.
proof.
    simplify.
    (* Simplifies the goal to x + 6 = 6 + x.  *)

    simplify.
    trivial.
    (* These make no progress, but don't fail either. *)

    apply addzC.
```

```
    (* Discharges the goal *)
qed.
```

Code block 13. Using the `simplify` tactic

5.2.4 Tactics: `move`, `rewrite`, `assumption`

So far, we saw lemmas without any assumptions except for specifying the type of a variable. More often than not, we will have assumptions regarding variables. We need to treat these assumptions as a given and introduce them into the context. This is done by using `move =>` followed by the name you want to give the assumption. Additionally, when these assumptions show up as goals, instead of applying the assumptions, we can call the tactic `assumption` to discharge the goal directly. EasyCrypt will automatically look for assumptions that match the goal when we use `assumption`.
In the following example, we use `addz_gt0`, which is this result:

```
axiom nosmt addz_gt0: forall (x y : int), 0 < x => 0 < y => 0 < x + y.
```

With that result, we get the following proof script:

```
lemma x_pos (x: int): 0 < x => 0 < x+1.
proof.
    move => x_ge0.
    (*
    Moves the assumption 0 < x into the
    context, and names it x_ge0.
    *)

    rewrite addz_gt0.
    (* Splits into two goals. *)

        (* Goal 1: 0 < x *)
        assumption.

        (* Goal 2: 0 < 1 *)
        trivial.
qed.
```

Code block 14. Proof for `x_pos`

`rewrite` "rewrites" the pattern provided, so in our case it rewrites our goal here `0 < x + 1`, with the pattern that we provided which is `addz_gt0` (`0 < x + y`), and then requires us to prove the assumptions of the pattern which are `0 < x` and `0 < 1`.

Sometimes we might have a lemma or an axiom that we can rewrite to the goal, but the LHS and RHS might be flipped. To rewrite a lemma or axiom in reverse, we simply add the "-" in front of the lemma to switch the sides like so:

```
lemma int_assoc_rev (x y z: int): x + y + z = x + (y + z).
proof.
    rewrite -addzA.
    trivial.
qed.
```

Code block 15. Rewriting in reverse

These tactics form the basics of theorem proving and working with EasyCrypt at the level of ambient logic.

A point to note here is that there are many more options and intricacies even within these simple tactics. For instance, there are many introduction patterns with the `move =>` tactic, and the keyword `move` can be replaced by other tactics as well. Presenting these introduction patterns with good examples could be an important addition to this chapter on the basic tactics.

5.2.5 Commands: `search` and `print`

Often when it comes to working with theorems, we need the ability to search through results that are already in the environment. We saw a few examples of printing in the content that we have covered so far. Let us take a slightly more detailed look at this part of EasyCrypt that we use from time to time.

The `print` command prints out the request in the response pane. We can print types, modules, operations, lemmas etc., using the print keyword. Here are some examples:

```
print op (+).
(* Response: abbrev (+)  :
int -> int -> int = CoreInt.add. *)
print op min.
(* Response: op min (a b : int) : int
    = if a < b then a else b. *)
print axiom Int.fold0.
(* Response: lemma fold0 ['a]:
    forall (f : 'a -> 'a) (a : 'a), fold f a 0 = a. *)
```

Code block 16. Using the `print` command

The keywords simply act as qualifiers and filters. You can print even without those. The qualifiers simply help us to narrow the results.

The `search` command allows us to search for axioms and lemmas involving a list of operators. It accepts arguments enclosed in the following braces:

1. [ ] - Square brackets for unary operators

2. ( ) - Round brackets for binary operators
3. Names of operators
4. Combination of these separated by a space

```
search [-].
search (+).
(* Shows lemmas and axioms that
include the specified operators. *)
search ( * ).
(* Notice the extra space for the "*" operator.
We need that since (* *) also indicates comments. *)

search min.
(* Shows lemmas and axioms that
include the operator "min". *)

search (+) (=) (=>).
(* Shows lemmas and axioms which have
all the listed operators *)
```

Code block 17. Using the `search` command

In the file, we have peppered some exercises that require the reader to search for specific results to make progress.

5.2.6 External solvers: `smt`

An important point to understand is that EC was built to work with cryptographic properties and more complex things. So although general mathematical theorems and claims can be proven in EC, it will be quite painful to do so. We will employ powerful automated tools to take care of some of these low-level tactics and logic. EC offers this in the form of the `smt` tactic. When we run `smt`, EC sends the conclusion and the context to external smt solvers like `Z3`, `Alt-Ergo` etc., that have been configured to be used by EasyCrypt. If they can solve the goal, then `smt` will discharge the specific sub-goal that it was invoked on. If not, `smt` fails, and the burden of the proof is still on us.
For example, if we wish to prove the result:

$$\forall x \in \mathbb{R}, \forall a, b \in \mathbb{Z}, \text{ and } x \neq 0 \implies x^a * x^b = x^{a+b}$$

We would do it like so:

```
lemma exp_product (x: real) (a b: int):
    x <> 0%r
    => x^a * x^b = x^(a + b).
proof.
    move => x_pos.
    rewrite -RField.exprD.
    assumption.
    trivial.
qed.
```

Code block 18. Manual proof for `exp_product`

However, it can be simplified to:

```
lemma exp_product_smt (x: real) (a b: int):
    x <> 0%r
    => x^a * x^b = x^(a + b).
proof.
    smt.
qed.
```

Code block 19. Using `smt` to prove `exp_product`

The key takeaway is that we will rely on external solvers to do a fair amount of heavy lifting when it comes to results related to low-level math.

That concludes this chapter on ambient logic. In this chapter, we covered the basic tactics like `apply`, `simplify`, `move`, `rewrite` etc., we also saw how to use the `search` and `print` commands and learnt to work with external solvers. We have also provided exercises that help the reader practice using these tactics. We will introduce more tactics and other techniques as we progress in the subsequent chapters.

6 Hoare Logic

Working through Chapter 5 should give you a good grasp of the ambient logic and tactics for reasoning with simple math. Up until now, we were working with mathematical proofs that only used logical reasoning. However, when working with programs and procedures, we need a way to reason with what the programs do.

For instance, let us think about an exponentiation program for integers like so:

