Formalising Sigma-Protocols and commitment schemes within EasyCrypt

Nikolaj Sidorenco, 201504729

Master's Thesis, Computer Science

June 4, 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, June 4, 2020.

CONTENTS

1 INTRODUCTION

►Maybe add database example from CertiCrypt paper?◄

In recent years, blockchains have been a breakthrough in the area of secure, decentralized, computing on an open network. At its core, a blockchain provides a distributed ledger/database. Blockchain has in particular caught interest from the financial sector, namely from bitcoins which were the first to use the blockchain as a distributed ledger, where each monetary transaction is first publicly verified and then appended to the blockchain. This function is similar to how a bank would process transactions, but with two distinct differences; all transactions are publicly available and the transactions are verified by the user of the blockchain rather than a central authority, e.g. a bank [14].

The introduction of bitcoins has since led to a myriad of different blockchains with unique focal points. Notably Ethereum, ZCash, and Concordium. Ethereum extends the original design of the blockchain with a rich programming language to allow for so-called "smart contracts". Programs written as a smart contract can then be added to the blockchain and then computed by the joint computational power of the blockchain. A recent example of this is the *Board-room voting protocol* [12], which is a zero-knowledge based protocol that allows a few people to participate in an online vote where the individual votes are confidential but the final tally of the vote is accessible to the voters. Moreover, smart contracts can be used to realise multi-party computation protocols, which allows specific users to jointly compute on private data through the blockchain, whilst only learning the result of the computation, but not the private data.

ZCash and Concordium more predominantly deal with the privacy issues relating to the blockchain. In ZCash every transaction has the possibility of being performed completely anonymously. This is in contrast to bitcoins, where every transaction is pseudonymous, meaning that every transaction can be traced back to an identifier also called a pseudonym, but the users' real identity cannot necessarily be identified.

The level of privacy ZCash provides, however, lacks compliance with regulations like "Know Your Customer" (KYC) and "Anti Money Laundering" (AML), which require financial institutions to be able to trace money of illicit origin. This is the problem that the Concordium blockchain has tried to solve with its "ID-layer", which grants its users total privacy under normal use but, also enables authorities to revoke the privacy of certain users if they deem it necessary[fn:id-layer].

Common for these three blockchains are their reliance on zero-knowledge. A zero-knowledge proof is a core primitive in cryptography which allows two parties, Alice and Bob, to share a relation R and public input x. Alice then knows some secret input y such that R(x, y) is true, i.e. Alice's secret makes the relation true. A zero-knowledge proof is then the result of running a protocol, which can be given to Bob to convince him that Alice indeed knows the value y, but without Bob attaining any information about the value y.

For ZCash and Concordium zero-knowledge protocols are deeply embedded within the functionality of the blockchain itself: The zero-knowledge proofs are used to prove ownership of an account, without revealing your personal information.

For Ethereum many protocols which depends on zero-knowledge can be implemented as smart contracts. The earlier example of board-room voting is such a smart contract, but many more exists, for example, the Ethereum Aztec library https://www.aztecprotocol.com.

Since zero-knowledge is essential for some blockchain applications, but also other cryptographic protocols, numerous techniques exist that proving zero-knowledge for any arbitrary relation. These are known as zero-knowledge compilers[fn:zk-overview].

A zero-knowledge compiler takes in a triplet of (relation, public input, secret input), where the relation is usually expressed as a computable function or mathematical relation. The triplet is then translated into an intermediate representation, This representation is usually either an Arithmetic/Boolean circuit or a constraint system. This is process is referred to as the /front end/.

The intermediate representation is then fed into the /back end/, which compiles it into a zero-knowledge argument that can be sent to the other parties to prove knowledge of the secret.

Most zero-knowledge compilers differ in their combination of front end and back end. Different back ends usually offer significant run time differences, i.e. one back end might be more efficient for relations that are expressed as short functions. Front ends usually differ in what relation they accept. A front end like libsnark's[fn:libsnark] accepts relations written as c functions, while others target languages like Rust or JavaScript.

Because of the many combinations of front ends and back ends, standardisation efforts have been commenced. One example of such a proposed standardisation is the zkinterface proposed by the https://zkproof.org community. This standardisation aims to allow the users to match any of the front ends to any of the other back ends. An example of this could be combining a Rust front end with the libsnark back end.

This standardisation effort is two-fold: first, it allows the user to pick the back end that is the most efficient for their use case. Moreover, having a standardisation is a sign of a more maturing field. The protocols considered for the standardisations have proven themselves efficient and reliable. The widespread adoption of a select few zero-knowledge compilers makes them an ideal target for formal verification since they are generally applied to longer-lasting programs, where it is not always possible to change the underlying zero-knowledge implementation.

Formal verification of protocols like zero-knowledge compilers has recently become more attainable thanks to proof assistants like EasyCrypt and CryptHOL, which enables researchers to formally reason about cryptographic protocols using the "game-based" approach [5].

In the game-based approach, security is modelled as a game against an adversary where the adversary's goal is to break the indented design of the protocol. This is usually done by a series of game reductions where it is proven that the probability of the initial game is equivalent to winning another game, which is easier to reason about. Ultimately a sequence of game reductions leads to a final game, which is either mathematically impossible for the adversary to win or equivalent to a difficult problem, like the discrete logarithm problem.

The benefit of using tool supporting game-based security like EasyCrypt is that it becomes possible to formally verify protocols in a representation very close to the one in cryptographic literature. This gives a more direct connection between the formal proofs and how the protocols are used in practice. These proof assistants also open the possibility of extracting the verified protocol into an efficient language, which can be run on most computers. One example of this is EasyCrypt, where a low-level language called Jasmin has been successfully embedded within [4]. Cryptographic protocols written in a low-level machine language can then, through EasyCrypt, be formally proven secure and extracted to assembly code.

The recent introduction of this embedding also indicates the possibility of exporting the high-level and formally verified implementation from EasyCrypt to a low-level implementation in Jasmin while still having the same security guarantees.

This would allow researchers to take a protocol description written in a cryptographic paper and then, almost directly, prove its security in a tool like EasyCrypt. The implementation done in EasyCrypt would then ultimately be extracted to an efficient implementation. This creates a direct link between the protocol description, the code run in practice, and the proof of security.

These advances in blockchain and formal verification research leaves an interesting gap, where it is now possible to formally verify complex cryptographic protocol like the ones utilised by the blockchain. These protocols have, thanks to the recent advances in blockchain research, seen more applications in the industry. But while showing functional correctness has proven feasible through the usage of tools like Coq, the research into proving cryptographic security of smart contracts using proof assistants has been relatively unexplored.

In this paper we will therefore look at the ZKBoo protocol by Giacomelli et al. [11], 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.

