Formalising Sigma-Protocols and commitment schemes within EasyCrypt

Nikolaj Sidorenco, 201504729

Master's Thesis, Computer Science

March 31, 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 31, 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

This section aims to introduce some of the fundamental definitions and concepts used throughout this thesis. This section will foremost give a rudimentary and informal introduction, while the later chapters will provide more rigours formalisations and proofs of security. First, we start by describing Σ-protocols, along with its security definitions and extensions.

3.1 Sigma Protocols

Originally introduced by Cramer, Σ-protocols are two-party protocols with 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.

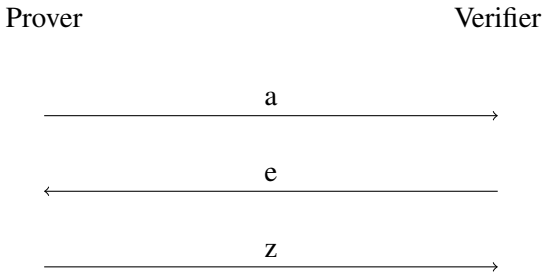

Figure 3.1: Σ-Protocol

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 [3].

►**explain the flow of the protocol. what is a . . .** ◄

To prove security of a Σ-protocols, we require three properties, namely, **Completeness**, **Special Soundness**, and **Special Honest Verifier Zero Knowledge (SHVZK)**.

Completeness Protocol should always succeed with correct output, if both parties are honest.

Special Soundness Given two, accepting, transcripts, with different challenges it is possible to compute an accepting witness for the statement in the relation.

The special soundness property is important for ensuring that a cheating prover cannot succeed. Given special soundness, if the protocol is run multiple times, his advantage becomes negligible, since special soundness implies that there can only exists one challenge, for any given message a, which can make the protocol accept, without knowing the witness. Therefore, given a challenge space with cardinality c, the probability of a cheating prover succeeding in convincing the verifier is $\frac{1}{c}$. The protocol can then be run multiple times, to ensure negligible probability.

Can also be generalised to s-Special Soundness, which requires that the witness can be constructed, given s accepting conversations.

SHVZK Zero knowledge is proven by giving a simulator. Not possible to define for corrupted verifier, since the challenge can become exponentially long.

- 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.

►Compound protocols◄
►Relation in literature not always correct◄
►non-interactive by Fiat-Shamir◄

3.2 Commitment Schemes

Commitment schemes is another fundamental building block in cryptography, and has a strong connection to Σ-Protocols **►reference◄**. Commitment schemes facilitates an interaction between two parties, P1 and P2, where P1 can generate a commitment of a message, which he can send to P2, without revealing what his original message where. At a later point P1 can then send the message to P2, who is then able to verify that P1 has not altered his message since creating the commitment. More formally a commitment schemes is defined as:

Definition 3.2.1 (Commitment Schemes) *A commitment schemes is a tuple of algorithms (KeyG, Com, Ver), where:*

- $(ck, vk) \leftarrow KeyG()$*, provides key generation.*
- $(c, d) \leftarrow Com(ck, m)$ *generates a commitment c of the message m along with a opening key d, which can be revealed at a later time.*
- $\{true, false\} \leftarrow Ver(vk, c, m, d)$ *checked whether the commitment c was generated from m and opening key d.*

For a commitment schemes to be secure it is required to satisfy three properties: **Correctness**, **Binding**, and **Hiding**.

Correctness A commitment scheme is said to be correct, if a commitment (c,d) made by a honest party will always be accepted by the verification procedure of another party, i.e:

$$Pr[Ver(vk,c,m,d)|(c,d) = Com(ck,m) \wedge (ck,vk) \leftarrow KeyG()] = 1.$$

Binding The binding property states that a party committing to a message will not be able to successfully convince another party, that he has committed to another message, i.e. $(c,d) \leftarrow Com(ck,m)$, will not be able to find an alternative opening key d' and message m' such that $(c,d') \leftarrow Com(ck,m')$.

►write binding game◄

►Stat, perfect, comp variants◄

►All games parametrised by security param. Not used in formal definition◄

Hiding The Hiding property states that a party given a commitment c, will not be able to extract the message m, which the commitment was based on.

3.3 Multi-Part Computation

Chapter 4

Formalising commitment schemes

Chapter 5

Formalising Sigma-Protocols

Chapter 6

Towards generalised zero-knowledge compilation

We have previously seen a concrete instantiation of a Σ-protocol with the relation being the discrete logarithm problem, namely Schnorr's protocol. We have also seen how it was possible to prove Schnorr's protocol to be secure in a formal setting though EasyCrypt.

There exists an infinite set of possible relations, for which we could want to provide zero-knowledge proofs of. It is therefore infeasible to design a protocol for each relation and proving its security.

An example of this is the GMW-compiler . . . , or the example system from [2].

We therefore need a more generalised approach, that is able to generate zero-knowledge proof for an entire family of relations rather than a specific relation. One such family of relations is the pre-image under group homomorphisms . . .

Chapter 7

ZKBoo

Generalised Σ-protocol for relations of the form $\phi(x) = y$, where ϕ and y are public inputs, and x is the witness for the relation.