```
exp(x, n):
    r = 1
    i = 0
    while (i < n):
        r = r * x
        i = i + 1
    return r
```

Code block 20. Pseudo code for exponentiation

When presented with a program like this, our objective is to figure out if the program behaves correctly. At first glance, this program seems correct. However, a glaring mistake here is that the program will always return 1 as a result if we pass any negative integer as the second argument. That isn't the behaviour we expect from an exponentiation function. So saying that the program is correct would be a false claim. So to make claims about the behaviour of the program, mathematically, we would say something like:

$$\text{Given } \underbrace{x \in \mathbb{Z}, n \in \mathbb{Z} \;\&\; n \geq 0}_{\text{pre-condition}}, \underbrace{exp(x, n)}_{\text{program}} \text{ returns } \underbrace{r = x^n}_{\text{post-condition}}$$

6.1 The Hoare triple

As marked in the statement above, claims that we make generally have three distinct parts: preconditions, the program and postconditions. Hoare logic formalises these three parts and introduces them as a **Hoare triple**.

A Hoare triple is denoted like so:

$$\{P\}\, C\, \{Q\}$$

Here P and Q are conditions on the program variables used in C. Conditions on program variables are written using standard mathematical notations together with logical operators like $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not') and $\implies$ ('implies'). Additionally, we have special conditions $true$, which always holds and $false$ which never holds.

C is a program in some specified language.

We say that a Hoare triple, $\{P\}\, C\, \{Q\}$, holds if whenever C is executed from a state satisfying P and if the execution of C terminates, then the state in which C's execution terminates satisfies Q. We will limit our discussion to programs which terminate.

6.1.1 Examples

1. $\{x = n\}\, x := x + 1\, \{x = n + 1\}$ holds. ($:=$ is the assignment operator)

   In this triple, we can observe two kinds of variables. n is called an ambient variable, and it comes from the context that we have. Whereas x is called a program variable since the program acts upon this variable.

2. $\{x = n\}\, x := x + 1\, \{x = n + 2\}$ doesn't hold.

3. $\{true\}\, C\, \{Q\}$ is a triple in which the precondition always holds. So we'd say that this triple holds for every C that satisfies the postcondition Q.

4. $\{P\}\, C\, \{true\}$, similarly, this triple holds for every precondition P that is satisfied, and every program C.

5. $\{false\}\, C\, \{Q\}$, is an interesting triple which, according to our definitions, doesn't hold since false is a statement that never holds. However, this is a slightly special case, as we will see in EasyCrypt.

6.1.2 Exercises

1. Does $\{x = 1\}\, x := x + 2\, \{x = 3\}$ hold?
2. How about $\{true\}\ exp(x, a)\ \{r = x^a\}$? Why?
3. What about $\{2 = 3\}\, x := 1\, \{x = 1\}$?

6.2 Strength of statements

Informally, if a statement can be deduced from another, then the statement that was deduced is a weaker statement.

Mathematically if we have $P \implies Q$, we say that P is a stronger statement than Q. For example,

$$x = 5 \implies x \geq 5$$

$x = 5$ is a stronger statement than $x \geq 5$

As discussed earlier, we have two special statements, $true$, which always holds, and $false$, which never holds. These are the weakest and the strongest statements, respectively.

Since any statement that holds can imply that $true$ holds. The reason is that it always holds, $true$ is the weakest statement there is.

Similarly, $false$ never holds. So, no statement can imply false; hence, it is the strongest statement there is.

6.3 Proof trees

So far, we looked at Hoare triples for simple statements, these can be thought of "programs" with a single instruction as well. However, we need to be able to work with multiple statements and more complex statements. We achieve this by formalising a set of axioms that we believe to be true or that follow from the definition of the semantics of the programming language. We then combine these to make claims about complex statements. We often visualise this series of steps in which we put the axioms together into *schemas* or *proof trees*.

For example: For a statement S, we say it is or provable by denoting it with a turnstile ($\vdash$) symbol like so $\vdash S$, and its proof can be denoted by:

$$\frac{\vdash S_1, \vdash S_2, \ldots, \vdash S_n}{\vdash S}$$

This says the conclusion S may be deduced from the $S_1, \ldots, S_n$ which are the hypotheses of the rule. The hypotheses can either all be theorems or axioms of Hoare Logic or mathematics.

6.4 Axioms of Hoare Logic

We will now take a look at some of the axioms of Hoare logic with examples to give you a flavour of how they work. We cover these because these axioms form the basis for the tactics we use in Hoare logic in EasyCrypt. The goal here is to simply familiarise the reader with the basics and then let the machine take care of the specifics.

1. **Assignment**:

   $$\vdash \{P[E/V]\}\, V := E\, \{P\}$$

   Where V is any variable, E is any expression, P is any statement, and the notation $P[E/V]$ denotes the result of substituting the term E for all occurrences of the variable V in the statement P.

   Example: $\vdash \{y = 5\}\, x := 5\, \{y = x\}$

2. **Precondition strengthening**:

When we have a Hoare triple $\{P'\}\, C\, \{Q\}$, where P' is a statement that follows from a stronger statement, P. Then we can say,

$$\frac{\vdash P \implies P', \vdash \{P'\}\, C\, \{Q\}}{\vdash \{P\}\, C\, \{Q\}}$$

Example: Let

$$C = [x := x + 2]$$

$$P = \{x = 5\}$$

$$P' = \{x \geq 5\} \text{ , and}$$

$$Q = \{x \geq 7\}$$

Using the precondition strengthening axiom we have,

$$\frac{\vdash \{x = 5\} \implies \{x \geq 5\}, \vdash \{x \geq 5\}\, x := x + 2\, \{x \geq 7\}}{\vdash \{x = 5\}\, x := x + 2\, \{x \geq 7\}}$$

A point to note here is that weaker preconditions imply stronger Hoare judgements (triples). To make this intuition clear, consider the triples:

$$\{true\}\, x := 5\, \{x = 5\} \text{ and } \{x = 3\}\, x := 5\, \{x = 5\}$$

The first triple's precondition is the weakest precondition that we can have and a statement that always holds. This triple holds, and it implies that no matter the starting state executing $x := 5$ yields $\{x = 5\}$ as a postcondition.

While the second triple only holds when the condition $\{x = 3\}$ is satisfied to begin with. This is fairly specific compared to $true$ of the previous triple and is only satisfied when x is, in fact, 3 to begin with.

Hence, we say that the first Hoare judgement is stronger when compared to the second one.

3. **Postcondition weakening**:

   Similarly, when we have a Hoare triple $\{P\}\, C\, \{Q'\}$, where Q' is a strong statement, and if Q follows from Q'. Then we can say,

$$\frac{\vdash \{P\}\, C\, \{Q'\}, \vdash Q' \implies Q}{\vdash \{P\}\, C\, \{Q\}}$$

   Example: Let

$$C = [x := x + 2]$$

$$P = \{x = 5\}$$

$$Q' = \{x = 7\} \text{ , and}$$
$$Q = \{x \geq 7\}$$

With the postcondition weakening axiom, we have,

$$\frac{\vdash \{x = 5\}\, x := x + 2\, \{x = 7\}, \vdash x = 7 \implies x \geq 7}{\vdash \{x = 5\}\, x := x + 2\, \{x \geq 7\}}$$

Together the precondition strengthening and postcondition weakening axioms are known as the **consequence rules**.

4. **Sequencing**:

   For two programs C_1, C_2, we have the following:

$$\frac{\vdash \{P\}\, C_1\, \{Q'\}, \vdash \{Q'\}\, C_2\, \{Q\},}{\vdash \{P\}\, C_1; C_2\, \{Q\}}$$

   Example: Let

$$C_1 = [x := x + 2]$$
$$C_2 = [x := x * 2]$$
$$P = \{x = 5\}$$
$$Q' = \{x = 7\} \text{ , and}$$
$$Q = \{x = 14\}$$

   Using the sequencing axiom, we have,

$$\frac{\vdash \{x = 5\}\, x := x + 2\, \{x = 7\}, \vdash \{x = 7\}\, x := x * 2\, \{x = 14\}}{\vdash \{x = 5\}\, x := x + 2, x := x * 2\, \{x = 14\}}$$

We went through these examples to get a sense of what formal proof trees look like and the theory that formal verification is based on. The proof trees that we've used are already simplified to exclude the assignment axiom and steps that we as humans can easily understand and gloss over. Proof trees get quite large and unwieldy as soon as we do anything non-trivial. This is exactly where formal verification tools come into the picture. So, let us now switch to EasyCrypt and work with Hoare triples.

Note: Hoare logic has been studied quite extensively, and there are plenty of good textbooks [29, 30] that one can refer to for mathematical rigour and completeness. The objective here is to give the reader an intuitive understanding of the math and enough working knowledge required to work with EasyCrypt.

6.5 HL in EasyCrypt

With a basic understanding of HL, we can now proceed to work with it in EasyCrypt. The examples we present here are all included in the `hoare-logic.ec` file. As before, it is recommended to work with the file in the Proof General + Emacs environment.

6.5.1 Basic Hoare triples

Let us start small and first work with some examples that we saw earlier. We first define a module to define two procedures for the programs.