2

EASYCRYPT

In this chapter we introduce the EasyCryptproof assistant for proving security of cryptographic protocols. To do so 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 of 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 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.

```
require Int.

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

Listing 1: EasyCrypttheories: importing definitions

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 ►**Like interfaces from OO**◄ 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 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**◄

2.4 PROBABILISTIC HOARE LOGIC

►**programs are distribution transformers**◄ To formally prove security of a cryptographic protocol we commonly have to argue that some procedure perform random choices will terminate with a certain event with some probability, regardless of the random choices made during execution.

To this end pHL logic, which helps us express precisely this. To express running procedure $p(x)$ which is part of a module M we can use the following EasyCrypt notation:

$$phoare[M.q : \Psi \implies \Phi] = p$$

Which informally corresponds to: If the procedure with global variables from M is executed with any memory/state which satisfy the precondition Ψ then the result of execution will satisfy Φ with probability p.

Alternatively this can be stated as:

$$\Psi \implies \forall x, \&m. \Pr[M.q(x) @ \&m : \Phi] = p \tag{1}$$

Where we note that the first representation implicitly quantifies over all arguments to the procedure q and memories while the latter requires us to explicitly quantify over them.

To understand how the pHL logic works we adopt the notions by Barthe et al. [7], which states that procedures are “distribution transformers”. This means...

The deduction rules for pHL... Reasoning about adversarial code...

2.5 PROBABILISTIC RELATIONAL HOARE LOGIC

The pRHL logic allows us to reason about indistinguishability between two procedures under a specific pre- and postcondition. More formally the pRHL logic allows us to determine if two procedures are perfectly indistinguishability wrt. to the given pre- and postcondition.

We recall from section 2.4 that procedures can be seen as distribution transformers. By observing procedures as distribution transformers indistinguishability between procedures equates to arguing that both procedures transform their output distributions in a way that makes the post condition true.

In EasyCryptwe have the following notation for comparing two procedures:

$$equiv[P \sim Q : \Psi \implies \Phi]$$

Where Ψ is the precondition and Φ is the post condition.

Formally we say that two procedures are indistinguishability if:

$$\Pr[P @ m_1 : A] = \Pr[Q @ m_2 : B] \wedge \Psi \implies (A \iff B) \wedge m_1 \Psi m_2$$

More informally this can be understood as: The procedures P and Q running in respective memories m_1 and m_2 are indistinguishability wrt. to precondition Ψ and postcondition Φ, if the both memories satisfy the precondition. Moreover, if we can run procedure P and get event A and procedure Q to get event B then the procedures are indistinguishable if the postcondition implies that the two events are isomorphic.

When dealing with pRHL statement we have two types deduction rules; they are either one-sided or two-sided. The one-sided rules allow us to use the pHL deduction rules on either one of the two programs we are comparing in isolated. We refer to the two programs by their side of the $\sim$ operator. In the above example P is the left side and Q is the right. These one-sided rules allows us to step one of the side forward without reasoning about the other size. By doing this we alter all the term relating to which side we called the rule on.

The two-sided rules allow us to step both sides, if they are both about to call the a command of the same shape. In this sense the two-sided rules are much more restrictive, since we can only use them if the programs are similar in structure.

In particular the two-sided rules allows us to reason about random assignments and adversarial calls. Since random assignment and adversarial call are inherently indeterminable and can possible not terminate there it is not possible for our one-sided rules to step the programs forward. By using the two-sided rules this is not an issues, since if both procedures performs the indeterminable choice then it does not matter what the choice where, or if it terminated, just that both procedures performed the same choice.

This allows us to step both procedures forward under the assumption that both procedures transformed their output distribution in exactly the same way for the computation step.

2.6 DISTRIBUTIONS AND DEALING WITH RANDOMNESS

To introduce randomness/non-determinism to procedures EasyCrypt allows random assignments from distributions. EasyCrypt support this functionality in two way: sampling from a distribution and calling an adversary.

In EasyCrypt distribution are themselves procedures with a fixed output distribution. More formally a distribution in easycrypt is a monad converting a *discrete* set of events into a sub-probability mass function over said events. **►reference?◄**

When dealing with distribution we have three important characteristics:
Lossness : A procedure (or distribution) is said to be lossless if it always produces an output. More formally this means that the probabilistic mass functions sums to one. **Full :** A distribution is said to be full if it is possible to sample every element of the type the distribution is defined on from the distribution **Uniform:** A distribution is uniform if every event is equally likely to be sampled.

As an example a distribution over a type t can be defined as follows:

