# Formalising Sigma-Protocols and commitment schemes within EasyCrypt

Nikolaj Sidorenco, 201504729

Master's Thesis, Computer Science

May 23, 2020

Advisor: Bas Spitters

Co-advisor: Sabine Oechsner

AU AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

# Abstract

**►in English…◄**

# Resumé

**►in Danish. . . ◄**

# Acknowledgments

►…◄

*Nikolaj Sidorenco,*
*Aarhus, May 23, 2020.*

# Contents

**Abstract** iii

**Resumé** v

**Acknowledgments** vii

**1 Introduction** 1

**2 EasyCrypt** 3
- 2.1 Types and Operators . . . 3
- 2.2 Theories, Abstract theories and Sections . . . 3
- 2.3 Modules and procedures . . . 4
- 2.4 Probabilistic Hoare Logic . . . 5
- 2.5 Probabilistic Relational Hoare Logic . . . 6

**3 Background** 7
- 3.1 Zero-Knowledge . . . 7
- 3.2 Sigma Protocols . . . 7
- 3.3 Commitment Schemes . . . 8
- 3.4 Multi-Part Computation (MPC) . . . 9

**4 Formalising commitment schemes** 11
- 4.1 Key-based commitment schemes . . . 11
- 4.2 Key-less commitment schemes . . . 12
- 4.3 Concrete instantiation: Pedersen Commitment . . . 12

**5 Formalising Sigma-Protocols** 13
- 5.1 Composition Protocols . . . 16
  - 5.1.1 AND . . . 16
  - 5.1.2 OR . . . 17
  - 5.1.3 Fiat-Shamir Heuristic . . . 17
  - 5.1.4 Concrete Example: Schnorr protocol . . . 17

**6 Generalised Zero-Knowledge compilation** 19
- 6.1 ZKBOO . . . 19
  - 6.1.1 (2,3)-Function Decomposition . . . 20
  - 6.1.2 Making the protocol zero-knowledge . . . 21

**7 Formalising ZKBoo** **23**
7.1 Formalising Arithmetic circuits . . . . . 23
7.2 (2,3) Decomposition of circuits . . . . . 26
7.2.1 Correctness . . . . . 27
7.2.2 2-Privacy . . . . . 28
7.3 ZKBOO . . . . . 30

**8 Related Work** **37**

**9 Conclusion** **39**

**Bibliography** **41**

# Chapter 1

# Introduction

**►Maybe add database example from CertiCrypt paper?◄**

Computer-aided cryptography is an active area of research that combines the fields of formal verification and cryptography to develop formal machine-checkable proofs approaches to cryptographic protocol design as well as the implementations of said protocols [2].

**►Formal verification is ...◄**

the field of cryptography has in recent times lend itself to "game hopping" approach to design-level security of protocols. In this approach security is modelled as the probability of an adversary, with varying degrees of power, breaking the intended design of the protocol. This is done usually done by either proving that it is impossible for the adversary to break to protocol by mathematical reasoning on the possible choices the adversary can make or proving that breaking the protocol is computationally equivalent to solving a, presumably, difficult problem.

The problem with this approach is that these game reduction are often prone to implicit assumptions and rely on peer-reviewing of the proof of security.

Finally, cryptographic design does not led it self to implementation level security issues, which is usually left to domain expert when implementing the protocols.

Computer-aided cryptographic can help us solve these problems by making the assumptions more transparent, but moreover it also makes the logical reasoning more transparent. Since formal proofs can only proceed by applying logical rules, then an error can only arise from having a flaw in the core logic or in one of the assumptions made for the protocol.

Having formally verified software is especially important when it is used as a building block in bigger and more complex protocols. One such example of a fundamental building block in cryptology is Zero knowledge arguments, which since their introduction has been in many complex cryptographic protocols. Most notably they have found important use on the blockchain and in the GMW compiler, which can compile any passively secure protocol into a active secure one.

One problem with Zero-knowledge arguments is that they usually fall into two categories: They are either efficient, but to specialised, or they are generalised but too inefficient. A example of the former is Schnorr protocol, which allows us to prove knowledge of a discrete logarithm.

This has led to a new class of protocols known as *Succinct Non-interactive ARgu-*

*ments of Knowledge*, also known as SNARKS, which aim to improve on the generalised zero knowledge protocols by reducing the size of the proof and moves the bulk of the work to prover, such that SNARKS can be quickly verified but are expensive to make. **►reference◄**

Since these generalised zero-knowledge compilers can be used within applications where security is critical it is important to that they have been rigorously studied and formally proven.

In this paper we will therefore look at the ZKBoo protocol by Giacomelli et al. [6], which can generate zero-knowledge proofs for any relation, assuming the relation can be expressed as a circuit, with a bound on the proof size.

In doing so we will develop a rich formalisation of Σ-protocols they relate to Zero-knowledge. Moreover, we will show how to create a fully verified toolchain for constructing a generalised zero-knowledge compiler based on ZKBoo.

the main contribution of this thesis is . . .

**►Goal: Develop a rich formalisation that be the basis for future formal analysis of zero-knowledge protocols◄**

**Outline** In chapter 2 . . . Then in chapter 3 we introduce the relevant background in regards to Σ-Protocols, Commitment schemes and Multi-part computations.

# 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

```
require Int.

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

Listing 2.1: EasyCrypttheories: importing definitions

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**◄