```
module Func1 = {
  proc add_1 (x:int) : int = { return x+1; }
  proc add_2 (x: int) : int = { x <- x + 2; return x; }
}.
```

Code block 21. Function definitions (`Func1`)

A Hoare triple denoted by $\{P\}\, C\, \{Q\}$ on paper is expressed as `hoare [C : P ==> Q ]` in EasyCrypt, with the usual definitions. Additionally, the return value of the program, C, is stored in a special keyword called `res`.

So the triple $\{x = 1\}$ `Func1.add_1` $\{x = 2\}$ would be expressed in EasyCrypt like so:

```
lemma triple1: hoare [ Func1.add_1 : x = 1 ==> res = 2].
```

When working with Hoare logic or its variants, the goal will be different from what a goal in ambient logic looks like. For instance, evaluating the `lemma triple1` produces the following goal.

```
Current goal
Type variables: <none>
----------------------------
pre = x = 1

    Func1.add_1

post = res = 2
```

Code block 22. Goal upon evaluating `triple1`

To make progress here, we first need to tell EasyCrypt what `Func1.add_1` is. The way to do that is by using the `proc` tactic. It simply fills in the definitions of the procedures that we define. Since `Func1.add_1` is made up of only a return statement, `proc` replaces `res` with the return value. This leaves us with an empty program.

When working with Hoare judgements and programs, we work through how the program statements change the preconditions and postconditions according to axioms and lemmas that we have. Our goal is to get to a state where we have consumed all the program statements. Once we have consumed all the program statements, we can transform the goal from an HL goal to a goal in ambient logic using the **skip** tactic.

The **skip** does the following:

$$\frac{P \implies Q}{\{P\}\ \text{skip;}\ \{Q\}}\texttt{skip}$$

skip; signifies an empty program, while `skip` is the tactic itself

This puts us back in the familiar territory of ambient logic, and we can use all the tactics that we learnt in Chapter 5. The only difference is that transitioning a goal from Hoare logic to ambient logic introduces some qualifiers about the memory that the program works on. Hence, we need to handle those as well. In this example, the goal after evaluating **skip** will simply read: `forall &hr, x{hr} = 1 => x{hr} + 1 = 2`, where `&hr` refers to a memory, and `x{hr}` to the context of `x` in `&hr`. The proof for which follows pretty trivially. The only difference is that we need to move the memory into the assumption by prepending the & character in the **move** `=>` tactic.
So the full proof for this simple example looks like so:

```
lemma triple1: hoare [ Func1.add_1 : x = 1 ==> res = 2].
proof.
    proc.
    skip.
    move => &m H1. (* &m moves memory to the environment *)
    subst. (* Substitutes variables from the assumptions *)
    trivial.
qed.
```

Code block 23. Proof of `triple1`

Now let us work with a program where the body is not empty.

```
lemma triple2: hoare [ Func1.add_2 : x = 1 ==> res = 3 ].
```

Using **proc** produces the following goal.

```
Current goal
Type variables: <none>
----------------------------
Context : Func1.add_2

pre = x = 1
```

```
(1)  x <- x + 2

post = x = 3
```

Code block 24. Goal upon evaluating `triple2`

When we are faced with $\{P\}S1; S2; S3; \{Q\}$ with the usual definitions, applying the **wp** tactic consumes as many ordinary statements as possible from the end of the program and computes a precondition R. This is chosen such that it breaks down the original judgement into two judgements which hold by the **seq** rule. The first judgement has the original precondition P, the unconsumed statements of the program, and R as the postcondition, while the second judgement has R as the precondition, the consumed statements of the program, and Q as the postcondition. R is chosen to be as weak as possible as well in order to have a strong second judgement.

This might be confusing to read but easy to understand with the help of a proof tree.

For instance, when we have $\{P\}S1; S2; S3; \{Q\}$ and $S2; S3;$ are statements that can be dealt with some axioms or rules, then **wp** does the following:

$$\frac{\{P\}S1; \{R\} \quad \frac{\cdots}{\{R\}S2; S3; \{Q\}}\,\text{(Other rules)}}{\{P\}S1; S2; S3; \{Q\}}\,\texttt{seq}$$

The judgement $\{R\}S2; S3; \{Q\}$ is guaranteed to hold, and hence the goal transforms to just $\{P\}S1; \{R\}$.

In our context, **wp** consumes the only instruction that we have in the program and produces the judgement $\{x = 1\}$ skip; $\{x + 2 = 3\}$ to which we apply **skip**. This gives us the following proof tree:

$$\frac{\dfrac{x = 1 \implies x + 2 = 3}{\{x = 1\}\ \text{skip};\ \{x + 2 = 3\}}\,\texttt{skip}}{\{x = 1\}\ x := x + 2\ \{x = 3\}}\,\texttt{wp}$$

Then we continue the proof as before and get the following proof:

```
lemma triple2: hoare [ Func1.add_2 : x = 1 ==> res = 3 ].
proof.
    proc.
    wp. (* The only new addition *)
    skip.
    move => &m x.
    subst.
    trivial.
qed.
```

Code block 25. Proof of `triple2`

6.5.2 Automation, and special cases

Now that we have an understanding of how we can make progress, let us take a look at how we can use automation since we still have powerful external solvers at our disposal, and we would like to use them whenever we can. We will be working with the following procedures in this section.

```
module Func2 = {
  proc x_sq (x:int) : int = { return x*x; }
  proc x_0  (x:int) : int = { x <- x*x; x <- x-x; return x; }
  proc x_15 (x:int) : int = { x <- 15; return x; }
}.
```

Code block 26. Function definitions (`Func2`)

For instance, let us take a look at a judgement, which states that if you square any integer greater than or equal to 4, the result is greater than or equal to 16. Pretty trivial and straightforward when you think about it. However, the proof for something simple like this becomes quite tiresome, hence we will simply ask `smt` to handle it. The only issue, however, is that smt solvers can only work on goals in ambient logic. So it is up to us to bring the goal to a state that doesn't involve Hoare logic. In this example, since `x_sq` consists of a single `return` statement, `proc` and `skip` are enough.

```
lemma triple3: hoare [ Func2.x_sq : 4 <= x ==> 16 <= res ].
proof.
    proc.
    skip.
    smt.
qed.
```

Code block 27. Proof for `triple3`

Let us now look at the triple $\{\texttt{false}\}\, x := 15\, \{x = 0\}$. Theoretically, we know that this judgement doesn't hold since `false` never holds. We have the `proc x_15` in the `module Func2` that we can use to express that triple in EasyCrypt. The interesting thing is that we can actually write proof for the triple in question.

```
lemma triple4: hoare [ Func2.x_15 : false ==> res=0 ].
proof.
    proc.
    wp.
    skip.
    move => _ f.
    trivial.
qed.
```

Code block 28. Proof for `triple4`

The reason we can do this is that we essentially force the assumption that `false` holds, and we want to prove the postcondition $15 = 0$. As absurd as it is, we know that `false` is the strongest statement there is. By getting EasyCrypt to the state where `false` holds would imply that anything and everything can be derived from it. Hence we can actually "prove" that $15 = 0$.

The point to understand here is that we could only do this because we moved `false` into the context manually when we used the `move =>`. So our math is still consistent, and the world hasn't exploded yet. The way to think about this triple is assuming `false` holds implies that $15 = 0$.

6.5.3 Conditionals and loops

Let us now work with some more interesting functions and triples. We define a flipper function which simply returns the opposite of the boolean value that it gets.