```
op dt : challenge distr.
```

Furthermore, we specify the distribution to be lossless, full and uniform as:

axiom: is_lossless dt.**axiom:** is_funiform dt.

We can then express a random assignment form the distribution as x <$ dt

By introducing random assignments in our procedures we change the output of the procedure from a value to a distribution over possible output values.

Moreover, with distributions it is possible to reason about indistinguishability with the use of EasyCrypt's coupling functionality...

2.7 EASYCRYPT NOTATION

We use notation $\Pr[P = b] = p$ to express that procedure P can be run with output value b with probability p We use notation $\Pr[P : A] = p$ to express that the output distribution of procedure P will satisfy A with probability p.

When comparing two procedures P and Q in the relational logic, i.e:

$$equiv[P \sim Q : \Psi \implies \Phi]$$

We use the notation x^P to denote the value variable x wrt. procedure P. Likewise, we let x^Q denote the value of x when observing the run of procedure Q.

When stating probabilistic Hoare statements on the form of equation 1 we omit the quantification of the arguments when the quantification can be inferred from the context. Furthermore, we also omit quantification over initial memory configurations.

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.

NOTATIONS While this thesis has been performed entirely in easycrypt the code in this thesis will be given in a more pseudo-code style to make it more general. For the most party we will avoid easycrypt specific notation when writing procedures and solely focus on what tools easycrypt provides us with for proving procedures.

Most notably we adopt the list indexing notation of $l[0]$ to mean the 0'th index of the list l. Formally this notation is not sound, since it does not specify what will happen if the index not exists. The is solved in EasyCrypt by declaring a default element to return, should the indexing fail. We omit omit default value form our code examples.

Moreover, when referring to indistinguishability we are referring the perfect indistinguishability unless stated otherwise.

3.1 ZERO-KNOWLEDGE

Zero-knowledge can be separated into two categories: *arguments* and *proofs-of-knowledge*. We start by defining the former.

An Zero-knowledge argument is protocol run between an probabilistic polynomial time (PPT) prover P and and a PPT verifier V. The prover and verifier then both know a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, which expresses a computational problem. We refer to the first argument of the relation as h and to the second argument as w. The goal of the protocol is then for P to convince V that we knows the pair (h,w) whilst only revealing h. At the end of the protocol the verifier will then either output **accept/reject** based on whether P convinced him or not. We then require that the verifier following the protocol always output **accept** if we P knew (h,w) and followed the protocol. This is known as *correctness*. Moreover, we requires that that a cheating adversary who does not know w can only make the verifier output **accept** with some probability ε.

The stronger variant of *proofs-of-knowledge* shares the same definitions as above, but require that the verifier only output **accept** if the prover indeed knew the pair (h,w).

Common amongst both variants is that the require that verifier learns no information, whatsoever, about w. This is more formally defined as:

Definition 3.1.1 (Zero-knowledge from Damgaard [10])**.** Any proof-of-knowledge or argument with parties (P,V) is said to be zero-knowledge if there for every PPT verifier V* there exists a simulator Sim$_{V^*}$ running in expected polynomial time can output a conversation indistinguishable from a real conversation between (P, V*).

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.

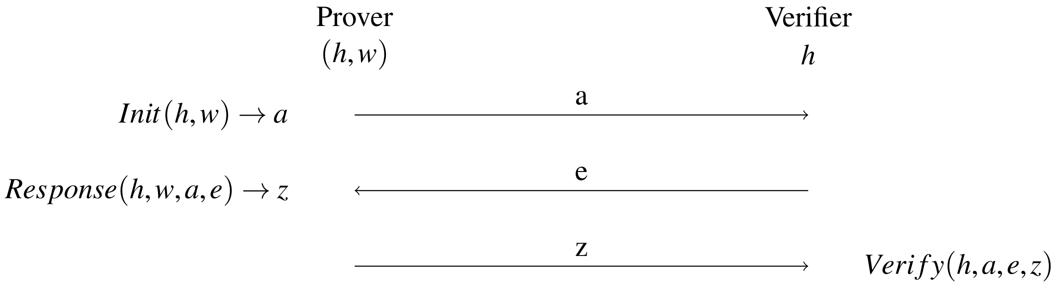

Figure 1: Σ-Protocol

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 [10].

►**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)**.

►**define honest**◄

Definition 3.2.2 (Completeness)**.** Assuming both P and V are honest then V will always output **accept** at the end of the protocol.

Definition 3.2.3 (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 algorithm, which we call the "witness_extractor", that given the two transcripts outputs a valid witness for the relation R.

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 3.2.4 (SHVZK)**.** A Σ-Protocol S is said to be SHVZK if there exists a polynomial time simulator Sim which given instance h and challenge e as input produce a transcript (a,e,z) indistinguishable from the transcript produced by S

Σ-Protocols has the interesting property of being able to construct zero-knowledge protocols from any secure Σ-Protocol in the random oracle model with no additional computations. This effectively allows us to construct a secure zero-knowledge protocol whilst only having the prove that the protocol is zero-knowledge in the case of a honest verifier. This transformation from Σ-Protocol to zero-knowledge protocol is known as the "Fiat-Shamir" transformation. More details about this transformation can be found in section 5.3. Moreover, it is possible to turn any Σ-Protocol into a zero-knowledge argument with one additional round of communication between the Prover and Verifier or a proof-of-knowledge with two extra rounds of communication without assuming access to a random oracle [10].

3.3 COMMITMENT SCHEMES

Commitment schemes is another fundamental building block in cryptography, and has a strong connection to Σ-Protocols where it is possible to construct commitment schemes from Σ-Protocols [8]. A commitment schemes facilitates an interaction between two parties, P1 and P2, where P1 generates a commitment of a message, which he then sends 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 (Gen, Com, Ver), where:

- $(ck, vk) \leftarrow Gen()$, 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**.

Definition 3.3.2 (Informal 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:

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

Definition 3.3.3 (Informal 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 different from the original 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 $(c, d') \leftarrow Com(ck, m')$.

The scheme is said to have *perfect binding* if it impossible to change the opening, *statistical binding* if there is a negligible probability of changing the opening and *computation binding* if producing a different opening is equivalent to a hard computation problem.

Definition 3.3.4 (Informal hiding)**.** The Hiding property states that a party given a commitment c, will not be able to guess the message m, which the commitment was based on.

The scheme is said to have *perfect hiding* if it is impossible to distinguish two commitment of different messages from each other, *statistical hiding* if there is a negligible probability of distinguishing the commitments and *computational hiding* if distinguishing the commitments is equivalent to a hard computational problem.

3.4 MULTI-PART COMPUTATION (MPC)

Consider the problem, where n parties, called $P_1,\ldots,p_n$, with corresponding input values $\mathbf{x} = x_1,\ldots,x_n$ where the parties are allowed to free communicate with each other over a secure channel. The parties then want to compute a public function, $f : (\text{input})^n \to \text{output}$, where each party contribute with their own input to the function and 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.

To achieve this the parties jointly run a MPC protocol Φ_f. This protocol is defined in the term of rounds. In each round each party P_i computes a value dictated by the protocol Φ_f and sends it to another party, or broadcasts it to all parties. This value is computed by a deterministic function of the private input x_i and all previously computed values by P_i along with all messages sent to P_i. We define the collection of computed values and received values as view_i.

Once all rounds of the protocol has been completed the output values y can be directly computed based on view_i.

Definition 3.4.1 (informal correctness)**.** A MPC protocol Φ_f computing a function f is said to have perfect correctness if $f(x) = \Phi_f(x)$ for all x.

Definition 3.4.2 (d-Privacy)**.** A MPC protocol Phi_f is said to have d-privacy if d parties colluding cannot about information about any of other $n-d$ private inputs.

More formally, the protocol has d-privacy if it is possible to define a simulator S_A producing views that are indistinguishable from the views of the d colluding parties.

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 specifically fix a type "randomness" which is responsible for 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:

Here verification of commitments and key pairs are modelled as function, since we assume these function to always be deterministic and lossless. There should be need to

```
op validate_key (sk : secret_key, pk : public_key) : bool.
op verify (pk : public_key) (m : message) (c : commitment) (d :
    randomness) : bool.

module type Committer = {
  proc * key_gen() : secret_key * public_key
  proc commit(sk : secret_key, m : message) : commitment *
    randomness
}.
```

Listing 2: Key-Based commitment specification

sample additional randomness to verify these. Mo rover, if the verification algorithms cannot terminate within a reasonable amount of time, then it is probably not wroth studying the commitment scheme further.

The committer is modelled as a module with two procedures. One for generating key pairs and one for committing to messages. This models the fact that a commit is able to hold state and make random choice, while the commitments he makes should be easily verifiable by anyone knowing the public key, without having to keep state about the committer.

By separating the verification functions from the committers procedures we get a formalisation closer to the real world, where verification functions should not be able to read any of the state of the committer. This could alternatively have been modelled with the verifier being a module, but allowing the verify to keep state complicates proofs, since verifier two messages could potentially have an effect on each other ►**rewrite this**◄. This is in contrast to previous work [13], which has proven problematic to work with, when applying the formalisation of commitment schemes in larger protocols (Section 7.3). ►**Rewrite in sec 7.3 that it is needed to swap the order of verifying commitments**◄

We define a commitment scheme C to be an implementation of the functions and procedures in listing 2.

►**Mention randomness distributions?**◄

4.2 KEY-LESS COMMITMENT SCHEMES

►**Is this section necessary?**◄ We furthermore formalise a variant of commitment schemes that we key-less. This is formalised independently from the key-based commitment schemes, since the change is function signatures makes it incompatible with the key-based formalisation, they would potentially be merged into one formalisation, which allows for both to be used whenever a commitment scheme is required. The main reason for not doing this is that proofs of protocols depending on commitment schemes can become easier when it is not necessary to quantify over keys (See section 7.3 for an example of this). Ideally it should be proven that the two formalisation are compatible wrt. security, and one can be used in place of the other, but this has been beyond the scope of this thesis.

The functions and procedures used by the key-less commitment schemes are identical to the ones listed in Figure 2 for the key-based commitment schemes with the only difference being all references to the public and secret keys has been removed. Furthermore, the Committer module now only contains one procedure commit, since there is no longer a need to generate key pairs.

Moreover, the security definitions remain the same but, again, with the key generation removed along with the references to the secret and public keys.

4.3 SECURITY

For both the key-based and key-less variant of the give the same definitions of security, which is based on the work of Metere and Dong [13].

Definition 4.3.1 (Correctness)**.** A commitment scheme C is correct if:

$$\forall m. \Pr[Correctness(C).main(m) = true] = 1.$$

where Correctness(C) is defined as:

```
module Correctness(C : Committer) = {
  main(m : message) = {
    (sk, pk) = C.key_gen(); (* Omitted in the key-less case *)
    (c, d) = C.commit(sk, m);
    valid = verify pk m c d;
    return valid;
  }
}.
```

Definition 4.3.2 (Hiding)**.** A Commitment scheme C can have to following degrees of hiding *perfect hiding:* $\forall Adv.\Pr[HidingGame(C,Adv).main() = true] = \frac{1}{2}$ *computation hiding:* $\forall Adv.\Pr[HidingGame(C,Adv).main() = true] = \frac{1}{2} + \varepsilon$

Where we define the adversary Adv and HidingGame as follows:

```
module type HidingAdv = {
  proc * get() : message * message
  proc check(c : commitment) : bool
}.

module HidingGame(C : Committer, A : HidingAdv) = {
  proc main() = {
    (sk, pk) = C.key_gen();
    (m, m') = A.get();
    b <$ {0,1};
    if (b) {
      (c, r) = C.commit(sk, m);
    } else {
      (c, r) = C.commit(sk, m');
    }
    b' = A.check(c);
    return b = b';
  }
}.
```

Definition 4.3.3 (Binding)**.** A commitment scheme C can have to following degrees of binding: *perfect binding:* $\forall Adv.\Pr[BindingGame(C,Adv).main() = true] = 0$ *computational binding:* $\forall Adv.\Pr[BindingGame(C,Adv).main() = true] = \varepsilon$

Where we define the adversary Adv and BindingGame as:

```
module type BindingAdv = {
  proc bind(sk : secret_key, pk : public_key) : commitment *
   message * message * randomness * randomness
}.

module BindingGame(C : Committer, B : BindingAdv) = {
  proc main() = {
    (sk, pk) = C.key_gen();
    (c, m, m', r, r') = B.bind(sk, pk);
    v =  verify pk m c r;
    v' = verify pk m' c r';
    return (v /\ v') /\ (m <> m');
  }
}.
```

In our definitions of hiding and binding we do not have a formalisation of the statistical variant, since it is still unclear how to express those in EC ►reference◄

4.4 ALTERNATIVE DEFINITIONS OF SECURITY

Based on the previously defining notions of security we also introduce a number of alternative definitions, some which can be directly derivable from our original definitions, whilst the others does not offer an easy reduction but intuitively capture the same aspects of security.

Lemma 4.4.1 (Alternative correctness)**.** A commitment scheme C is correct if:

$$\begin{aligned}&\forall m, sk, pk.\\&\qquad \text{validate\_key } sk\ pk \wedge \Pr[\mathrm{key\_fixed}(m, sk, pk) = true] = 1\\&\qquad \implies \Pr[\mathrm{Completeness(C).main(m)}] = 1.\end{aligned}$$

Where key_fixed is given by the following procedure:

```
proc key_fixed(m : message, sk : secret_key, pk : public_key) =
    {
  (c, d)    = C.commit(sk, m);
  b         = verify pk m c d;
  return b;
}
```

Proof. We start by introducing an intermediate game:

```
proc intermediate(m : message) = {
  (sk, pk) = C.key_gen();
  b = key_fixed(m, sk, pk);
  return b;
}
```

We then prove that intermediate is equivalent to Completeness(C).main by inlining all procedures and observing that both procedures are equal in structure.

We are then left with showing:

$$\begin{aligned}&\forall m, sk, pk.\\&\qquad \text{validate\_key } sk\ pk \wedge \Pr[\mathrm{key\_fixed}(m, sk, pk) = true] = 1\\&\qquad \implies \Pr[\mathrm{intermediate}(m)] = 1.\end{aligned}$$

We then use the assumption that key_fixed is correct to prove that it returns true when called as a sub-procedure in intermediate. Last we have to prove that (sk, pk) are a valid key pair, but since they are generated by C.key_gen they must be valid. □

Definition 4.4.2 (Perfect Hiding)**.** A commitment scheme C offers perfect hiding, if the output distribution of two committers with the same state but different messages are perfectly indistinguishable.

$$equiv[commit \sim commit : =\{sk, m, \mathbf{glob}\ Committer\} \implies =\{res, \mathbf{glob}\ Committer\}]$$

Definition 4.4.3 (Alternative Binding)**.** A commitment scheme C offers binding with probability p if: $\Pr[alt\_binding(c,m,m') = true] = p$

for procedure binding given by:

```
proc alt_binding(c : commitment, m m' : message) = {
  v1 = verify m c;
  v2 = verify m' c;
  return v1 /\ v2 /\ (m ≠ m');
}
```

The commitment schemes offers *perfect binding* if $p = 0$

The alternative definition of hiding only works in the perfect case, but it is much easier to work with within EasyCrypt when this is the case. This is due to most proofs being stated is indistinguishability proofs, which are bothersome to convert to adversarial proofs. **►rewrite this◄**

The alternative definition of binding allows us to use the ambient logic to reason about the probability of breaking the binding property instead of the Hoare logics by the way of an adversary. The benefit of reasoning about statement in the ambient logic is that they are usually easier to reason about while offering better modularity since we can use ambient logic to reason about probabilities of different procedures. Additionally, computational binding can be shown by proving equality between two procedures rather than constructing an adversary.

4.5 CONCRETE INSTANTIATION: PEDERSEN COMMITMENT

To show the workability of the proposed formalisation we show that it can be used replicate the results of Metere and Dong [13]. Pedersens commitment scheme is based on the discrete logarithm assumption

The Pedersen commitment scheme is a protocol run between a committer C and a receiver R. Both parties have before running the protocol agreed on a group $(\mathcal{G}, q, g)$, where q is the order of $\mathbb{G}$ and g is the generator for the group.

When the committer want to commit the a message m he does the following:

- He lets R sample a key $h \in_{\mathrm{R}} \mathbb{G}$ and send it to him
- Sample a random opening $d \in_{\mathrm{R}} \mathbb{Z}_q$ and sends the key and commitment $c = g^d h^m$ to R.

At a later time, when C is ready to show the value he committed to, he sends the message and random opening, (m', d') to R, when then runs the following verification steps:

- R computes $c' = g^{d'} h^{m'}$ and checks that $c = c'$.

From this description it is clear that the verification step is simply a function taking as input the key, commitment, message and opening and does a deterministic computations. This fits perfectly within our formalisation of the Receiver, we therefore instantiate our commitment scheme framework with the following:

```
clone export Commitment as Com with
  type public_key <- group (* group element *)
  type secret_key <- group
  type commitment <- group
  type message    <- F.t   (* Finite field element, like ℤq *)
  type randomness <- F.t

  op dm = FDistr.dt, (* Distribution of messages *)
  op dr = FDistr.dt, (* Distribution of randomness *)
  op verify pk (m : message) c (r : randomness) = g^r * pk^m = c
    ,
  op valid_key (sk : secret_key) (pk : public_key) = (sk = pk).