- (2,3)-Decomposition of ϕ
- "MPC in the head" of the decomposition
- Σ-protocol proving that MPC computation succeed and was not tampered with.

Chapter 8

Formalising ZKBoo

To formalise the ZKBoo protocol we need the following:

- A formalisation of Arithmetic circuits
  - With evaluation semantics
- A formalisation of the (2,3)-Decomposition
  - Proof of completeness
  - Proof of 2-Privacy
- An instantiation of a Σ-protocol for "MPC in the head"

8.1 Formalising Arithmetic circuits

Since the function we are computing are on the form $\phi(x) = y$, where ϕ can perform any arbitrary computation depending on x we need gates with out-degree of ≥ 1.

We are restricted to finite integer rings, and therefore, we only need four basic gates to express our circuit, namely, Add to constant (ADDC), multiply by constant (MULTC), addition of two wires, and multiplication of two wires.

For ADDC and MULTC the gates only have one in-going edge, but can have arbitrary out-going edges. ADD and MULT have two in-going edges, but can also have an arbitrary number of out-going edges.

Depth of circuits Since arithmetic circuits can usually be expressed as an upside-down tree, the depth of the circuit is depth of the tree. We chose the visualize our circuit is a different way, namely that, each gate in the circuit is its own layers and the depth of the circuit is therefore the number of gates in the circuit. The reason for this, is to simplify the proofs of correctness and privacy, which both depend on the structure on the circuits. **►This can be seen in the following sections . . . ◄**.

►Tikz example showing the two representations◄

From this representation each gate can be labelled according to its depth. This gives an ordering on the execution for all the gates. From this ordering a natural choice is the represent the circuit as a list of gates, such that gate at depth 0 is the gate at index 0 in the list.

Listing 8.1: Type declaration of gates

```
type gate = [
  | ADDC of (int * int)
  | MULTC of (int * int)
  | MULT of (int * int)
  | ADD of (int * int)
].

type circuit = gate list.
```

One remaining problem is then, that gates can have input wires coming from any depth of the circuit, and the is no way to calculate the incoming gates based on its location within the list. We therefore encode the incoming wires directly into our representation of the gates, e.g. the multiplication gate will consist of a tuple (i, j) where i is the index of left incoming wire and j is the index of right incoming wire.

To differentiate between the tuples we use EasyCrypt's type system to make types for each gate, which can be seen in figure 8.1. Here the unary gates of ADDC and MULTC are also represented as tuples. The first entry of the tuple is the incoming wire and the second entry is the constant used.

From this representation of circuits as a list of gates, where gates are types, it is not possible to define the semantic meaning of this representation, by defining the evaluation function, which can be seen in figure 8.2. The evaluation is broken into two parts: one for evaluation one gate, and one for evaluating the entire circuits, based on the former. To evaluate one gate, we first need to determine which gate it is. This is done by matching on the type of the gate. Then, since execution order follows the ordering of indexes, it can be assumed that if we are computing index i of the circuit, then indices $[0 \ldots i-1]$ have already been computed to an value representing the result of computing the gate. Perform the appropriate function then simply reduces to looking the values of the incoming wires and computing the function.

Computing the entire gate then follows from the same fact, that gate are always evaluated in the order the appear in the list, and the no gate can depend on the result of gates, which have a higher index than itself. By continually performing gate evaluation of the first entry of the list, saving the result into "state" where each index correspond to the computed value of the gate at that index in the circuit, and the calling recursively on the list with the first entry removed, then the output of the gate will be in the last entry of the state, when there are no more gates to compute. Assuming that there is only one output gate.

►Add function to verify correctness of circuit?◄

8.2 (2,3) Decomposition of circuits

►We put no assumptions on the circuit being valid, since the decomposition will fail with the same values as the circuit evaluation.◄

Listing 8.2: Circuit evaluation function

```
op eval_gate (g : gate, s : int list) : int =
  with g = MULT inputs => let (i, j) = inputs in
                          let x = (nth 0 s i) in
                          let y = (nth 0 s j) in x * y
  with g = ADD inputs =>  let (i, j) = inputs in
                          let x = (nth 0 s i) in
                          let y = (nth 0 s j) in x + y
  with g = ADDC inputs => let (i, c) = inputs in
                          let x = (nth 0 s i) in x + c
  with g = MULTC inputs => let (i, c) = inputs in
                          let x = (nth 0 s i) in x * c.

op eval_circuit_aux(c : circuit, s : int list) : int list =
    with c = [] => s
    with c = g :: gs =>
     let r = eval_gate g s in
     eval_circuit_aux gs (rcons s r).

op eval_circuit (c : circuit, s : state) : output =
    last 0 (eval_circuit_aux c s).
```

Chapter 9

Related Work

Chapter 10

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] José Bacelar Almeida, M. Barbosa, E. Bangerter, Gilles Barthe, Stephen Krenn, and Santiago Zanella-Béguelin. Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols. In *19th ACM Conference on Computer and Communications Security*, pages 488–500. ACM, 2012. URL http://dx.doi.org/10.1145/2382196.2382249.

[3] 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 CryptHOL

Appendix B

CertiCrypt

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

B.1 Encoding a protocol within CertiCrypt

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 EasyCrypttthe 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