```
module Flip = {

proc flipper (x: bool) : bool =
  {
  var r: bool;
  if (x = true)
  { r <- false; }
  else
  { r <- true; }
  return r;
  }
}.
```

Code block 29. Definition of `Flip` module

Let us say that we would like to prove the fact related to this program, that if the input is `true`, `Flip.flipper` returns `false`.

We use a slightly verbose proof here to demonstrate how to open up an `if` block. Using the `if` tactic in the proof script gives us two goals, one in which the `if` condition holds, and another in which it doesn't. In our case, it splits into one goal with `x = true`, and another with `x <> true`. Additionally, when the current goal is an HL, a pHL, or a pRHL statement, the `auto` tactic uses various tactics in an attempt to reduce the goal to a simpler one automatically. It never fails, but it may fail to make any progress. For instance, in this first usage of the tactic, it does the job of the `wp`, `skip`, and `trivial` tactics.

```
lemma flipper_correct_t:
    hoare [ Flip.flipper : x = true ==> res = false ].
proof.
```

```
proc.
if.
  (* Goal 1:  x = true *)
  auto.

  (* Goal 2: x <> true. *)
  auto.
  smt.
qed.
```

Code block 30. Proof for `flipper_correct_t`

Notice the repetition of proof steps in the branches. This can be reduced by using tacticals. In order to tell EasyCrypt to repeatedly use certain tactics on all resulting new goals, we use the ";" tactical. So, we can simplify the above proof like so

```
lemma flipper_correct_f:
    hoare [ Flip.flipper : x = false ==> res = true ].
proof.
    proc.
    if; auto; smt.
qed.
```

Code block 31. Proof for `flipper_correct_f`

However, since this program is quite simple, we can actually make the proof shorter. We can also make the logic more abstract like so:

```
lemma flipper_correct (x0: bool):
    hoare [ Flip.flipper : x = x0  ==> res <> x0 ].
proof.
  proc.
  auto.
  smt.
qed.
```

Code block 32. Proof for `flipper_correct`

This is how proofs are polished and made shorter. We first write a verbose proof, then keep experimenting to find shorter and more elegant proofs.
Let us now increase the difficulty and work with a slightly more involved example. We define the exponentiation function that we saw earlier in EasyCrypt.

```
module Exp = {
    proc exp (x n: int) : int =
      {
      var r, i;
```

```
      r <- 1;
      i <- 0;
      while (i < n){
        r <- r*x;
        i <- i+1;
      }
      return r;
      }
}.
```

Code block 33. Definition of `Exp` module

Let us formulate a Hoare triple that says that `exp(10, 2) = 100`, since of course $10^2 = 100$.

We would have the triple $\{x = 10 \wedge n = 2\}$ `Exp.exp` $\{$`res` $= 100\}$. In EasyCrypt we would state the lemma as we have done earlier. For the proof, we will employ loop unrolling. To unroll a loop with **`unroll`** `n`, where `n` is the line of code with the loop statement, a **`while`** loop in our case. We show the intermediate state of the proof and the goal state after unrolling the loop once.

```
lemma ten_to_two:
    hoare [ Exp.exp : x = 10 /\ n = 2 ==> res = 100 ].
proof.
    proc.
    simplify. (* Makes the goal more readable *)
    unroll 3.
    ...
```

Code block 34. Unrolling a loop

The goal right after **`simplify`**

```
Current goal
Type variables: <none>
----------------------------
Context : Exp.exp
pre = x = 10 /\ n = 2

(1--)  r <- 1
(2--)  i <- 0
(3--)  while (i < n) {
(3.1)    r <- r * x
(3.2)    i <- i + 1
(3--)  }

post = r = 100
```

Code block 35. Goal before `unroll`

As we can see the while loop is on line 3, and hence we use `unroll 3` to unroll it. That gives us the following goal state:

```
Current goal
Type variables: <none>
----------------------------
Context : Exp.exp
pre = x = 10 /\ n = 2

(1--)  r <- 1
(2--)  i <- 0
(3--)  if (i < n) {
(3.1)    r <- r * x
(3.2)    i <- i + 1
(3--)  }
(4--)  while (i < n) {
(4.1)    r <- r * x
(4.2)    i <- i + 1
(4--)  }

post = r = 100
```

Code block 36. Goal after using `unroll`

We adopt this method since we know that the `while` loop will be executed twice, we unroll the loop twice. The second time, the loop is on line 4. So we use `unroll 4`. With the loops unrolled, we get two `if` conditions which we know will hold, and a `while` loop for which the condition will not hold. To reason with loops and conditions like these EasyCrypt provides two tactics `rcondt`, and `rcondf`. They can be read as "remove the condition with a true/false assignment".

In our case, the loop we want to target is the `while` loop at line 5 (unrolling again pushes the loop one line lower). So we use `rcondf 5`, which produces two goals. The first goal keeps the program up until the loop and asks us to prove that the boolean in the `while` block doesn't, giving us the postcondition `!(i<n)`. The second goal will remove the `while` loop and preserve our original postcondition (`r = 100`).

```
lemma ten_to_two:
    hoare [ Exp.exp : x = 10 /\ n = 2 ==> res = 100 ].
proof.
    proc.
    simplify. (* Makes the goal more readable *)
    unroll 3.
```

```
unroll 4.
rcondf 5.
...
```

Code block 37. Removing a condition

Produces the goals:

```
Current goal (remaining: 2)
Type variables: <none>
------------------------------
Context : Exp.exp
pre = x = 10 /\ n = 2

(1--)  r <- 1
(2--)  i <- 0
(3--)  if (i < n) {
(3.1)    r <- r * x
(3.2)    i <- i + 1
(3--)  }
(4--)  if (i < n) {
(4.1)    r <- r * x
(4.2)    i <- i + 1
(4--)  }

post = ! i < n

Goal 2
------------------------------
Context : Exp.exp
pre = x = 10 /\ n = 2

(1--)  r <- 1
(2--)  i <- 0
(3--)  if (i < n) {
(3.1)    r <- r * x
(3.2)    i <- i + 1
(3--)  }
(4--)  if (i < n) {
(4.1)    r <- r * x
(4.2)    i <- i + 1
(4--)  }

post = r = 100
```

Code block 38. Goal after using `rcondf`

The proof for these goals work out to be fairly easy, giving us the following complete proof.

```
lemma ten_to_two:
    hoare [ Exp.exp : x = 10 /\ n = 2 ==> res = 100 ].
proof.
    proc.
    simplify. (* Makes the goal more readable *)
    unroll 3.
    unroll 4.
    rcondf 5.
    (* post = !(i<n) *)
    wp.
    skip.
    smt.
    (* post = r = 100  *)
    wp.
    skip.
    smt.
qed.
```

Code block 39. Complete proof of `ten_to_two`

As usual, we could have used some tacticals to shorten the proof. So let us do that, to clean up the previous proof.

```
lemma ten_to_two_clean:
    hoare [ Exp.exp : x = 10 /\ n = 2 ==> res = 100 ].
proof.
    proc.
    unroll 3.
    unroll 4.
    rcondf 5; auto.
qed.
```

Code block 40. Cleaned up proof for `ten_to_two`

For a loop that unrolls twice, it is easy to do it manually. However, this strategy wouldn't work for a different scenario. For instance, in order to prove that the program works correctly, we need to prove the correctness for every number, so we would prefer to work with abstract symbols and not concrete numbers like 10^2. In order to work up to it, let us try to prove that 2^{10} works as intended. But first, we need to understand that EasyCrypt was not built for computations. It can handle small calculations as we've seen

so far but asking EasyCrypt to do 2^{10} doesn't work as we'd like it to. For instance, take a look at the following lemma and our attempt to prove it.