module Pedersen : Committer = {
  proc key_gen() : secret_key * public_key = {
    a <$ dr;
    $h = g^a$;

    return (h, h);
  }

  proc commit(sk : secret_key, m : message) = {
    r <$ dr;
    $c = g^r \cdot (sk^m)$;

    return (c, r);
  }
}.
```

Listing 3: Pedersen instantiation

Here our formalisation assumes that the Committer samples the keys but as we will see in the following section we are still able to prove security of the scheme regardless of who generates the keys. Here we use the Cyclic Group theory from EC to generate the agreed upon group and model uniform distributions of messages and randomness by...

SECURITY To prove security of the protocol we should that the previous definitions of correctness, hiding and binding can be proven true.

Lemma 4.5.1 (Pedersen correctness)**.** $\forall m. \Pr[\text{Correctness}(\text{Pedersen}).\text{main}(m) = \mathit{true}] = 1$

Proof. correctness follows directly by running the procedure and observing the output. □

Lemma 4.5.2 (Pedersen hiding)**.** We show that Pedersen has perfect hiding by definition 4.3.2.

Proof. To prove hiding we start by introducing an intermediate hiding game where we commit to a random message instead of one of the messages chosen by the adversary:

```
module HidingIdeal(A : HidingAdv) = {
  proc main() = {
    (sk, pk) = Pedersen.key_gen();
    (m, m') = A.get();
    b $<DBool.dbool; r <\$ dr;
    c = $g^r$;
    b' = A.check(c);
    return b = b';
  }
}.
```

We then split the proof into two parts:
1) $\forall Adv.\Pr[\mathrm{HidingGame}(\mathrm{Pedersen},\ \mathrm{Adv}).main = true] = \Pr[\mathrm{HidingIdeal}(\mathrm{Adv}).main = true]$ Where we prove that for any choice of b the two procedures and indistinguishable. We start by prove indistinguishability with $b = 0$. To prove this we have to prove that $g^r \sim g^{r'} \cdot \mathrm{sk}^m$ Here we can use EasyCrypt's coupling functionality to prove that $r \sim r' \cdot \mathrm{sk}^m$ since both r, r' and sk^m are all group elements and the distribution of r is full and uniform.

The proof of $b = 1$ is equivalent.
2) $\forall Adv.\Pr[\mathrm{HidingIdeal}(\mathrm{Adv}).main = true] = \frac{1}{2}$

Since $c = g^r$ is completely random the adversary has no better strategy than to guess at random.

By the facts **1)** and **2)** we can conclude that Pedersen commitment scheme has perfect hiding. □

Lemma 4.5.3 (Pedersen Binding)**.** We show computation binding under definition 4.3.3

Proof. We prove computation binding of Pedersen commitment by showing that an adversary breaking binding can be used to construct a adversary solving the discrete logarithm.

```
module DLogPedersen(B : BindingAdv) : Adversary = {
  proc guess(h : group) = {
    (c, m, m', r, r') = B.bind(h, h);
    v = verify h m c r;
    v' = verify h m' c r';
    if ((v /\ v') /\ (m <> m')) {
      w = Some( (r - r') * inv(m' - m) );
    } else {
      w = None;
    }
    return w;
  }
}.
```

We then prove:

$$\begin{aligned}
&\forall Adv. \\
&\qquad \Pr[\mathrm{BindingGame}(\mathrm{Pedersen},\ \mathrm{Adv}).main() = true] \\
&\quad = \Pr[\mathrm{DLogGame}(\mathrm{Pedersen},\ \mathrm{Adv}).main() = true].
\end{aligned}$$

Fist we show that if DLogPedersen if given one commitment with two openings then the discrete logarithm can be solved. This is given by:

$$m \neq m' \tag{2}$$

$$\implies c = g^r \cdot g^{a^m} \wedge c = g^{r'} \cdot g^{a^{m'}} \tag{3}$$

$$\implies a = (r - r') \cdot (m' - m)^{-1} \tag{4}$$

Which is easily proven by EasyCrypt's automation tools.

Next we show that the two procedures are equivalent. Which follows by inlining all procedures and observing the output. Procedure DLogPedersen.main can only output true if equations 2 and 3 holds, which is what procedure BindingGame(Pedersen, Adv).main needs to satisfy to output true. We can therefore conclude that two procedures imply each other. □

5

FORMALISING Σ-PROTOCOLS

►Aim to port results from Isabelle to EC◄ 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.

►Cite other works about Σ-Protocols◄

5.1 DEFINING Σ-PROTOCOLS

We start by defining the types for any arbitrary Σ-Protocol:

type: statement
witness
message
challenge
response

These types corresponds to the types from Figure 1.

Furthermore, we define the relation for which the protocol operates on as a binary function mapping a statement and a witness to true/false : $R : (\text{statement} \times \text{witness}) \to \{0,1\}$ along with a distribution over challenges. This distribution is used to model a honest verifier which will always generate a random challenge. Since distribution are probabilistic programs within EasyCrypt we require that sampling from the distribution is always successful. This is referred to as the distribution being lossless.

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

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

Listing 4: Abstract procedures of Σ-Protocols

```
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: Completeness game for Σ-Protocols

Here all procedures are modelled into the same module. This allows the Verifier procedure to access the global state of the Prover. This could lead to invalid proofs of security. It is therefore important to not implement a verify procedure which access global state of the SProtocol module. This could have been alleviated by splitting the SProtocol module into multiple different modules with only the appropriate procedures inside. This would remove any potential for human error when defining a Σ-Protocol, but it makes it more bothersome to instantiate a Σ-Protocol in EasyCrypt . Ultimately, we decided on having everything defined within the same module.

►Here gen is . . . ◄

An instantiation of a Σ-Protocol is then an implementation of the procedures in Listing 5.

We then model security as a series of games:

Definition 5.1.1 (Completeness)**.** We say that a Σ-protocol, S, is complete, if the probabilistic procedure in 5 outputs 1 with probability 1, i. e.

$$\forall h\ w, \mathrm{R}\ h\ w \implies \Pr[\text{Completeness(S).main}(h,w) = \mathit{true}] = 1. \tag{5}$$

One problem with definition 5.1.1 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 5.1.2 (Alternative Completeness)**.** We say that a Σ-protocol, S, is complete if:

$$\forall h\ w\ e, \mathrm{R}\ h\ w \implies \Pr[\text{Completeness(S).special}(h,w,e) = \mathit{true}] = 1. \tag{6}$$

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 5.1.3. Given that definition 5.1.2 then it must hold that definition 5.1.1 holds, given the same public input and witness.

$$\Pr[\mathrm{special} : \mathit{true} \implies \mathit{res}] = 1 \implies \quad \Pr[\mathrm{Completeness(S).main} : \mathit{true} \implies \mathit{res}] = 1.$$

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:

$$\Pr[\mathrm{special} : \mathit{true} \implies \mathit{res}] = 1 \implies \qquad \Pr[\mathrm{intermediate} : \mathit{true} \implies \mathit{res}] = 1.$$

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

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

Which then is proven by the hypothesis of the “special” procedure being complete. □

Definition 5.1.4 (Special Soundness)**.** A Σ-Protocol S has special soundness if:

$$\begin{aligned}
&\forall h, w, a, e, e', z, z'.\\
&\qquad e \neq e'\\
&\qquad \mathrm{R}\; h\; w \implies\\
&\qquad \wedge \Pr[S.\mathit{verify}((h_1,h_2),(w_1,w_2),a,e,z)] = 1\\
&\qquad \wedge \Pr[S.\mathit{verify}((h_1,h_2),(w_1,w_2),a,e',z')] = 1\\
&\qquad\quad \implies \Pr[\mathit{SpecialSoundness}(\mathrm{ANDProtocol}(P_1,P_2)).\mathit{main}(h,a,[e;e'],[z;z'])] = 1
\end{aligned}$$