```
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'
  }
}
```

Listing 2.2: nextHopInfo: IND-CPA Game

## 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 Zero-Knowledge

## 3.2 Sigma Protocols

Originally introduced by Cramer ►**reference**◄, 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.

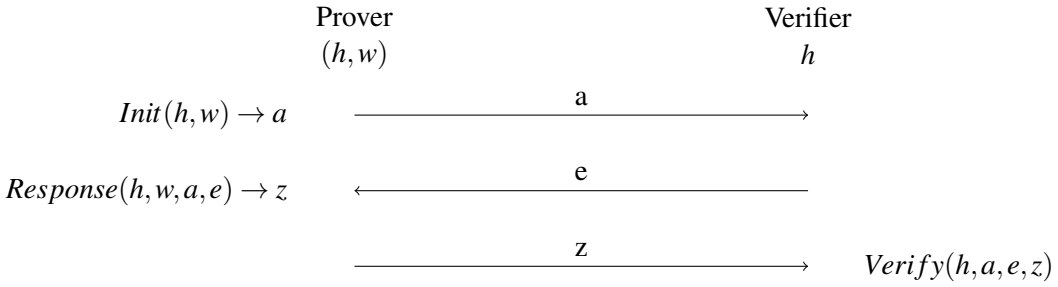


Figure 3.1: Σ-Protocol

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 [5].

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

**Definition 3.2.1** (Σ-Protocol Security)**.** 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.

►remove paragraphs and make definitions with text below.◄

**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.

**Definition 3.2.2** (Special Soundness)**.** Given a Σ-Protocol $S$ for some relation $R$ with public input $h$ and two any accepting transcripts $(a,e,z)$ and $(a,e',z')$ where both transcripts have the same initial message, $a$ and $e \neq e'$.

Then we say that $S$ satisfies 2-special soundness if, there exists an efficient ►what is efficient?◄ algorithm, which we call the "witness_extractor", that given the two transcripts outputs a valid witness for the relation $R$.

**SHVZK** ►Write zero-knowledge section and reference it◄ 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.

## 3.3 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 3.3.1** (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.

## 3.4 Multi-Part Computation (MPC)

Consider the problem, where $n$ parties, called $P_1, \ldots, p_n$, with corresponding input values $\mathbf{x} = x_1, \ldots, x_n$ want to compute a commonly known function, $f : (\text{input})^n \rightarrow$ output, such that every party agrees on the same output $y$, such that $y = f(\mathbf{x})$, but none of them learns the inputs to function, barring their own.

**►Different notions of security. We need passive.◄**

This is the exact problem that MPC aims to solve.

**Definition 3.4.1** (Correctness)**.**

**Definition 3.4.2** (d-Privacy)**.**

# Chapter 4

# Formalising commitment schemes

This section aims to give a generalised formalisation of commitment schemes and their security in a way that makes it possible and easy to reason about the security properties of arbitrary instantiations of commitment schemes. Moreover, the formalisation provides a standard interface for other protocols to interact with commitment schemes. In this section we will introduce two flavours of commitment schemes. The first version formalises key-based commitment schemes, where it is necessary for the two parties to share a key. The other is a more idealised variant of commitment schemes, which does not require the parties to share any keys between them, and only assumes they share the function specification of the commitment schemes. The latter variant is usually instantiated by one-way/hash functions.

## 4.1 Key-based commitment schemes

For Key-based commitment schemes we fix to following types:

**type:** public_key
secret_key
commitment
message
randomness

Here we specially fix a type "randomness", which is responsible to making two commitments to the same message look different. Technically this randomness could just be part of the "commitment" type, which is the type defining what values commitments takes. The choice of separating the two types, however, makes the formalisations of security easier to work with, which we will see later in this section.

With the types fixed we then define a key-based commitment scheme as the following functions and procedures...

## 4.2 Key-less commitment schemes

## 4.3 Concrete instantiation: Pedersen Commitment

# Chapter 5

# Formalising Sigma-Protocols

This section will aim to formalise Σ-protocols according to the definitions set out in section 3.2, with a sufficiently general set-up to allows easy instantiation of arbitrary concrete protocols.

Moreover, we show that any protocol that adheres to this abstract specification of a Σ-Protocol can be compounded together whilst still being secure.

We then end this section by formalising the Fiat-Shamir heuristic, which allows us to make any Σ-Protocol non-interactive in the random oracle model. This also implies that Σ-Protocol are Zero-knowledge in the random oracle model, since Special honest verifier zero-knowledge ensure Zero-knowledge in the presence of an honest verifier. If we remove the verifier, then he can always be assumed honest.

This abstraction is provided by EasyCrypt's abstract theories, where we let Σ-Protocol be quantified over the following types:

- input
- witness
- message
- challenge
- response
- randomness

And a relation $R : (\text{input} \times \text{witness}) \to \{0,1\}$.

Moreover, we require that there exists an uniform distribution, for which elements of type challenge can be sampled from. Now, since distribution are also probabilistic programs within EasyCrypt , we require that sampling from the distribution is always successful. This is refereed to as the distribution being lossless.

►**Relate to figure in background chapter?**◄

We then define the Σ-protocol itself to be a series of probabilistic procedures:

```
module type SProtocol = {
  proc gen() : statement * witness
  proc init(h : statement, w : witness) : message
  proc response(h : statement, w : witness,
```

```
module Completeness(S : SigmaProtocol) = {
  proc main(h : input, w : witness) : bool = {
      var a, e, z;
      a = S.init(h,w);
      e <$ dchallenge;
      z = S.response(h, a, e);
      v = S.verify(h, a, e, z);
      return v;
  }
}.
```

Listing 5.1: Completeness game for Σ-Protocols

```
                m : message, e : challenge) : response
  proc verify(h : statement, m : message, e : challenge, z : response
    ) : bool
  proc witness_extractor(h : statement, m : message, e : challenge
    list, z : response list) : witness option
  proc simulator(h : statement, e : challenge) : message * response
}.
```

**►Procedures are allowed to share global state - even verify. important wrt. security◄**

**►Here gen is . . . ◄**

An instantiation of a Σ-Protocol is then an implementation of the above procedures, which can be seen in Listing 5.1.

We then model security as a series of games:

**Definition 5.0.1** (Completeness)**.** We say that a Σ-protocol, S, is complete, if the probabilistic procedure in 5.1 outputs 1 with probability 1, i.e.

$$\forall \&mhw, Rhw \implies Pr[\text{Completeness(S).main}(h,w)@\&m : res] = 1\%r. \tag{5.1}$$

One problem with definition 5.0.1 is that quantification over challenges is implicitly done when sampling from the random distribution of challenges. This mean that reasoning about the challenges are done within the probabilistic Hoare logic, and not the ambient logic. If we at some later point need the completeness property to hold for a specific challenge, then that is not true by this definition of completeness, since the ambient logic does not quantify over the challenges. To alleviate this problem we introduce a alternative definition of completeness:

**Definition 5.0.2** (Alternative Completeness)**.** We say that a Σ-protocol, S, is complete if:

$$\forall \&mhwe, Rhw \implies Pr[\text{Completeness(S).special}(h,w,e)@\&m : res] = 1\%r. \tag{5.2}$$

Where the procedure "Completeness(S).special" is defined as

```
  proc special(h : statement, w : witness, e : challenge) : bool = {
    var a, z, v;

    a = S.init(h, w);
```

```
    z = S.response(h, w, a, e);
    v = S.verify(h, a, e, z);

    return v;
  }
```

Now, since the alternative procedure no longer samples from a random distribution it is not possible to prove equivalence between the two procedure, but to show that this alternative definition is still captures what is means for a protocol to be complete we have the following lemma:

**Lemma 5.0.3.** Given that definition 5.0.2 then it must hold that definition 5.0.1 holds, given the same public input and witness.

This can be stated in EasyCryptas:

```
lemma special_implies_main (S <: SProtocol) h' w':
  (∀ h' w' e',
    phoare[special : (h = h' /\ w = w' /\ e = e') ==> res] = 1%r)
  =>
  phoare[Completeness(S).main : (h = h' /\ w = w') ==> res] = 1%r.
```

*Proof.* First we start by defining an intermediate game:

```
proc intermediate(h : input, w : witness) : bool = {
   e <$ dchallenge;
   v = special(h, w, e);
   return v;
}
```

From this it is easy to prove equivalence between the two procedures "intermediate" and "main" by simply inlinining the procudures and moving the sampling to the first line of each program. This will make the two programs equivalent.

Now, we can prove the lemma by instead proving:

```
lemma special_implies_main (S <: SProtocol) h' w':
  (forall h' w' e', phoare[special : (h = h' /\ w = w' /\ e = e') ==>
    res] = 1%r) =>
  phoare[intermediate : (h = h' /\ w = w') ==> res] = 1%r.
```

The proof then proceeds by first sampling $e$ and then proving the following probabilistic Hoare triplet: $true \vdash \{\exists e', e = e'\}$special(h,w,e)$\{true\}$. Now, we can move the existential from the pre-condition into the context:

$$e' \vdash \{e = e'\}\text{special(h,w,e)}\{true\}$$

Which then is proven by the hypothesis of the "special" procedure being complete.
**►variable declarations have been omitted◄** □

**Definition 5.0.4** (Special Soundness)**.** Given a Σ-Protocol $S$ for some relation $R$ with public input $h$ and two any accepting transcripts $(a,e,z)$ and $(a,e',z')$ where both transcripts have the same initial message, $a$ and $e \neq e'$.

We then define special soundness as winning the game defined in Listing 5.2 with probability 1.

```
module SpecialSoundness(S : SProtocol) = {
  proc main(h : statement, m : message, c c' : challenge, z z' :
    response) : bool = {
    var w, v, v';

    v  = S.verify(h, m, c, z);
    v' = S.verify(h, m, c', z');

    w = S.witness_extractor(h, m, c, c', z, z');

    return (c <> c' /\ (R h w) /\ v /\ v');
  }
}.
```

Listing 5.2: 2-special soundness game

**Definition 5.0.5** (Special Honest Verifier Zero-Knowledge)**.** Given a Σ-Protocol $S$ for some relation $R$ with public input $h$ and witness $w$, then $S$ is said to be Special Honest Verifier Zero-knowledge (SHVZK), if there exists a simulator that given input $h$ outputs a transcript $(a,e,z)$, which is indistinguishable from the transcript observed by running $S$ on input $h,w$.

**►define indistinguishability◄ ►Statistical and computation indistinguishability in EC?◄**

**Definition 5.0.6.** S is said to be a Σ-Protocol if it implements the procedures in figure 5 and satisfy the definitions of completeness, special soundness, and special honest verifier zero-knowledge.

**►Argue that games corresponds to original definitions◄**

**►To prove composition we assume to following relations to be true . . . and this only hold if both inputs are in the domain of R.◄**

## 5.1 Composition Protocols

Given our formalisation of Σ-Protocols we now show that our formalisation composes is various ways. More specially it is possible to prove knowledge of relations compounded by the logical operators "and" and "or". The benefit of this is...

### 5.1.1 AND

**►Needed to change definition of games to relate challenges◄**

In this section we prove the AND construction to be a Σ-protocol by definition 5.0.6.

**►Write protocol as diagram?◄ ►Write procedure definitions?◄**

**Lemma 5.1.1** (AND completeness)**.**

$$Rhw \wedge \Pr[Completeness.main(S_1)] = 1 \wedge \Pr[Completeness.main(S_2)] = 1 \implies \Pr[Completeness.main(AND(S_1,S_2))] = 1$$

**►Missing parameters and quantification◄**

*Proof.* The proof proceeds by game-hopping to:

```
proc intermediate(h : input, w : witness) = {
  (h_1, h_2) = h;
  (w_1, w_2) = w;
  v_1 = Completeness(S_1).main(h_1, w_1)
  v_2 = Completeness(S_2).main(h_2, w_2)