```
lemma twototen: 2^10 = 1024.
proof.
    (* We can't make any progress with these. *)
    trivial.
    simplify.
    auto.
    (*
    smt fails as well, we will admit this result,
    since we know it is true.
    *)
    admit.
qed.
```

Code block 41. Admitting $2^{10} = 1024$

Again, the point here is that EasyCrypt wasn't built for tasks like these. For the time being, we will admit this lemma, since we know that 2^{10} is in fact 1024. We need this to prove the next few lemmas relating to Hoare triples.

At this point, we'd like to prove that `exp(2,10)` works as expected, however, we can't do so with direct computation since it would be painful and also not the purpose of using EasyCrypt, so to reason with loops, we need to be able to think of loop invariants and think about the program variables which change. For instance, we know that the variable `x` remains the same throughout the computation. So let us try to get rid of the `while` loop stating that this is the only invariant. We know that this is obviously not enough, since doing this will in a sense forget about what happens to the other variables. However, examples that get stuck are instructive as well as they allow us to understand what we did wrong. The following proof reaches a point in which the postcondition states:

```
post = ··· forall (i0 r0 : int), ! i0 < n => x = 2 => r0 = 1024
```

This can't hold since it states that the result `r0` is 1024 for every `r0`, hence we abort this attempt.

```
lemma two_to_ten:
    hoare [ Exp.exp : x = 2 /\ n = 10 ==> res = 1024 ].
proof.
    proc.
    simplify.
    while ( x = 2 ).
    wp.
    auto.
```

```
abort.
```

Code block 42. A failed attempt to prove `two_to_ten`

Similarly, we try another invariant which helps, but still gets stuck since it doesn't account for how the variable `r` changes after every iteration of the loop. We abort this attempt as well.

```
lemma two_to_ten:
    hoare [ Exp.exp : x = 2 /\ n = 10 ==> res = 1024 ].
proof.
    proc.
    while ( x = 2  /\ 0 <= i <= n).
    wp.
    auto.
    smt.
abort.
```

Code block 43. Another failed attempt to prove `two_to_ten`

We know that after every iteration, the variable `r` is multiplied by `x`. So in this case, since we have `x = 2`, essentially at the end of `i` iterations of the loop we have the fact that $r = 2^i$. This is an invariant, and it binds r to the variables that are passed to the loop. With this, we finally have all the ingredients for the invariant, and we complete the proof, like so:

```
lemma two_to_ten:
    hoare [ Exp.exp : x = 2 /\ n = 10 ==> res = 1024 ].
proof.
    proc.
    while (x = 2  /\ 0 <= i <= n /\ r = 2^i).

    wp.
    skip.
    smt.

    wp.
    simplify.
    auto.

    (*
    Sometimes, the goal can be quite complicated, and we
    can use "progress" to break it down into smaller goals
    *)
    progress.
```

```
    (* 2 ^ 0 = 1 *)
    smt.

    (* 2^10 = 1024 *)
    smt. (* Uses the axiom twototen from above *)
qed.
```

Code block 44. Full proof for `two_to_ten`

Finally, we have an invariant that works. Let us clean up the proof, and also if we think about it, the condition `x=2` isn't really needed, since the program never modifies the value of `x`. Let us get rid of that condition while we are at it.

```
lemma two_to_ten_clean: hoare [ Exp.exp : x = 2 /\ n = 10 ==>
     res = 1024 ].
proof.
    proc.
    simplify.
    while ( r = x^i /\ 0 <= i <= n); auto; smt.
qed.
```

Code block 45. Cleaned up proof for `two_to_ten`

Now the proof seems so innocuous and straightforward. However, it is important to understand that these proofs and figuring out the loop invariants always takes a few tries, and sometimes crafting the right invariant can be an art by itself. This also gets quite hard when there are a lot of variables to keep track of. So it is good practice to work with smaller examples first.

Now let us try to work with abstract symbols, the stuff that EC was actually built for. In order to claim that the `exp` function is correct, we need to have the condition that the exponent that we provide is greater than zero. We use `x0`, and `n0` for the ambient variables, in order to differentiate from the program variables `x, n`.

```
lemma x0_to_n0_correct (x0 n0: int):
  0 <= n0 =>
  hoare [ Exp.exp : x = x0 /\ n = n0 ==> res = x0 ^ n0 ].
proof.
    move => Hn0.
    proc.
    while (r=x^i /\ 0 <= i <= n).
    wp.
    skip.
    smt.

    wp.
```

```
    skip.
    progress.
    smt.
    smt.
qed.
```

Code block 46. Proof for the correctness of `Exp.exp`

Again, we can clean up the proof like so:

```
lemma x0_to_n0_correct_clean x0 n0:
  0 <= n0 =>
  hoare [ Exp.exp : x = x0 /\ n = n0 ==> res = x0 ^ n0 ].
proof.
    move => Hn0.
    proc.
    while (r=x^i /\ 0 <= i <= n); auto; smt.
qed.
```

Code block 47. Cleaned up proof for the correctness of `Exp.exp`

With that, we conclude this chapter on Hoare logic. In this chapter, we first presented the theory of Hoare logic, and we saw how to work with HL in EasyCrypt. Starting with simple Hoare triples we worked our way up to reasoning with more advanced Hoare triples, and along the way, we learnt some new tactics that allowed us to work with the HL goals.

7 Relational Hoare Logic

Hoare Logic enables us to reason about programs and build formal proof trees, but it doesn't have any way to work with pairs of programs.

For instance, let us say we build a compiler which optimises some user-written code. At the very least, the users would expect the compiler to preserve program behaviour across optimisations. So, we would like to provide some assurance to the users about how the compiled and optimised code still performs the same tasks. To do this, we need to relate the two programs (user written and optimised) and conclude that given the same initial conditions executing both the programs yield the same final state. However, there is no straightforward way to do this with classical Hoare Logic.

To address the deficiency discussed above, a simple variation of classical Hoare logic called **Relational Hoare Logic** (RHL) was conceived. RHL allows us to make judgements about two programs with the introduction of a **Hoare quadruple**. A Hoare quadruple is denoted like so:

$$\{P\}\, C \sim D\, \{Q\}$$

Here as with classical Hoare logic P and Q, are conditions on the program variables used in C and D. The only difference is that we now have to understand that C and D act on two different memories. The conditions P and Q have to take this into account. The rest of the ideas carry over quite naturally from classical Hoare logic.

We say that a Hoare quadruple, $\{P\}\, C \sim D\, \{Q\}$, holds, if whenever C and D are executed from a state satisfying P and upon the execution and termination of C and D, the resultant state satisfies Q.

The axioms and rules from classical Hoare logic are also modified slightly to work with quadruples instead of triples.

For instance, consider the sequencing axiom from HL (the symbols have their usual meanings):

$$\frac{\vdash \{P\}\, C_1\, \{Q'\}, \vdash \{Q'\}\, C_2\, \{Q\},}{\vdash \{P\}\, C_1; C_2\, \{Q\}}$$

In RHL, we modify it to work with quadruples like so:

$$\frac{\vdash \{P\}\, C_1 \sim D_1\, \{Q'\}, \vdash \{Q'\}\, C_2 \sim D_2\, \{Q\},}{\vdash \{P\}\, C_1; C_2 \sim D_1; D_2\, \{Q\}}$$

This is a recurring theme when it comes to discussing Hoare logic and its variations. Each variation addresses a deficiency that the previous version has, and we update or add axioms to work with the modifications that we introduced. We will rely on EasyCrypt to take care of the details.

7.1 RHL in EasyCrypt

Now to get a little practice with RHL, let us switch to EasyCrypt. As usual, we start with some simple exercises.

7.1.1 Basic Hoare quadruples

We begin with two functions, `swap1` and `swap2`, which swap variables. However, the way they swap variables is slightly different, and we'd like to establish the fact that they accomplish the same task. We define `swap1` and `swap2` like so:

```
module Jumbled = {
  proc swap1 (x y: int) : int*int = {
  var z;
  z <- x;
  x <- y;
  y <- z;
  return (x,y);
  }

  proc swap2 (x y: int) : int*int = {
  var z;
  z <- y;
  y <- x;
  x <- z;
  return (x,y);
  }
}.
```

Code block 48. Definition of the module `Jumbled`

In both functions, `z` is the temporary variable. In `swap1`, `x` is first stored in `z`, while in `swap2` `y` is stored first. However, both the functions accomplish the task of swapping variables.

A RHL quadruple, $\{P\}\, C_1 \sim C_2\, \{Q\}$, is expressed in EasyCrypt with the statement `equiv [C1 ~ C2 : P ==> Q]`. As with Hoare triples, we access the results of both the programs using the `res` keyword. However, since we now have two programs, we need to add identifiers to the variables that we use and also to the results to convey the program that we are speaking about. For instance, to prove that `swap1` is equivalent to itself, we would have the following lemma.

```
lemma swap1_equiv_swap1:
equiv [Jumbled.swap1 ~ Jumbled.swap1 : x{1}=x{2} /\ y{1}=y{2}
    ==> res{1} = res{2}].
proof.
```

```
    proc.
    simplify.
    auto.
qed.
```

Code block 49. Proof for `swap1_equiv_swap1`

The proof for this lemma is quite straightforward and since there isn't much to reason about, **auto** is strong enough to complete the proof.

Next we prove that `swap1` is equivalent to `swap2`. Now we have different programs, and the way we work with them is by using similar tactics that we used for HL. The only difference now is that we have to add identifiers to the tactics for them to target specific sides and lines of code. For instance, for the sake of a demonstration, we use the **wp** in an asymmetric way. **wp** `n1 n2` will try to consume code up to the $n1^{th}$ line from the left program, up to the $n2^{th}$ line from the right program and will modify the postconditions depending on the instructions that have been consumed. The logic is similar to what we saw in HL.

The proof script begins as usual with the **proc**, and we simplify the result to make it more readable.

```
lemma swaps_equivalent:
    equiv [Jumbled.swap1 ~ Jumbled.swap2 : ={x, y} ==> ={res
    }].
proof.
    proc.
    simplify.
    ...
```

Code block 50. Relating `swap1` and `swap2`

This gives us the goal:

```
Current goal
Type variables: <none>
----------------------------
&1 (left ) : Jumbled.swap1
&2 (right) : Jumbled.swap2

pre = ={x, y}

z <- x                    (1)  z <- y
x <- y                    (2)  y <- x
y <- z                    (3)  x <- z

post = ={x, y}
```

Code block 51. A simple RHL goal

Now we use the tactic `wp 2 3`, which consumes the program statements till line 2 in the left program, and line 3 (meaning it does nothing) in the right program.

```
Current goal
Type variables: <none>
----------------------------
&1 (left ) : Jumbled.swap1
&2 (right) : Jumbled.swap2

pre = ={x, y}

z <- x                      (1)  z <- y
x <- y                      (2)  y <- x
                            (3)  x <- z

post = ={x} /\ z{1} = y{2}
```

Code block 52. Consuming statements using `wp`

For the sake of demonstration, we use `wp 2 2`, and this does nothing to the left program, while it consumes one line of code from the right program. Giving us:

```
(* Part of the output that changed *)
...

z <- x                      (1)  z <- y
x <- y                      (2)  y <- x

...
```

Code block 53. More statements consumed using `wp`

We continue to consume the program statements with `wp` and then reach a state where both the programs are empty, then we continue as usual with the `skip` tactic to get a goal in ambient logic. All of these steps together give us the following proof:

```
lemma swaps_equivalent:
    equiv [Jumbled.swap1 ~ Jumbled.swap2 : ={x, y} ==> ={res
    }].
proof.
    proc.
    simplify.
    wp 2 3.
```

```
    wp 2 2.
    wp 0 1.
    wp 0 0.
    skip.
    trivial.
qed.
```

Code block 54. Complete proof of `swaps_equivalent`

Since we will never really use `wp` the way we did above, we can replace the large block of different calls to `wp n1 n2.` with just `wp.` and EasyCrypt accomplishes the same automatically. Further, we also notice that the proof only uses `wp`, `skip` and `trivial` tactics, so we can use `auto` instead. This gives us a fairly trivial proof for the lemma of the swap functions being equivalent.

```
lemma swaps_equivalent_clean:
    equiv [Jumbled.swap1 ~ Jumbled.swap2 : ={x, y} ==> ={res
    }].
proof.
    proc.
    auto.
qed.
```

Code block 55. Cleaned up proof for `swaps_equivalent`

7.1.2 Abstract modules and adversaries

Now let us take a small detour here and build on the module types that we saw earlier when modelling the abstract IND-RoR game. When working with cryptography, we generally don't know about the inner workings of an adversary or an oracle. In order to model these in EasyCrypt, we have the module types.

```
module type Adv = {
  proc eavesdrop_one(): bool
  proc eavesdrop_two(): bool
}.
```

Code block 56. Definition of module type `Adv`

By defining the module type Adv, we are instructing EasyCrypt that any concrete module which is of the `Adv` type has to, at the very least, implement `eavesdrop_one` and `eavesdrop_two` procedures. What is interesting is that EasyCrypt allows us to reason with the abstract module types as well. For example, let us define a module which expects an Adv as input.

```
module Abstract_game(A:Adv) = {
  proc one(): bool = {
      var x;
      x <- A.eavesdrop_one();
      return x;
  }
}.
```

Code block 57. Definition of module `Abstract_game`

At this stage, we don't know what `A.eavesdrop_one` does and neither does EasyCrypt. However, we can still prove certain facts related to it. Just for a demonstration, we provide a reflexivity example. Notice that we have a new term **`glob`** `A` in the precondition. It stands for the global variables of the module `A`. So in this following lemma, we claim that if the global state of the `A` which is of type `Adv` is equal at the beginning of the program, then running the same program yields us the same result. Quite a simple lemma, however the point to note here is that we haven't defined what the function is.

```
lemma eavesdrop_reflex (A<:Adv):
    equiv [Abstract_game(A).one ~ Abstract_game(A).one :
          ={glob A} ==> res{1} = res{2} ].
proof.
  proc.
  call (_: true).
  auto.
qed.
```

Code block 58. Proof for `eavesdrop_reflex`

`call` does a few complicated things under the hood, but at this point of time, we can think that **`call`** `(_: true)`, knocks off a call to the same abstract function on both sides and adapts the postcondition accordingly.

Let us also define a concrete instantiation of `Adv` called `A` and work with it. Module `A` is quite basic, and either always returns true or always returns false.

