Formalising Sigma-Protocols and commitment schemes within EasyCrypt

Nikolaj Sidorenco, 201504729

Master's Thesis, Computer Science

March 15, 2020

Advisor: Bas Spitters

Co-advisor: Sabine Oechsner

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English…◄

Resumé

►in Danish. . . ◄

Acknowledgments

►…◄

Nikolaj Sidorenco,
Aarhus, March 15, 2020.

Contents

Chapter 1

Introduction

►**motivate and explain the problem to be addressed**◄
►**get your bibtex entries from** https://dblp.org/◄

Chapter 2

EasyCrypt

It this chapter we introduce the EasyCryptproof assistant ...

EasyCryptprovides us with three important logics: a relational probabilistic Hoare logic (**rPHL**), a probabilistic Hoare logic (**pHL**), and a Hoare logic. Furthermore, EasyCryptalso has an Higher-order ambient logic, in which the three previous logics are encoded within. This Higher-order logic allows us to reason about mathematical constructs, which in turn lets us reason about them within the different Hoare logics. The ambient logic also allows us to relate judgement the three different types of Hoare logics, since they all have an equivalent representation in the ambient logic.

2.1 Types and Operators

2.2 Theories, Abstract theories and Sections

To structure proofs and code EasyCryptuses a language construction called theories. By grouping definitions and proofs into a theory they become available in other files by "requiring" them. For example, to make use of EasyCrypt's existing formalisation of integers, it can be made available in any giving file by writing:

To avoid the theory name prefix of all definitions "require import" can be used in-place of "require", which will add all definitions and proof of the theory to the current scope without the prefix.

Any EasyCryptfile with the ".ec" file type is automatically declared as a theory.

Abstract Theories To model parametric protocols, i.e. protocols that can work on many different types we use EasyCrypt's abstract theory functionality. A abstract theory allows us to model protocols and proof over generic types. There is currently two ways of declaring an abstract theory. First, by using the "theory" keyword within any

Listing 2.1: EasyCrypttheories: importing definitions

```
require Int.

const two : int = Int.(+) Int.One Int.One.
```

file allows the user to define abstract types, which can be used though-out the scope of the abstract theory, i.e. everything in-between the "theory" and "end theory" keywords. Second, an abstract theory file can be declared by using the ".eca" file type. This works much like using the ".ec" file type to declare theories.

Sections Sections provide much of the same functionality, but instead of quantifying over types sections allows us to quantify everything within the section over modules axiomatised by the user.

An example of this, is having a section for cryptographic security of a protocols, where we quantify over all instances of adversaries, that are guaranteed to terminate.

2.3 Modules and procedures

To model algorithms within EasyCryptthe module construct is provided. A module is a set of procedures and a record of global variables, where all procedures are written in EasyCryptembedded programming language, pWhile. pWhile is a mild generalization of the language proposed by Bellare and Rogaway [2006]?

Modules are, by default, allowed to interact with all other defined modules. This is a due to all procedures are executed within shared memory. This is to model actual execution of procedures, where the procedure would have access to all memory not protected by the operating system.

From this, the set of global variables for any given module, is all its internally defined global variables and all variables the modules procedures could potentially read or write during execution. This is checked by a simple static analysis, which looks at all execution branches within all procedures of the module.

A module can been seen as EasyCrypt's abstraction of the class construct in object-oriented programming languages.

►**Example of modules**◄

Modules Types Modules types is another features of EasyCryptmodelling system, which enables us to define general structures of modules, without having to implement the procedures. A procedure without an implementation is called abstract, while a implemented one (The ones provided by modules) are called concrete.

An important distinction between abstract and non-abstract modules is that, while non-abstract modules define a global state, in the sense of global variables, for the procedures to work within, the abstract counter-part does not. This has two important implications, first it means that defining abstract modules does not affect the global variables/state of non-abstract modules. Moreover, it is also not possible to prove properties of abstract modules, since there is no context to prove properties within.

It is, however, possible to define higher-order abstract modules with access to the global variables and procedures of another abstract module.

This allows us to quantitate over all possible implementations of an abstract module in our proofs. This implications of this, is that it is possible to define adversaries and then proving that no matter what choice the adversary makes during execution, he will not be able to break the security of the procedure.

►**Example of abstract modules**◄

Listing 2.2: nextHopInfo: IND-CPA Game

```
module IND–CPA(A : Adversary) = {
  proc main() : bool = {
    var m0, m1, b, b', sk, pk;

    (sk, pk) = key_gen();

    (m0, m1) = A.choose(pk);

    b <\$ dbool;

    b' = A.guess(enc(sk, m_b));

    return b == b'
  }
}
```

2.4 Probabilistic Hoare Logic

To formally prove security of a cryptographic protocol we commonly do we in the way of so-called game-based proofs, we define a game against a malicious adversary, and say that the protocol is secure, if the adversary cannot win said game. An common example of this is IND-CPA security, where an probabilistic polynomial time adversary is given access to an PKE-oracle and is allowed to send two messages to the oracle, namely m_0, m_1. The oracle then chooses a random bit, b, and sends the encryption of m_b back to the adversary. Then, if the adversary can guess which of the two messages where encrypted we wins. To reason about such within EasyCryptwe first describe the game within a module:

►Problems with game: A should know what PKE schemes is used. A bit too psuedo-code-ish◄

Now, to prove security we would like to show that the adversary cannot win this game with probability better than randomly guessing values of b', i.e. $\frac{1}{2}$.