  return v_1 ∧ v_2
}
```

This intermediate game will return true with probability 1 since both $v_1$ and $v_2$ will always return true by our assumption of completeness for $S_1$ and $S_2$. □

**Lemma 5.1.2** (AND special soundness)**.**

$$\begin{aligned} & R(h_1,h_2)(w_1,w_2) \\ \wedge\, & \Pr[S_1.verify(h_1,w_1,e,z)] = 1 \\ \wedge\, & \Pr[S_2.verify(h_2,w_2,e',z')] = 1 \\ \implies & \Pr[SpecialSoundness(AND(S_1,S_2)).main] = 1 \end{aligned}$$

### 5.1.2 OR

**►Needed to change relation◄**

### 5.1.3 Fiat-Shamir Heuristic

### 5.1.4 Concrete Example: Schnorr protocol

# Chapter 6

# 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.

►**Mention zkSNARKS?**◄

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 [1].

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 . . .

Another important factor of zero-knowledge compilers is the reduce the proof size. ZK proofs are notoriously expensive to run.

## 6.1 ZKBOO

The ZKBoo protocol, which was invented by Giacomelli et al. [6], is a zero-knowledge compiler for relations, which can be expressed as the pre-image of a group homomorphism, i.e.

$$R\ \mathrm{h}\ \mathrm{w} = \phi(w) = h$$

Where $h$ is the public input and $w$ is the witness.

The principle idea of this protocol is based on a technique called "MPC in the head". Recall from section 3.4, that the theory of Multipart Computation allows us, for any given function taking $n$ inputs to securely compute output $y$ for the function. We then have by definition 3.4.2 that as long as less than $d \leq n$ views are available to the adversary, then the inputs to the function are private.

Now, if we instead of proving the knowledge of a witness satisfying $\phi(w) = h$ we revealed a run, i.e. the views, of a MPC protocol computing the above function, but with the witness distributed amongst all parties then we have the claims:

**Lemma 6.1.1.** By correctness (definition 3.4.1), and assuming that the input share to the parties where indeed a valid distribution of the witness, then we can conclude that the witness is the pre-image of the public input

**Lemma 6.1.2.** By d-privacy (definition 3.4.2) if less than $d$ views are revealed, then the witness is not revealed.

Which ultimately gives us:

**Lemma 6.1.3.** From lemma 6.1.1 and lemma 6.1.2 it follows that MPC can be used to create an zero-knowledge protocol for the pre-image of a group homomorphism.

Before we go into proving the above lemmas, we first need to address how we are to actually perform the MPC protocol. Having to depend on $n$ different parties to perform a zero-knowledge protocol is not a feasible solution, so instead of recruiting the help of $n$ external parties to perform the protocol we instead perform the entire protocol locally by simulating every party in the protocol. This is commonly refereed to as performing the protocol "in the head".

**►implication on security by having all parties locally◄**

The following section we then, in order, be dedicated to explaining how to distribute the witness to multiple parties, and decomposing the original single input into an MPC protocol computing the function take $n$ inputs. Then, having properly defined the MPC protocol, we will show how to use the "MPC in the head" protocol to make a zero-knowledge protocol to and prove lemma 6.1.3.

### 6.1.1 (2,3)-Function Decomposition

(2,3)-Function decomposition is a general technique given a function $f : X \to Y$ and a input value $x \in X$ to compute the value $f(x)$ by splitting it into three computational branches, where revealing tow of the branches reveals no information about the input $x$. Through-out this section we will simply refer the a (2,3)-Function decomposition as $\mathscr{D}$.

**►General description◄**

Based on the security definitions from MPC (Section 3.4) we can then define the two necessary properties from [6] for security of our (2,3)-Function decomposition, namely, correctness and privacy.

**Definition 6.1.4** (Correctness)**.** A (2,3)-decomposition $\mathscr{D}$ is correct if $\forall x \in X, \Pr[\phi(x) = \Phi^*_\phi(x)] = 1$. **►Change notation to account for randomness◄**

**Definition 6.1.5** (Privacy)**.** A (2,3)-decomposition $\mathscr{D}$ is 2-private if it is correct and for all challenges $e \in \{1,2,3\}$ there exists a probabilistic polynomial time simulator $S_e$ such that:

$$\forall x \in x, \left(\{\mathbf{k}_i, \mathbf{w}_i\}_{i\in\{e,e+1\}}, \mathbf{y}_{e+2}\right) \equiv S_e(x)$$

Where $\left(\{\mathbf{k}_i, \mathbf{w}_i\}_{i\in\{e,e+1\}}, \mathbf{y}_{e+2}\right)$ is produced by running $\mathscr{D}$ on input $x$

#### (2,3)-Function Decomposition for Arithmetic circuits

**►For the scope of this thesis we simplify this assumption, and only look at functions, which can be represented as a arithmetic circuit in some finite integer ring.**

**If the function is represented as an integer arithmetic circuit, then a general technique exists for perform the (2,3)-Decomposition◄**

**►Describe technique for 2,3 decomp of arith.◄**

### 6.1.2 Making the protocol zero-knowledge

**►Proof size of ZKboo◄**

# Chapter 7

# Formalising ZKBoo

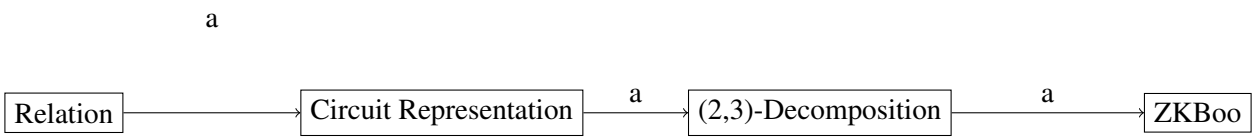


Figure 7.1: Outline of ZKBoo formalisation

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"

Thought out this section we assume that all challenges are sampled within the set of $\{1,2,3\}$ and define that $3+1=1$.

**►François et al. formalised Yao's protocol with a circuit representation similar to mine◄**

## 7.1 Formalising Arithmetic circuits

To express arbitrary computations in a ring of finite integer we need four gates, namely, Add to constant (ADDC), multiply by constant (MULTC), addition of two wires (ADD), and multiplication of two wires (MULT).

For an arithmetic circuit consisting of the aforementioned four gates we define the following notation:

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

type circuit = gate list.
```

Listing 7.1: Type declaration of gates

**Definition 7.1.1** (Arithmetic Circuit)**.** An Arithmetic circuit is a graph C = (W, G, i, o) where W is the internal wires between the gates and G is the set of gates within the circuit, finally i is the input gate and o is the designated output gate, for which there must exists a path from i to o in W.

... ►**parent/child nodes**◄ ►**Values flow through wires, but wires are not needed**◄

To express in a well-defined way it is necessary to define a computational order. Since our circuit is expressed as a graph with a singular start and end node it is possible to define the notion of depth. The depth of a node n defined as the number of nodes between i and n. Nodes at the same level can be computed in parallel since ...

We simplify our notion of the computation order and remove all parallel computations. To achieve this we define a linear order on the gates by perform a breadth-first numbering of the nodes starting from the input gate i and ending at o.

By performing this we are ensure that no nodes are computed before their parent nodes have been computed.

This ordering allows us to convert the graph representation into a list representation, where the gate at index $i$ is the node with index $i$ in the BFS numbering of the graph. But we need to encode W into this list. To do this we encode the information about input wires into the types of the gates themselves, as seen in figure 7.1. A gate is then a type, which defined its operation along with a tuple $(l, r)$ where $l$ is the index of left input wire and $r$ is the index of the right input wire. In the case of unary gates like ADDC and MULTC the tuple is $(l, c)$ where l is the input wire and $c$ is the constant used in the computation.

**Definition 7.1.2** (List representation of arithmetic circuits)**.** To represent an arithmetic circuit as a list we ...

►**Tikz example showing the two representations**◄

**Definition 7.1.3** (Valid circuit)**.** An arithmetic circuit in list representation C is valid if:

- For every entry $i$ in the list it holds that
  - C[i] is a gate type
  - the input wires of C[i] have index less than $i$.
  - the input wires of C[i] have index greater than or equals to 0.

```
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).
```

Listing 7.2: Circuit evaluation function

By representing gates as types encoding wire information we can use EasyCrypt's type system to pattern matching system to perform computations relevant to the gate type, as seen in 7.2.