```
module A : Adv = {
  proc eavesdrop_one() = {
    return true;
  }

  proc eavesdrop_two() = {
    return false;
  }
}.
```

```
module Games = {
  proc t(): bool =
  { var x; x <- A.eavesdrop_one(); return x; }
  proc f(): bool =
  { var x; x <- A.eavesdrop_two(); return x; }
}.
```

Code block 59. Definitions of modules: `A (Adv)`, and `Games`

Now we can reason with these concrete instances with the tactics that we've seen so far. For instance, since we know that `Games.t` and `Games.f` return different values, we can make a claim like the following and prove it. In this proof, we use **`inline`** `*`, which simply fills in (or "inlines") the concrete code of `Games.t` and `Games.f`. The `*` in the tactic is a selector to select everything that can be inlined.

```
lemma games_quadruple (A<:Adv): equiv [Games.t ~ Games.f :
    ={glob A} ==> res{1} <> res{2} ].
proof.
  proc.
  inline *.
  wp.
  simplify.
  trivial.
qed.
```

Code block 60. Proof for a simple Hoare quadruple

The key takeaway of this detour is that when we work with cryptographic proofs, we will be dealing with both concrete and abstract adversaries. We can now go back to working with some more challenging Hoare quadruples.

7.1.3 Advanced Hoare quadruples

As we discussed earlier, one of the use cases of RHL is to ensure that compiler optimisations preserve program behaviour. Let us take a look at an example of this with a simple compiler optimisation called *invariant hoisting*. Take a look at the programs defined below.

```
module Compiler = {
  proc unoptimised (x y z: int) : int*int = {
    while (y < z){
      y <- y + 1;
      x <- z + 1;
    }
    return (x,y);
```

```
  }
  proc optimised (x y z: int) : int*int = {
    if(y < z){
      x <- z + 1;
    }
    while (y < z){
      y <- y + 1;
    }
    return (x,y);
  }
}.
```

Code block 61. Definition of module `Compiler`

As you can observe, if the condition of the **`while`** loop in `unoptimised` holds for even one iteration the `x` is set to `z+1`. However, the subsequent iterations of the loop don't change `x`. Hence to save on computation, the compiler hoists that line out of the scope of the **`while`** loop, giving us `optimised`.

Now let us try to prove the fact that the behaviour of both the programs is equivalent. At this point, there can be two possibilities:

1. `!(y < z)`:
   In this case, neither the **`while`** loop nor the **`if`** condition is satisfied. So, both the programs effectively do nothing to the variables.

2. `(y < z)`:
   The **`while`** loop and the **`if`** condition are executed at least once. In this case, the variables are modified.

So to prove that the optimisation is correct, we can break our proof into these two cases, work on them independently and then put them back together.

Let us first work with the part where the code is never executed, and call that **`lemma`** `optimisation_correct_a`. We demonstrate the relevant parts of goals and how the tactics that we use change them. Again, as usual, we use **`proc`** and **`simplify`** at the beginning giving us the proof:

```
lemma optimisation_correct_a:
equiv [Compiler.unoptimised ~ Compiler.optimised:
      ={x, y ,z} /\ !(y{1} < z{1}) ==> ={res}  ].
proof.
  proc.
  simplify.
  ...
```

Code block 62. Proving `optimisation_correct_a`

At this stage the goal state is:

```
Current goal
Type variables: <none>
----------------------------
&1 (left ) : Compiler.unoptimised
&2 (right) : Compiler.optimised

pre = ={x, y, z} /\ ! y{1} < z{1}

while (y < z) {              (1--)  if (y < z) {
  y <- y + 1                 (1.1)    x <- z + 1
  x <- z + 1                 (1.2)
}                            (1--)  }
                             (2--)  while (y < z) {
                             (2.1)    y <- y + 1
                             (2--)  }

post = ={x, y}
```

Code block 63. Goal when proving `optimisation_correct_a`

Note: EasyCrypt prints this goal in a way that is slightly hard to read, we have edited the white spaces to make it easier to read.

Now we can first deal with the **if** condition with the use of the **seq** tactic. The **seq** tactic does this:

$$\frac{\{P\}A1; \sim B1; B2; \{R\} \quad \{R\}A2; A3; \sim B3; \{Q\}}{\{P\}A1; A2; A3 \sim B1; B2; B3\{Q\}}\ seq\ 1\ 2 : (R)$$

The idea behind using the **seq** is to break the programs into manageable chunks and deal with them separately. Since we know that the code is not executed we can pass $R = P$, so in our case $R =$ `={x, y, z} / ! y{1} < z{1}`. With this we can knock off the **if** from `optimised` using **seq**.

```
lemma optimisation_correct_a:
equiv [Compiler.unoptimised ~ Compiler.optimised:
      ={x, y ,z} /\ !(y{1} < z{1}) ==> ={res}  ].
proof.
  proc.
  simplify.

  seq 0 1: ( ={x, y ,z} /\ !(y{1} < z{1}) ).
  ...
```

Code block 64. Using **seq** to progress in the proof

At this stage, the goal splits into two. The first goal, has only the `if` condition from the right program, with the pre and postconditions which are equal. The second goal has the rest of the code from both the programs, i.e the `while` loops, with the `pre = ={x, y, z} / ! y{1} < z{1}`, and `post = ={x, y}`

The first goal is easily proved using `auto`. Then we use `rcondf` to deal with the `while` loops since we know that they won't be executed. Together we get the following proof:

```
lemma optimisation_correct_a:
equiv [Compiler.unoptimised ~ Compiler.optimised:
      ={x, y ,z} /\ !(y{1} < z{1}) ==> ={res}  ].
proof.
  proc.
  simplify.

  seq 0 1: ( ={x, y ,z} /\ !(y{1} < z{1}) ).
  auto.

  (* We knock off the while loop on the left *)
  rcondf {1} 1.
  auto.

  (* Similarly, we remove the while loop on the right *)
  rcondf {2} 1.
  auto.

  auto.
qed.
```

Code block 65. `optimisation_correct_a`: Complete proof

Now let us work with the second part of the proof that deals with the part where the loops are executed. We first deal with the `if` condition on the right program. This time we know that the loop gets executed, so we first preserve the parts of the precondition that are unaffected by the loop and then add `x{2} = z{2} + 1` to it.

Then we work with the `while` loop. The only complex part of this proof is figuring out the invariant. In this case, we know that after every iteration of the loop `y` and `z` on both sides are equal, this gives us `={y,z}`, and since the `if` condition was executed on the right, we have `x{2} = z{2} + 1` as well. Then `(y{1}<z{1} / x{1} = z{1} + 1)` comes through since in the left program, `y{1}<z{1}` holds if the body of the loop hasn't been executed or `x{1} = z{1} + 1` holds after at least one execution of the loop. Putting these together we get the slightly complex invariant, but the rest of the proof comes through easily, giving us the following complete proof.

```
lemma optimisation_correct_b:
```

```
equiv [Compiler.unoptimised ~ Compiler.optimised:
      ={x, y ,z} /\ y{1}<z{1} ==> ={res}  ].
proof.
  proc.
  (* Dealing with the if condition in optimised *)
  seq 0 1: (={y,z} /\ y{1}<z{1} /\ x{2} = z{2} + 1).
  simplify.
  auto.
  (* Dealing with the while loops in both *)
  while (={y,z} /\ x{2} = z{2} + 1 /\
        (y{1}<z{1} \/ x{1} = z{1} + 1)).
  auto.
  auto.
  smt().
qed.
```

Code block 66. `optimisation_correct_b`: Complete proof

Now we can put these both together. In this proof, we use **proc***, which modifies the goal in a way similar to **proc** but without losing the fact that the code is a procedure call i.e without inlining the definition of the procedure. Then we split on the boolean expression of the **while** loop. Upon splitting we get two goals which correspond to the proofs that we did above. We use **call** `lemma_name` with the appropriate lemma name to use the proofs that we've already written. Together these ideas give us the following proof:

```
lemma optimisation_correct:
equiv [Compiler.unoptimised ~ Compiler.optimised:
      ={x, y ,z} ==> ={res}  ].
proof.
  proc*.