This formulated in one of two ways:

$$forall(Adv <: Adversaries)\&m, Pr[IND-CPA(Adv).main()@\&m : res] = 1\%r/2\%r \tag{2.1}$$

$$forall(Adv <: Adversaries), phoare[IND-CPA(Adv).main : true ==> res] = 1\%r/2\%r \tag{2.2}$$

Both are equivalent representations, but the former is in the ambient logic of EasyCrypt, whilst the latter is in the pHL logic.

To prove this, we step though the game using the logic rules of the pHL logic. But, how can we guarantee that there does not exists any possible implementation of the adversary, such that we he is able to succeed? To prove this we either have to somehow

prove, that such an adversary cannot exists within the current game, or we can relate this game to another one, where adversary can only perform random guesses. To do this we need to utilise the pRHL logic.

2.5 Probabilistic Relational Hoare Logic

the pRHL logic allows us to reason about the join outcome distribution of two programs. This allows us to reason about equality of games.

To bing it back to the game of IND-CPA, we could define an alternative game, where the Oracle always sends back a random element from the ciphertext space. This is often referred to as the ideal case, where as the oracle previously introduced is referred to as the real one.

We can then formulate equality between the two games as:

$$forall(Adv <: Adversary)\&m, Pr[IND-CPA(Adv).main()@\&m : res] = Pr[IND-CPA(Adv).real()@\&m : res] \tag{2.3}$$

$$forall(Adv <: Adversary), equiv[IND-CPA(Adv).main\, IND-CPA(Adv).real : true ==> = \{res\}] \tag{2.4}$$

where $= \{res\}$ is notation for the outcome distributions begin equal.

Chapter 3

Background

3.1 Sigma Protocols

Originally introduced by Cramer, Σ-protocols are protocols on a three-move-form, based on a, computationally hard, relation R, such that $(h, w) \in R$ if h is an instance of a computationally hard problem, and w is the solution to h. Σ-protocols then allows a prover, P, who knows the solution w, to convince a verify, V, of the existence of w, without explicitly showing w to him.

The following section aims to introduce the definition of Σ-protocols, along with its notions of security. The following section is based on the presentation of Σ-protocols by Damgaard [2].

3.1.1 Definition

- 3-move form
- Completeness
  - Protocol should always succeed with correct output, if both parties are honest.
- Special soundness
  - Given two transcripts, with different challenges it is possible to compute an accepting witness for the statement in the relation.
- Special Honest Verifier Zero-Knowledge.
  - Exists an polynomial-time simulator M.
  - Given statement, x, and challenge, e, output an accepting conversation (a, e, z).
  - Conversation should have the same distribution as an conversation between honest parties.

3.2 Commitment Schemes

3.2.1 Definition

Chapter 4

Related Work

Chapter 5

Conclusion

►conclude on the problem statement from the introduction◄

Bibliography

[1] EasyCrypt reference manual. https://www.easycrypt.info/documentation/refman.pdf. Accessed: 2020-04-03.

[2] Ivan Damgaard. On Σ-protocols. lecture notes, Aarhus University, 2011.

Appendix A

CryptHOL

Alternative to EasyCrypt. Based on Isabelle.

►**What is the differences between the ambient logics in Isabelle/Coq/EasyCrypt?**◄

A.1 Encoding a protocol within Isabelle

Appendix B

CertiCrypt

Predecessor of EasyCrypt. Implemented in Coq. But there have been cases, where CertiCrypt has been a relevant alternative to EasyCrypt. See ZKCrypt paper.

B.1 Encoding a protocol within Isabelle

Appendix C

An Introduction to EasyCrypt

C.1 Game Hopping

C.2 Proving procedures incrementally

This sections aim to give the reader an introduction on how to apply EasyCrypt's (r)pHL logic to prove statement judgements **►explain what is a statement judgement◄**. This section, however, will not cover every possible of manipulating Hoare judgements, but rather aim to familiarise the reader with the general techniques observed and applied though-out this master thesis. For a more comprehensive overview of the tactics supplied by EasyCryptthe reader is encouraged to read the reference manual [1].

When working with statement judgements, there are five general categories, for which all procedural statements can be categorised into, which corresponds exactly to the five different cases of valid instructions within the BNF representation of pWhile. For the work of master thesis reasoning about procedures containing loops has not been necessary and they are, therefore, also excluded from this section.

statements depending only on local/global variables This type of statement is seen, when assigning variables in EasyCrypt.

To deal with the assignment in Example C.1, we have two fundamental tactics,

Listing C.1: Example: assigning values in EasyCrypt

```
module Assignment = {
  var a : int
  proc main() = {
    var x;
    a = 5;
    x = a;
    (* statements... *)
    x = 7;
  }
}
```

which relate to the logical rules in formal Hoare logic.

The first tactic is **sp**, which given a Hoare triple {pre}prog{post} consumes the longest prefix of assignment statements in prog, and then replaces the precondition of the Hoare triple with the strongest postcondition, R, for which it holds: {pre}prefix{R}.

The second tactic is **wp**, which given a Hoare triple {pre}prog(){post} consumes the longest suffix of assignment statements in prog, and then replaces the postcondition of the Hoare triple with the weakest precondition, R, for which it holds: {R}suffix{post}.

statements depending on (concrete) procedure calls When dealing with concrete procedure calls it is possible to inline the procedure, such that solving statement of this kind effectively reduces to proving a program only consisting of statements of the other three kinds.

statements depending on (abstract) procedure calls call tactic

statements depending on distributions

C.3 Higher order procedure proofs

rewrite and apply is used on ambient logic. If we want to use knowledge about procedures, we have to use call in most cases.

conseq, byphoare and byequiv can also take proof terms as arguments.

C.4 Example : OTP