From this representation of circuits as a list of gates, where gates are types, it is possible to define the semantic meaning of this representation, by defining the evaluation function, which can be seen in figure 7.2. 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, by the circuit being valid and the execution order follows the ordering of indexes, we know that if we are computing index $i$ of the circuit, then indices $[0 \dots i-1]$ 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 circuit then follows from the same fact, that gates 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 next entry in the list and 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.

"state" can then be defined as

$$\text{state}_0 = \text{input value}$$
$$\text{state}_{i>0} = \dots$$

We then have that any valid circuit c can be compute to a value y as $\text{eval\_circuit}(\text{c},$

```
proc compute(c : circuit, w1 w2 w3 : view, k1 k2 k3 : random_tape)
 = {
  while (c <> []) {
    g = oget (ohead c);
    r1 <$ dinput;
    r2 <$ dinput;
    r3 <$ dinput;
    k1 = (rcons k1 r1);
    k2 = (rcons k2 r2);
    k3 = (rcons k3 r3);
    v1 = eval_gate g 1 w1 w2 k1 k2;
    v2 = eval_gate g 2 w2 w3 k2 k3;
    v3 = eval_gate g 3 w3 w1 k3 k1;
    w1 = (rcons w1 v1);
    w2 = (rcons w2 v2);
    w3 = (rcons w3 v3);
    c = behead c;
  }
  return (k1, k2, k3, w1, w2, w3);
}
```

Listing 7.3: Incremental decomposition procedure

[input]) = y. This can also be stated as a probabilistic procedure as Pr[eval_circuit(c, [input]) = $y] = 1$.

To reason about functions and procedures about functions we have the following lemma:

**Lemma 7.1.4** (Function/Procedure relation)**.** $\forall$ f, inputs, output: f(inputs) = output $\iff$ Pr$[f(inputs) = output] = 1$.

## 7.2 (2,3) Decomposition of circuits

In its most general form, we can define the decomposition as a procedure taking as input three views and random tapes, and a circuit and produces three new views. More specifically the decomposition work by evaluating a gate based on previously compute views, which yield new shares that can be appended to the view. This process of evaluating a single gate based on the view of evaluating the previous gate can then be repeated until all gates have been computed. This overall idea has been captured in the procedure in figure 7.3, where it is assumed access to a function eval_gate, which has signature: $eval\_gate : circuit \Rightarrow party \Rightarrow (view * view) \Rightarrow (random\_tape * random\_tape) \Rightarrow share$, where *party* is a integer in $\{1,2,3\}$ that determines which party is computing the share. **►Say that eval gate implements the function in the ZKBoo section◄**

The output of the decomposition can then be defined as summing the last share from each view that has been compute by the aformentioned procedure.

**►Define simulator...◄**

**►All functions and proof quantify over random tapes but they have been omitted from this write-up since they are just random values that need to be return to reconstruct the results◄**

### 7.2.1 Correctness

Ultimately want to prove:

**Lemma 7.2.1** (Decomposition correctness)**.**

$$\Pr[\text{eval\_circuit}(\text{c}, [\text{input}]) = y] = \Pr[\text{decomposition}(\text{c}, [\text{input}]) = y]$$

i.e. The output distribution of the two programs are perfectly indistinguishable. From lemma 7.1.4 we have that circuit evaluation always succeeds, this lemma, therefore, also implies that the decomposition always succeeds.

**Definition 7.2.2** (Correctness of views)**.** For any three views (list of shares), $w_1, w_2, w_3$, with equal length, we say that they contain valid shares of computing a circuit c, if it holds:

$$\forall 0 \leq i < sizec, w_1[i] + w_2[i] + w_3[i] = s[i]$$

where s is the list of intermediate values produces by calling $\text{eval\_circuit\_aux}$ in figure 7.2.

Additionally a share is only valid, if it has been produced by functions used by the decomposition.

$$\forall e \in \{1,2,3\} \forall 0 \leq i < sizec - 1, w_e[i+1] = \text{phi\_decomp}c[i]1w_e w_{e+1} k_e k_{e+1}/$$

To express that the views satisfy the above definition we use the notation $\textbf{Valid}(c, w1, w2, w3)$ to express that $w1, w2, w3$ are valid views for the decomposition of $c$

To prove the above lemma we first introduce a helper lemma:

**Lemma 7.2.3** (Stepping lemma for decomposition)**.** For any valid circuit c in list representation, it is possible to split the circuit into two parts $c_1, c_2$ where $c = c_1 + + c_2$ (++ is list concatenation). let $w_1, w_2, w_3$ be the resulting views of decomposing $c$ and $\textbf{Valid}(c_1, w_1, w_2, w_3)$ and let computing $c_2$ with initial views $w_1, w_2, w_3$ output views $w'_1, w'_2, w'_3$. Then $\textbf{Valid}(c, w'_1, w'_2, w'_3)$.

Alternatively this is stated as:

$$\textbf{Valid}(c_1, w_1, w_2, w_3) \implies \Pr[\text{compute}(c_2, w_1, w_2, w_3) : \textbf{Valid}(c, w'_1, w'_2, w'_3)] = 1$$

*Proof.* The proof proceeded by induction on the list c.

- Base case $c = []$: trivially true since an empty circuit is the identity function.
- Induction step $c = c' + + [g]$:
  - Inline definitions to get compute_stepped
  - We use to induction hypotheses to compute c' which give us $\textbf{Valid}(c_1, w_1, w_2, w_3)$.
  - We then need to prove, that we can compute any gate on top of the valid views to produce a new set of valid views.

□

*Proof of lemma 7.2.1.* By unfolding the definition we are left with proving that the last share from each of the views produced by compute are equal to the output of evaluating the circuit, which is true by lemma 7.2.3 □

**►Our formalisation differs by imposing stricter restrictions on the shares computed...◄**

### 7.2.2 2-Privacy

To prove 2-Privacy we need to first define a simulator capable of producing indistinguishable views for two of the parties. To simulator is given by the procedure simulate and function simulator_eval in figure 7.4. simulator_eval is a function that evaluates a single gate from the point of view of party "p". For the cases of evaluating ADDC ADD MULTC gates the simulator simply calls the eval_gate function, since these computations are performed "locally" for each party, i.e. they do not depend on the shares of the other two parties in the protocol. When evaluating MULT gates shares needs to be distributed amongst the parties, but to evaluate the output of the MULT gate for any given party it only depends on the parties own share and the share of the "next" party, i.e. for party one he only depends on the shares from himself and the shares from party two. Since the simulator simulates the view of party $e$ and $e+1$ the view of party $e$ can be computed normally with the eval_gate function. For simulating the view of party $e+1$ we use the fact that shares should be uniformly random distributed, and simply sample a random value for the view. In this case the view of party $e$ can always be computationally reconstructed by looking at the view of party $e+1$, but the view of party $e+1$ cannot be verified, since the view of party $e+2$ is unknown, which makes it seem valid.