  (* Branching on the boolean *)
  case (y{1}<z{1}).

    (* y{1}<z{1} *)
    call optimisation_correct_b. auto.
    (* ! y{1}<z{1} *)
    call optimisation_correct_a. auto.
qed.
```

Code block 67. Putting things together for `optimisation_correct`

That concludes the chapter on Relational Hoare Logic, in the practice file we've included a couple of exercises for the reader to complete with helpful hints about how

to solve them as well. For instance, we have another compiler optimization with two variants, an easy one in which the reader can manually unroll the loops and prove that the optimisation is correct, and a similar optimisation in which the reader needs to think more in abstract terms in order to prove the result.

8 Conclusion

We present the first few chapters of learning material for EasyCrypt, a toolset for reasoning about cryptographic proofs. We adopt a teaching style and exposition inspired by Software Foundations, a series of interactive textbooks to learn about Coq, a formal proof management system. Our work is open-sourced and hosted on GitHub [9]. We present the following material for the learners to work through:

1. Introduction to formal verification
2. An introduction to EasyCrypt
3. Ambient logic
4. Hoare logic and its variants:
   (a) Classical Hoare logic (HL)
   (b) Relational Hoare logic (RHL)

The chapters with practice files consist of detailed proof scripts that teach the user the basic concepts required to work with EasyCrypt and provide a lot of practice exercises to strengthen the said concepts.

Our work attempts to address the severe lack of educational material that is required for learning tools such as EasyCrypt. The value of our work comes in the form of creating a good starting point for more beginners to get into the field. The material can directly be used as instructional material for both bachelor's and master's level courses on EasyCrypt, and can easily span the first few weeks of workload.

8.1 Future work

Admittedly, this work is by no means complete since we barely scratched the surface of what EasyCrypt is capable of doing and the current state of the art when it comes to cryptography.

We believe that our work will form the basis of learning for beginners in the future. We hope that they will learn the material faster and contribute to improving and covering more concepts. Here are some possible future directions for this project:

1. **Covering more EasyCrypt concepts**:
   Notably, the next few chapters should present the readers with practice on Probabilistic Hoare logic (pHL) and Probabilistic Relational Hoare logic (pRHL). While working with these logics, the material can already start introducing cryptography.

2. **Further depth of already covered concepts**:
   Our treatment of the tactics has been such that it gives the readers a quick idea of what they do. However, many tactics have different variations and can use a more detailed explanation. For instance, the `move =>` tactic has several introduction patterns that we haven't addressed in this work.

3. **Automated scripts to create beautiful documentation**:
   Our current process of writing this material has been to first write a lot of material in the `.ec` files and then manually pick the important concepts to highlight in the theory. Or write some theory in LaTeX first and then create the `.ec` files to provide practice for the ideas covered.

   This creates the problem of both the documents diverging and additional work to fix these issues of diverging files. This problem can be avoided if we create scripts that can read the `.ec` and automatically create readable documentation based on unique delimiters in the text. This would simplify the process of writing educational material quite a lot.

References

[1] Benjamin C. Pierce et al. Software Foundations. https://softwarefoundations.cis.upenn.edu/.

[2] Coq Developer Team. Coq Documentation. https://coq.inria.fr/documentation.

[3] Coq Developer Team. Guide to contributing to Coq. https://github.com/coq/coq/blob/master/CONTRIBUTING.md.

[4] Mike Rosulek. The Joy of Cryptography. https://joyofcryptography.com.

[5] Dominique Unruh. Verification of cryptography with EasyCrypt. https://courses.cs.ut.ee/2015/easycrypt/spring/Main/HomePage.

[6] Tejas Anil Shah. Basics of EasyCrypt. https://youtube.com/playlist?list=PLZBF71EUPkRsSI-8YzntddhkwRw466dbo.

[7] EasyCrypt. Commit hash: ce56b1055fabec3d5b50e230ad15642956101f1e. https://github.com/EasyCrypt/easycrypt/commit/ce56b105.

[8] EasyCrypt. Commit hash: ea77e0ed6285f9d5f3cc5b26db8d476d7d9bb68a. https://github.com/EasyCrypt/easycrypt/commit/ea77e0ed.

[9] Tejas Anil Shah. The Joy of Easycrypt. https://github.com/tejasanilshah/the-joy-of-easycrypt.

[10] NASA. Mars Climate Orbiter: In Depth. https://solarsystem.nasa.gov/missions/mars-climate-orbiter/in-depth/.

[11] Eric M. Johnson. Timeline: Boeing's 737 MAX crisis. https://www.reuters.com/article/uk-boeing-737max-timeline-idUKKBN27Y1RQ.

[12] Emacs. A guided tour of emacs. https://www.gnu.org/software/emacs/tour/.

[13] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429 – 490, 1977.

[14] Kenneth Appel, Wolfgang Haken, and John Koch. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491 – 567, 1977.

[15] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptology ePrint Archive*, 2005:181, 01 2005.

[16] Shai Halevi. EME*: extending EME to handle arbitrary-length messages with associated data. Cryptology ePrint Archive, Paper 2004/125, 2004. https://eprint.iacr.org/2004/125.

[17] Mike Rosulek. The Joy of Cryptography. https://joyofcryptography.com/pdf/chap1.pdf.

[18] Matt Bauer and Mike Dodds. Who is verifying their cryptographic protocols? https://galois.com/blog/2021/05/who-is-verifying-their-cryptographic-protocols/.

[19] David Basin, Jannik Dreier, et al. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1383–1396, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Manuel Barbosa, Gilles Barthe, et al. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Paper 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[21] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017.

[22] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, et al. A machine-checked proof of security for aws key management service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 63–78, New York, NY, USA, 2019. Association for Computing Machinery.

[23] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, et al. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1607–1622, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Alt-Ergo. An SMT solver for software verification. https://alt-ergo.ocamlpro.com/.

[25] Z3. A theorem prover from Microsoft Research. https://github.com/Z3Prover/z3.

[26] CVC4. An efficient open-source automatic theorem prover. https://cvc4.github.io/.

[27] Why3. Where Programs Meet Provers. https://why3.lri.fr/.

[28] Alley Sthoughton. Easycrypt Installation Instructions. https://alleystoughton.us/easycrypt-installation.html.

[29] Michael Huth and Mark Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2nd ed edition, 2004.

[30] John C. Mitchell. *Foundations for programming languages*. Foundations of computing. MIT Press, 1996.

Appendix

Glossary

CAC	...	Computer aided Cryptography
HL	...	Hoare Logic
IND-RoR	...	Indistinguishability - Real or Random
JoC	...	Joy of Cryptography
pHL	...	Probabilistic Hoare Logic
pRHL	...	Probabilistic Relational Hoare Logic
RHL	...	Relational Hoare Logic
SMT	...	Satisfiability Modulo Theories
SoK	...	Systematisation of Knowledge

Figures and tables

Code blocks

IV. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Tejas Anil Shah**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   The Joy of EasyCrypt,

   supervised by Prof. Dominique Unruh.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tejas Anil Shah
08/08/2022