With $\mathrm{SpecialSoundness}$ defined as:

Definition 5.1.5 (Special Honest Verifier Zero-Knowledge)**.** To define SHVZK we start by defining a module SHVZK containing two procedures: We then say a Σ-Protocol S is special honest verifier zero-knowledge if:

$$\mathit{equiv}[\mathrm{SHVZK}.\mathit{real} \sim \mathrm{SHVZK}.\mathit{ideal} : = \{h,e\} \wedge \mathrm{R\ h\ w}^{\mathit{real}} \implies = \{\mathit{res}\}]$$

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

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

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

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

Listing 6: 2-special soundness game

```
proc real(h, w, e) = {
   a = init(h,w);
   z = respose(h,w,e,a);
   return (a, e, z);
}
```

```
proc ideal(h, e) = {
  (a, z) = simulator(h, e);
  return (a, e, z);
}
```

Figure 2: SHVZK module

Definition 5.1.6. S is said to be a Σ-Protocol if it implements the procedures in figure 4 and satisfy the definitions of completeness, special soundness, and special honest verifier zero-knowledge.

►Argue that games corresponds to original definitions◄

►SHVZK only captures perfect indis. Unclear how to do with equiv?◄

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

5.2 COMPOUND 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...

Formalisations of compound Σ-Protocols already exists for other proof assistants [6, 8], which we will also use as a basis for our EasyCrypt formalisation. By drawing on previous work we aim to make a formalisation that is workable and succinct within reason of what EasyCrypt allows us to do. Moreover, by recreating formalisations within new proof assistant we can gain valuable insight into how EasyCrypt compares to other proof assistant whilst reflecting on how to improve previous work.

HIGHER ORDER INSTANCES OF THEORIES **►Unsure how?◄**

5.2.1 *AND*

►Based on description from [10]◄ ►Not entirely correct. Need $h \in domainR$ to discharge axioms◄ Given two Σ-Protocols, S_1 with relation $R_1(h_1, w_1)$ and S_2 with relation $R_2(h_2, w_2)$ we define the AND construction to be a Σ-Protocol proving knowledge of the relation $R((h_1, h_2), (w_1, w_2)) = R_1(h_1, w_1) \wedge R_2(h_2, w_2)$.

The construction of AND protocol is then a Σ-Protocol running both S_1 and S_2 as sub-procedures. To formalise this we start by declaring the AND construction as an instantiation of a Σ-Protocol. To do this we first need to define the types for which the protocol works of. But before we can define the types of the AND construction we need to know the types of the underlying Σ-Protocols S_1 and S_2. To denote the types of S_i we use the notation: type_i

$$\begin{aligned}\textbf{Type: } &\text{statement} = \text{statement}_1 \times \text{statement}_2\\ &\text{witness} = \text{witness}_1 \times \text{witness}_2\\ &\text{message} = \text{message}_1 \times \text{message}_2\\ &\text{challenge} = \text{challenge}_1 = \text{challenge}_2\\ &\text{response} = \text{response}_1 \times \text{response}_2\end{aligned}$$

We then define the AND construction as a module parametrised by Σ-Protocols satisfying the type signatures of S_1 and S_2, which can be seen in Listing 7. This might seem restrictive, since the AND construction can now only be made from Σ-Protocol with the specific type signature of S_1 and S_2, but recall that the entire AND construction is quantified over the types given in the type declaration. This means that the types of S_1 and S_2 can be fixed to any arbitrary types and therefore can express any Σ-Protocol. But, if S_1 and S_2 are any arbitrary Σ-Protocols, then why are the AND construction parametrised by Σ-Protocols satisfying the type signatures of S_1 and S_2 rather than just parametrising the AND construction be any two Σ-protocols? Ideally, this would how the AND construction is formalised, but due to how EasyCrypt handles types we need to declare the types of the AND construction and ensure that the procedures are typeable. The only way of ensuring this is by fixing the types of the underlying Σ-Protocols before instantiation the AND construction as a Σ-Protocol.

►Explicitly mention axioms◄

SECURITY Given the AND constructions instantiation of a Σ-Protocols we simply need to prove the security definitions given in section 5.1 with regards to the module ANDProtocol

Lemma 5.2.1 (AND Completeness)**.** Assume Σ-Protocols P_1 and P_2 are complete then Module $\text{ANDProtocol}(P_1, P_2)$ satisfy completeness definition 5.1.1

Proof. By inlining the procedures of $\text{ANDProtocol}(P_1, P_2)$ in $\text{Completeness}(\text{ANDProtocol}).\text{special}$ we see that it is equivalent to: $\text{Completeness}(P_1).\text{special};\ \text{Completeness}(P_2).\text{special}$. Which is true by our assumption of P_1 and P_2 being complete. We need to use the special definition of the completeness game here, since the challenge e is given by a Verifier running the AND construction. And the sub-protocols are, therefore, not allowed to sample their own challenges and need to use the challenge from the AND construction.

Then by lemma 5.1.3 we get that $\Pr[\text{Completeness}(\text{AND}(P_1, P_2).\mathit{main}] = 1$ □

►Write protocol as diagram?◄

```
module ANDProtocol (P1 : S1, P2 : S2) = {
  proc init(h : statement, w : witness) = {
    (h1, h2) = h;
    (w1, w2) = w;

    a1 = P1.init(h1, w1);
    a2 = P2.init(h2, w2);
    return (a1, a2);
  }

  proc response(h : statement, w : witness, m : message, e :
   challenge) : response = {
    (m1, m2) = m;
    (h1, h2) = h;
    (w1, w2) = w;

    z1 = P1.response(h1, w1, m1, e);
    z2 = P2.response(h2, w2, m2, e);
    return (z1, z2);
  }

  proc verify(h : statement, m : message, e : challenge, z :
   response) : bool = {
    (h1, h2) = h;
    (m1, m2) = m;
    (z1, z2) = z;

    v = P1.verify(h1, m1, e, z1);
    v' = P2.verify(h2, m2, e, z2);

    return (v /\ v');

  }
```

Listing 7: AND construction

Lemma 5.2.2 (AND special soundness)**.** Given secure Σ-Protocols P1 and P2 the AND construction AND(P1, P2) satisfy definition 5.1.4

Proof. Since a transcript of AND(P1, P2) is the transcripts of running P1 and P2 combined simulating a transcript for AND(P1, P2) is equivalent to simulating transcripts for P1 and P2 and combining them. By SHVZK of P1 and P2 this will always succeed. □

Lemma 5.2.3 (AND SHVZK)**.** Given secure Σ-Protocols P_1 and P_2 then AND(P1, P2) satisfy definition 5.1.5.

Proof. We start by showing:

$$verify((h1,h2),(a1,a2),s,(e1,z1,e2,z2) \iff \tag{7}$$

$$P1.verify(h1,a1,e1,z1) \wedge P2.verify(h2,a2,e2,z2) \tag{8}$$

Since the relation $R\ (h1,h2)\ (w1,w2) = R1\ h1\ w1/landR2\ h2\ w2$ we need to produce valid witnesses for the protocol P1 and P2. Since both protocols have special soundness we can use equation 7 to apply the special soundness property of both P1 and P2, which completes the proof. □

5.2.2 *OR*

Here we use the definition of the OR construction by [10], which states that both sub-protocols must have the same witness type.