The procedure simulate is simply a wrapper around simulator_eval, which is responsible for constructing the views and sampling randomness.

To compare the views output by the simulator and the ones produced by the decomposition we fix two procedures real and simulated, where the first return two views and the final share of the third view and the latter returns the two views output by the simulator and a fake final share of the thrid view. These procedures can be seen in figure 7.5.

We are then ready to state 2-privacy as the following lemma:

**Lemma 7.2.4** (Decomposition 2-Privacy)**.** We say that the decomposition protocol offers 2-Privacy, if the output distributions between real and simulated are indistinguishable.

This can be stated in rPHL as:

$$h \in \textbf{Domain}(R) \implies \mathit{equiv}[\mathit{real} \sim \mathit{simulated} := \{e,h\} \implies = \{\mathit{res}\}].$$

To prove this lemma we first prove a that running compute and simulate with the same random choices will produce indistinguishable views corresponding to the challenge and summing the output shares of compute will yield the same value as evaluating the circuit. This effectively inlines the correctness property in the proof of the simulator. This is necessary to be able to reason about the existence of the view of

```
op simulator_eval (g : gate, p : int, e : int, w1 w2 : view, k1 k2 k3
    : int list) =
with g = MULT inputs =>
  if (p - e %% 3 = 1) then (nth 0 k3 (size w1 - 1)) else eval\_gate g
     p w1 w2 k1 k2
with g = ADDC inputs =>
    eval\_gate g p w1 w2 k1 k2
with g = MULTC inputs => eval\_gate g p w1 w2 k1 k2
with g = ADD inputs => eval\_gate g p w1 w2 k1 k2.

proc simulate(c : circuit, e : int, w1 w2 : view, k1 k2 k3 :
    random_tape) = {
  while (c <> []) {
    g = oget (ohead c);
    r1 <$ dinput;
    r2 <$ dinput;
    r3 <$ dinput;
    k1 = (rcons k1 r1);
    k2 = (rcons k2 r2);
    k3 = (rcons k3 r3);
    v1 = simulator_eval g e e w1 w2 k1 k2 k3;
    v2 = simulator_eval g (e+1) e w2 w1 k1 k2 k3;
    w1 = (rcons w1 v1);
    w2 = (rcons w2 v2);
    c = behead c;
  }
```

Listing 7.4: Simulator

```
proc real((c,y) : statement, w : witness, e : challenge) = {
    (y1,y2,y3,w1,w2,w3) = compute(c);
    return (we,we+1,ye+2)
}

proc simulated((c, y) : statement, e : challenge) = {
    (we,we+1) = simulate(c, e);
    ye = last we;
    ye+1 = last we+1;
    ye+2 = y-(ye+ye+1)
    return (we,we+1,ye+3)
}
```

Listing 7.5: Real/Simulated view of decomposition

party $e+2$, which would make the views produced by the simulated equal to honestly produces views. More specifically the inlined correctness property gives us ...

This is stated as the following lemma:

**Lemma 7.2.5.** Given a valid arithmetic circuit in list representation with challenge $e$ and intermediate circuit computations/state $s$ the following holds:

$$equiv[compute \sim simulated : = \{h, e, w_e, w_{e+1}\} \implies = \{w'_e, w'_{e+1}\}]$$

Moreover, we require that the input views to compute $w_1, w_2, w_3$ satisfy:

$$\forall 0 \leq i < sizew1, w_1[i] + w_2[i] + w_3[i] = s[i]$$

Additionally this property most also hold for the views $w'_1, w'_2, w'_3$ produced by running compute. This is equivalent to part of the **Valid** property used in the proof of correctness.

*Proof.* We proceed by induction on the list representation of the circuit c:

- Base Case $c = []$ : trivial
- Induction Case $c = g :: cs$ :
- Write this as program steps like in [3]?

$$compute(g :: gs, w_1, w_2, w_3) \sim simulate(g :: gs, w'_1, w'_2, w'_3)$$

□

*Proof of lemma 7.2.4.* By applying lemma 7.2.5 we have that the views output by both procedures are indistinguishable. All have left to prove is that $y_{e+2}^{real} \equiv y_{e+2}^{simulated}$. To prove this, we use the correctness definition, saying that the shares of the real views always sum to the intermediate values of computing the circuit, therefore: $y_{e+2}^{real} = y - (y_e^{real} + y_{e+1}^{real})$. Now, since $(y_e^{real} + y_{e+1}^{real}) \equiv (y_e^{real} + y_{e+1}^{real})$ it follows that

$$\begin{aligned} y_{e+2}^{simulated} &= y - (y_e^{simulated} + y_{e+1}^{simulated}) \\ &\equiv y - (y_e^{real} + y_{e+1}^{real}) \\ &= y_{e+2}^{real} \end{aligned}$$

□

## 7.3 ZKBOO

Since the ZKBoo protocol is an instantiation of a Σ-Protocol we start by defining the types as specified in section 5 and instantiating the Σ-Protocol framework.

```
type statement  = (circuit * int).
type witness    = int.
type message    = output * output * output * Commit.commitment *
    Commit.commitment * Commit.commitment.
type challenge  = int.
type response   = (random_tape * view * random_tape * view).
```

We then axiomatize that the challenge is always a integer in $\{1,2,3\}$. Since it is not supported by the type system?

We assume the existence of an idealised commitment scheme which satisfies ...

We then define a predicate for validating a view:

```
pred valid_view p (view view2 : view) (c : circuit) (k1 k2 :
    random_tape) =
  (forall i, 0 <= i /\ i + 1 < size view =>
    (nth 0 view (i + 1)) = phi_decomp (nth (ADDC(0,0)) c i) i p view
    view2 k1 k2).

op valid_view_op p (view view2 : view) (c : circuit) (k1 k2 :
    random_tape) =
  (foldr (fun i acc,
            acc /\ (nth 0 view (i + 1)) = phi_decomp (nth (ADDC(0,0))
    c i) i p view view2 k1 k2)
    true (range 0 (size view - 1))).
```

Predicates allows us to use quantifiers to assert properties, which are nice to reason about especially in pre and post condition of procedures. Predicates, however, have no computation aspect to them and are pure logical. A predicate, therefore, cannot be used within procedures, since they are not required to be computable(?). We, therefore, add function which check the same properties, but without the use of quantification. This requires us to use looping instead, which can be harder to reason about. Specially, it is much harder to reason about the *i*'th index of the view being computed correctly given that valid_view_op returns true, whilst it immediately follows from the predicate.

To bridge the gap between the computational and logical worlds we introduce the following lemma:

**Lemma 7.3.1** (valid_view predicate/op equivalence)**.** $\forall$ p, w1, w2, c, k1, k2: valid_view p w1 w2 c k1 k2 $\iff$ valid_view_op p w1 w2 c k1 k2

With a way to validate the views we can instantiate the ZKBoo protocol from section 6.1 as a $\Sigma$-Protocol in our formalisation by implementing the algorithms from figure 5, which can be seen in figure 7.6. **►Assumes existence of decomposition protocol◄ ►Assumes existence of ideal commitment functionality◄**

We then, automatically, by our formalisation of $\Sigma$-Protocols get definition of security and only need to prove them ... **►wording◄**

**Lemma 7.3.2.** ZKBoo satisfy correctness definition 5.0.1.

*Proof.* We start by observing that committing to $(w_i, k_i)$ in init and the verifying the commitment in verify is equivalent to the correctness game for commitment schemes defined in 4.

We therefore inline the completeness game, and replace the calls to the commitment procedures with the correctness game:

```
proc intermediate_main(h : statement, w : witness, e : challenge) = {
  (c, y) = h;
  (x1, x2, x3) = Phi.share(w);
  (k1, k2, k3, w1, w2, w3) = Phi.compute(c, [x1], [x2], [x3]);
  y_i = last 0 w_i;
```

```
global variables = w1, w2, w3, k1, k2, k3.

proc init(h : statement, w : witness) = {
  (x1, x2, x3) = Share(w);
  (k1, k2, k3, w1, w2, w3) = Decompose(c, x1, x2, x3);
  c_i = Commit((w_i,k_i));
  y_i = last 0 w_i;
  return (y1, y2, y3, w1, w2, w3);
}

proc response(h : statement, w : witness, m : message, e : challenge)
    = {
  return (k_e,w_e,k_{e+1},w_{e+1})
}

proc verify(h : statement, m : message, e : challenge, z : response)
   = {
  (y1, y2, y3, c1, c2, c3) = m;
  (c, y) = h;

  (k1', w1', k2', w2') = open;
  valid_com1 = verify (w'_e,k'_e) c1;
  valid_com2 = verify (w'_{e+1},k'_{e+1}) c2;
  valid_share1 = last 0 w'_e = y1;
  valid_share2 = last 0 w'_e = y2;
  valid = valid_view_op 1 w'_1 w'_2 c k'_1 k'_2;
  valid_length = size c = size w'_e-1 /\ size w'_1 = size w'_2;

  return y = y1 + y2 + y3 /\ valid_com1 /\ valid_com2 /\ valid_share1
    /\ valid_share2 /\ valid /\ valid_length
}
```

Listing 7.6: ZKBoo Σ-Protocol instantiation

```
  valid_com1 = Correctness.main((w_e,k_e));
  valid_com2 = Correctness.main((w_{e+1},k_{e+1}));
  commit((w_{e+2},k_{e+2}));
  valid_share1 = valid_view_output y_e w_e;
  valid_share2 = valid_view_output y_{e+1} w_{w+1};
  valid = valid_view_op e w_e w_{e+1} c k_e k_{e+1};

  valid_length = size c = size w_e - 1 /\ size w_e = size w_{e+1};

  return valid_output_shares y y1 y2 y3 /\ valid_com1 /\ valid_com2
    /\ valid_share1 /\ valid_share2 /\ valid /\ valid_length;
}
```

Listing 7.7: Intermediate game for completeness

We then prove the correctness of intermediate_main by showing that the procedure returns true for any $e \in \{1,2,3\}$.

**Case** $e = 1$: For the procedure to return true we need to following to hold:

- All variables must be true
- commit must be lossless such that procedure always terminates
  - commit must be lossless by completeness of commitment scheme
  - formalise this?

The other cases are the same.

□

**Lemma 7.3.3.** **►Explain idealised commitment scheme◄ ►Formalise perfect hiding◄. ►Perfect hiding must use alternative non-game-based definition◄** Assuming perfect hiding, ZKBoo satisfy Special Honest Verifier Zero-knowledge definition 5.0.5

*Proof.* To prove shvzk we show that running the real and the ideal procedures with the same inputs and identical random choices produces indistinguishable output values. The proof the proceeded by casing on the value of the challenge $e$. To proof for the different values are identical so we suffice in showing only the case of $e = 1$. When $e = 1$ the two procedures are:

By 2-Privacy of the decomposition we know that compute and simulate are indistinguishable procedures, when the view $e_{e+2}$ produced by compute is never observed. This is fortunately the case here, we when calling the sub-procedure simulate in the ideal case, we know that the properties ensured by the correctness of the decomposition must also hold in the ideal case. This means that the views produced by simulate must also produce views which satisfy the correctness property for the views 7.2.2. This is enough to make the verify procedure return true.

We, therefore, only need to argue that $c_{e+2}$ are identically distributed for both of the procedures. In the real case $c_{e+2}$ is simply committing to the view produces by the decomposition. In the ideal case, however, it is a commitment to a list of random values but due out assumption of perfect hiding these two commitments are identically distributed.

The rest of the out values are indistinguishable by the 2-Privacy property.

□

```
proc real(h, w, e) = {
  (x1, x2, x3) = Share(w);
  (k1, k2, k3, w1, w2, w3) =
    compute(c, x1, x2, x3);
  c_i = Commit((w_i,k_i));
  y_i = last 0 w_i;

  a = (y1,y2,y3,c1,c2,c3)
  z = (k_e,w_e,k_{e+1},w_{e+1})

  if (verify(h,a,e,z)) {
    Some return (a,e,z);
  }
  return None;
}
```

```
proc ideal(h, e) = {
    (* From Decomposition *)
    (w_e,w_{e+1},y_{e+2}) = simulated;

    (* Generate random list of
    shares *)
    w_{e+2} = dlist dinput (size w1);
    k_{e+2} = dlist dinput (size k1);
    y_e = last 0 w_e;
    y_{e+1} = last 0 w_{e+1};
    c_i = commit((w_i,k_i));
    a = (y1,y2,y3,c1,c2,c3);
    z = (w_e,w_{e+1});

    if (verify(h,a,e,z)) {
      Some return (a,e,z);
    }
    return None;
}
```

**Lemma 7.3.4.** Given a commitment scheme, where an adversary can produce three pairs commitments, where at least one pair has different openings with probability $p$, then ZKBoo satisfy the 3-Special Soundness property with probability $p$.