Given two Σ-Protocols, S_1 with relation $R_1(h_1,w)$ and S_2 with relation $R_2(h_2,w)$ we define the AND construction to be a Σ-Protocol proving knowledge of the relation $R((h_1,h_2),w) = R_1(h_1,w) \vee R_2(h_2,w)$.

The main idea behind the OR construction, is that by the SHVZK it is possible to construct accepting conversations for both S_1 and S_2 if the Prover is allowed to choose what challenge he responds to. Obviously, if the Prover is allowed to chose the challenge the protocol is would not be secure. Therefore, we limit the Prover such that he can choose the challenge for one sub-protocol, but must run the other sub-protocol with a challenge influenced by the Verifier. This is done by letting the Prover chose two challenges e_1 and e_2, which the Verifier will only accept, if the $e_1 \oplus e_2 = s$ where s is the challenge produced by the Verifier. By producing accepting transcripts for both sub-protocols it must be true that he knew the witness for at least one of the relations.

To formalise this we first need a way to express that the challenge type supports XOR operations. To do this we add the following axioms, which will have to be proven true before our formalisation can be applied.

$$\textbf{op}\ (\oplus)\ c_1\ c_2 : \text{challenge} \tag{9}$$

$$\textbf{axiom xorK}\ x\ c : (x \oplus c) \oplus c = x \tag{10}$$

$$\textbf{axiom xorA}\ x\ y : (x \oplus y) = y \oplus x \tag{11}$$

►**The protocol then proceeds as ...**◄

We then define the OR construction as a Σ-Protocol like in section 5.2.1. The procedures can be seen in listing 8.

►**Write protocol as diagram?**◄

```
proc init(h : statement, w : witness) = {
  (h1, h2) = h;

  if (R1 h1 w) {
    a1 = S1.init(h1, w);
    e2 <$ dchallenge;
    (a2, z2) = S2.simulator(h2, e2);
  } else {
    a2 = S2.init(h2, w);
    e1 <$ dchallenge;
    (a1, z1) = S1.simulator(h1, e1);
  }
  return (a1, a2);
}

proc response(h : statement, w : witness, m : message, s :
    challenge) = {
  (m1, m2) = m;
  (h1, h2) = h;

  if (R1 h1 w) {
    e1 = s ⊕ e2;
    z1 = S1.response(h1, w, m1, e1);
  } else {
    e2 = s ⊕ e1;
    z2 = S2.response(h2, w, m2, e2);
  }
  return (e1, z1, e2, z2);
}

proc verify(h : statement, m : message, s : challenge, z :
    response) = {
  (h1, h2) = h;
  (m1, m2) = m;
  (e1, z1, e2, z2) = z;

  v = S1.verify(h1, m1, e1, z1);
  v' = S2.verify(h2, m2, e2, z2);

  return ((s = e1 ⊕ e2) /\ v /\ v');
}
```

Listing 8: OR construction

SECURITY Given the OR constructions instantiation of a Σ-Protocols we simply need to prove the security definitions given in section 5.1 with regards to the module ORProtocol

Lemma 5.2.4 (OR Completeness)**.** Assume Σ-Protocols P_1 and P_2 are complete and shvzk then $\mathrm{ORProtocol}(P_1, P_2)$ satisfy completeness definition 5.1.1

Proof. To prove completeness we branch depending on which relation holds. If R1 $h1$ w holds then all P1 procedures can be grouped together as the P1 completeness game. We then need to prove that S2.verify output accept on the transcript generated by S2.simulator which is true by the assumption of SHVZK of P2. The proof when R2 $h2$ w holds follows similarly. □

Lemma 5.2.5 (OR SHVZK)**.** Given Σ-Protocols P_1 and P_2 that satisfy SHVZK then:

$$equiv[SHVZK(OR(P1,P2)).ideal \sim SHVZK(OR(P1,P2)).real]$$

With the Pre and Post condition given by definition 5.1.5.

Where the simulator for the OR construction is given by

```
proc simulator(h : statement, s : challenge) : message *
    response = {
  (h1, h2) = h;
  e2 <$ dchallenge;
  e1 = s ^^ c2;

  (a1, z1) = P1.simulator(h1, e1);
  (a2, z2) = P2.simulator(h2, e2);

  return ((a1, a2), (e1, z1, e2, z2));
}
```

Proof. We again split the proof based on which relation holds.
case (R1 h1 w): for this case we have to show the following.
1) that $e1$ and $e2$ are indistinguishable. This follows trivially since we assume both procedures make the same random choices and since the order in which the challenges are sampled they must be equal.
2) that the transcript $(a1, e1, z1)$ made by running P1 on input (h1,w) is indistinguishable from the transcript produced by P1.simulator(h, e1). The rest of the procedures is trivially equivalent since they call the same procedures with the same arguments. This follows from the SHVZK property of P1.

Both of these facts allow us that the procedures are indistinguishable in this case, since if the challenges are indistinguishable then the sub-procedures in both procedures are effectively called on the same inputs.
case (R2 h2 w): This proof follows the same steps as the other case with the only exception being step **1)**. In this step, since the challenges are sampled in a different order, we cannot assume them to be equal since they are sampled with different randomness. Instead we use EasyCrypt's coupling functionality to prove that $e_1^{ideal} \sim e_1^{real} \oplus s$ and $e_1^{real} \sim e_1^{ideal} \oplus s$ The indistinguishability follows trivially since the challenge distribution is assumed full and uniform.

From this we are left with showing:

$$\begin{aligned} e_1^{real} &= s \oplus e_2^{real} && \text{eq. 10 and 11} \\ &\sim s \oplus e_1^{ideal} \oplus s && \text{Coupling} \\ &= e_1^{ideal} && \text{eq. 10 and 11} \end{aligned}$$

Which completes the proof. □

Lemma 5.2.6 (OR special soundness)**.** Given secure Σ-Protocols P1 P2 then The OR construction OR(P1,P2) satisfy definition 5.1.4 with the witness extractor for the OR construction defined as:

```
proc witness_extractor(h, a, s : challenge list , z : response
    list) = {
  (h1, h2) = h;
  (a1, a2) = m;
  (e1, z1, e2, z2) = z[0];
  (e1', z1', e2', z2') = z[1];
  if (e1 ≠ e1') {
    w = P1.witness_extractor(h1, a1, [e1;e1'], [z1;z1']);
  } else {
    w = P2.witness_extractor(h2, a2, [e2;e2'], [z2;z2']);
  }
  return w;
}
```

Proof. We split the proof into two parts:

- $(e1 \neq e1')$: Here we must prove that P1.witness_extractor produce a valid witness for R.

  Here we use equation 7 from the special soundness proof of AND which lets us apply the special soundness property of P1, which gives us that $R1\ h1\ w \implies R1\ h1\ w \vee R2\ h2\ w = R\ (h1,h2)\ w$

- $\neg(e1 \neq e1')$ Here we prove the same, but with the special soundness property of P2 instead.