*Proof.* The proof has three distinct steps. First, we show that the inputs z1, z2, z3 to witness_extractor procedure will be valid and consistent openings revealing the views $w_1, w_2, w_3$ which has been produced by the same call to compute with probability $1-p$. Next, we show that given views $w_1, w_2, w_3$ which correspond to three views produced by the same call to compute, then a valid witness can be extracted. Ultimately, we show that Special Soundness game can be won with probability $(1-p)$

**Consistent views** For any one of the openings only the view corresponding to the challenge needs to be provably constructed by the decomposition. This means that we need all three openings to conclude that the views has all been produced by the decomposition. However, by breaking the binding property it is possible to provide an opening, which might not have been produced by the same call to compute. For example, z1 reveals $w_1^1, w_2^1$ whilst z2 reveals $w_2^2, w_3^2$, but if the binding property is broken, then $w_2^2$ might be a valid view, but it has been produced with randomness that is different from $w_2^1$. The probability of breaking the binding property is $p$, hence the probability of all openings being to the same views is $1-p$.

**Witness extraction** Given that all openings correspond to the same call of compute and **Valid**$(c, w_1, w_2, w_3)$ we must then show that $w = w_1[0] + w_2[0] + w_3[0] \implies y =$ eval_circuit$(c, w)$ i.e. the witness is the sum of all the input shares to the parties of the decomposition.

$$
\begin{aligned}
&\mathrm{eval\_circuit}(c, w_1[0] + w_2[0] + w_3[0]) = y \\
\iff &\Pr[\mathrm{eval\_circuit}(c, w_1[0] + w_2[0] + w_3[0]) = y] \\
&= \Pr[(w_1', w_2', w_3') \leftarrow \mathrm{compute}(c, w_1[0], w_2[0], w_3[0]); \left( \sum_{i \in \{1,2,3\}} \text{last } w_i' \right) = y]
\end{aligned}
$$

Now, we can it is possible to show that for each iteration of the while-loop in compute it must preserve the property that

$$\forall j \in \{1,2,3\} \forall 0 \leq i < \text{size } w_j' : \; w_j'[i] = w_j[i]$$

by **Valid**$(c, w_1, w_2, w_3)$, which asserts that each view has precisely been constructed by the compute procedure with the appropriate randomness.

Moreover, we have that $\sum_{i \in \{1,2,3\}} \text{last } w_i = y$ since the transcripts containing the views are accepted by the verify procedure, which proves that the witness can be reconstructed if all the views of the decomposition is given.

**Special Soundness** The soundness game can be restated as the following procedure returning true with probability $1 - p$

```
proc soundness(h, m, z1, z2, z3) = {
  v = consistent_views(h, m, z1, z2, z3);
  w = witness_extractor(h, m, [1;2;3], [z1;z2;z3]);

  if (w = None \/ !v) {
    return false;
  } else{
    w_get = oget w;
    return R h w_get;
  }
}
```

Which follows directly.

□

►**Formal verification does not tell us about efficiency**◄

►**Concluding remarks about the formalisation**◄ Formal proofs like these can help us gain insight into the security of the protocols. The security of the ZKBoo protocol is entirely dependent on the security properties of the underlying decomposition and commitment scheme being state properly. For example, if the decomposition does not ensure that all the shares in the views has been produced according to the decomposition algorithm, then ZKBoo offers no guarantee about

Moreover, they help us expose some of the more subtle details important for proving security of cryptographic protocols, like requiring certain procedures to be lossless since...

```
proc witness_extractor(h : statement, a : message, e : challenge list
    , z : response list) = {
  [z1; z2; z3] = z;
  (k1'', w1'', k2'', w2'') = z1;
  (k2', w2', k3'', w3'') = z2;
  (k3', w3', k1', w1') = z3;

  if (k1'' = k1' /\ w1'' = w1' /\ k2'' = k2' /\ w2'' = w2' /\ k3'' =
    k3' /\ w3'' = w3') {
    ret = Some( (first 0 w1') + (first 0 w2') + (first 0 w3') );
  } else {
    ret = None;
  }
  return ret;
}
```

Listing 7.8: ZKBoo witness extractor

# Chapter 8

# Related Work

Sigma protocols has been done in a lesser extend in EasyCrypt. Much of the same work has been done in Isabelle/CryptHOL by Butler et al..

Barthe et al. formalised Σ-Protocols within CertiCrypt, and proved the security of the $\Sigma^{\phi}$-Protocol, which proves knowledge of the pre-image of a group homomorphism. ZKBoo protocol described in section 6.1 and the $\Sigma^{\phi}$-Protocol prove knowledge of the same relation, but ZKBoo has reduced proof size? ►**Examine differences**◄.

phi assumes the group homomorphism to be special?

►**Differences between the works**◄

Commitment schemes has been formalized in EasyCryptby Metere and Dong.

►**Differences between the works**◄

formalised general zero knowledge compilers have been explored, with some notable work by Almeida et al. and PINOCCIO.

►**Differences between the works**◄

# Chapter 9

# Conclusion

►conclude on the problem statement from the introduction◄

# Bibliography


[1] 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.

[2] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[3] Gilles Barthe, Daniel Hedin, Santiago Zanella-Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of Sigma-protocols. In *23rd IEEE Computer Security Foundations Symposium, CSF 2010*, pages 246–260. IEEE Computer Society, 2010. URL http://dx.doi.org/10.1109/CSF.2010.24.

[4] David Butler, Andreas Lochbihler, David Aspinall, and Adria Gascon. Formalising Σ-protocols and commitment schemes using crypthol. Cryptology ePrint Archive, Report 2019/1185, 2019. https://eprint.iacr.org/2019/1185.

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

[6] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. *IACR Cryptology ePrint Archive*, 2016:163, 2016. URL http://eprint.iacr.org/2016/163.

[7] Roberto Metere and Changyu Dong. Automated cryptographic analysis of the pedersen commitment scheme. *CoRR*, abs/1705.05897, 2017. URL http://arxiv.org/abs/1705.05897.