□

5.3 FIAT-SHAMIR TRANSFORMATION

The Fiat-Shamir transformation is a technique for converting Σ-protocols into zero-knowledge protocols. Σ-Protocols almost satisfy the definition of zero-knowledge, the only problem is that Σ-Protocols only guarantee zero-knowledge given the verifier is honest. This is stated by the Special Honest Verifier Zero-Knowledge property. However, if we can alter the protocol to force the verifier to always be honest, then the protocol, by definition, must be zero-knowledge. The Fiat-Shamir transformation achieves this by removing the verifier from the protocol and thus making it non-interactive. The verifier is then replaced by an random oracle, which generates a random challenge based on the first message of the prover, thus it works exactly like an honest verifier in the interactive protocol. However, since the random oracle is a sub-procedure of the prover he is allowed to make polynomially many call to the oracle in the hopes of getting a good challenge.

5.3.1 *Oracles*

To formalise this transformation we first need a clear description of what a random oracle is.

To capture the functionality of a random oracle we define the following abstract module:

```
module type Oracle = {
  proc * init() : unit
  proc sample (m : message) : challenge
}.
```

In essence, an oracle should be able to initialise its state, which used to determine the random choices made by the oracle. Moreover, it exposes the procedure sample which maps messages to challenges.

In the case of a random oracle we require that oracle responds with the same challenge if sample is queried with the same message multiple times. This is implemented by the following module:

```
module RealOracle : Oracle = {
  global variable : h = (message ↦ challenge)

  proc init() = {
    h = empty map;
  }

  proc sample (m : message) : challenge = {
    if (m ∉ Domain(h)) {
      h[m] <$ dchallenge; (* Sample random value in entry m *)
      }
    return h[m];
    }
  }.
```

5.3.2 *Non-interactive* Σ*-Protocol*

We can define the non-interactive version of the protocol as the following procedure:

```
module FiatShamir(S : SProtocol, O : Oracle) = {
  proc main(h : statement, w : witness) : transcript = {
    O.init();
    a = S.init(h, w);
    e = O.sample(a);
    z = S.response(h, w, a, e);

    return (a, e, z);
  }
}.
```

Here, a non-interactive version of a Σ-Protocol is a procedure producing a transcript by first initialising the oracle and then sampling a challenge from it.

SECURITY To prove security of the Fiat-Shamir transformation we need to use the security definition of a zero-knowledge protocol.

Lemma 5.3.1. If the underlying Σ-Protocol S is secure and the random Oracle O is lossless then the Fiat-Shamir transformation is correct.

Proof. By comparing the completeness from the underling Σ-protocol to the transformation we see that the only different is that underlying protocol waits for the verifier to sample a challenge for him. Since a honest verifier will never fail to send the challenge (i. e.he is lossless) and it will always be uniformly chosen the two procedures are equivalent. □

Lemma 5.3.2. If the underlying Σ-Protocol S is secure and the random Oracle O is lossless then the Fiat-Shamir transformation is zero-knowledge

Proof. To prove zero-knowledge in the random oracle model we must define a simulator producing indistinguishable output from the real procedure. Moreover, the simulator is allowed to choose the choices made by the oracle for the real protocol.

From the correctness proof we know that the random oracle acts as a honest verifier. Therefore the SHVZK simulator for S proves zero-knowledge for the transformation. □

Soundness, however, cannot be proven by the definition of special soundness from Σ-Protocols, since the Prover has gained more possibilities of cheating the verifier. We could prove some arbitrary bounds, but to get a meaningful proof of soundness for the Fiat-Shamir transformation we would need the rewinding lemma, which still an open research topic to formalise within EasyCrypt [7].

5.4 CONCRETE INSTANTIATION: SCHNORR PROTOCOL

To show the workability of the proposed formalisation we show that it can be used to instantiate Schnorr's protocol. The Schnorr's protocol is run between a Prover C and a Verifier R. Both parties have before running the protocol agreed on a group $(\mathbb{G}, q, g)$, where q is the order of $\mathbb{G}$ and g is the generator for the group. Schnorr's protocol is a Σ-Protocol for proving knowledge of a discrete logarithm. Formally it is a Σ-Protocol for the relation R h w $= (h = g^w)$

When the P wants to prove knowledge of the w to V he starts by constructing a message $a = g^r$ for some random value r. The Verifier will the generate a random challenge, e, which is a bit-string of some arbitrary length that defines the security of the protocol. Based on this challenge P then constructs a response $z = r + e \cdot w$ and sends it to V. To verify the transcript (a, e, z) V then checks if $g^z = a \cdot h^e$.

From this general description it is clear that this protocol fits within out formalisation of Σ-Protocol procedures. We then define the appropriate types and instantiate the protocol using out Σ-Protocol formalisation:

```
clone export SigmaProtocols as Sigma with
  type statement <- group, (* group element *)
  type witness   <- F.t,   (* Finite field element, like $\mathbb{Z}_q$ *)
  type message   <- group,
  type challenge <- F.t,
  type response  <- F.t,

  op R h w =  $(h = g^w)$
  op dchallenge = FDistr.dt (* Distribution of messages *)
  proof *.
  realize dchallenge_llfuni. by split; [apply FDistr.dt_ll |
    apply FDistr.dt_funi].
```

```
module Schnorr : SProtocol = {
  var r : F.t
  proc init(h : statement, w : witness) : message = {
    r <$ FDistr.dt;
    return g^r;
  }

  proc response(h : statement, w : witness, a : message, e :
   challenge) : response = {
    return r+e·w;
  }

  proc verify(h : statement, a : message, e : challenge, z :
   response) : bool = {
    return (g^z = a·(h^e));
  }
}
```

Listing 9: Schnorr instantiation

Here we first discharge the assumption that the challenge are lossless, uniform and fully distributed by using the EasyCrypt theories about distributions and cyclic groups.

To prove security of the protocol we show that the it satisfies the security definitions from section 5.1.

Lemma 5.4.1 (Schnorr correctness)**.** R h w $\implies \Pr[\text{Completeness}(\text{Schnorr}).main(h,w)] = 1$

Proof. To prove correctness we need to prove two things:

1. That the procedure always terminates
2. That it always outputs true

1) Since all procedures bar the random sampling in Schnorr are arithmetic operations they can never fail. The random sampling have been proven to be lossless. Therefore the procedures always terminates.
2) After running all sub-procedures of the correctness game the output of the procedure is

$$\begin{aligned} & g^{r+e\cdot w} = g^r \cdot h^e \\ \iff & g^{r+e\cdot w} = g^r \cdot g^{w^e} && \text{R h w} = (h = g^w) \\ \iff & g^r \cdot g^{e\cdot w} = g^r \cdot g^{w\cdot e} \end{aligned}$$

Which is easily proven by EasyCrypt automation tools for algebraic operations. □

Lemma 5.4.2 (Schnorr soundness)**.**

$$\begin{aligned} & e \neq e' \implies \\ & \Pr[\text{verify}(a,e,z)] = 1 \implies \\ & \Pr[\text{verify}(a,e',z')] = 1 \implies \\ & \Pr[\text{Soundness}(\text{Schnorr})(a,[e;e'],[z;z'])] = 1 \end{aligned}$$

Proof. We start by defining the witness extractor for Schnorr's protocol:

```
proc witness_extractor(h : statement, m : message, e : challenge
     list, z : response list) : witness= {
  return (z[0] - z[1]) / (e[0] - e[1]);
}
```

►Define list indexing in background chapter◄ To prove that the soundness game succeeds we need the following

1. Both transcripts are accepting

2. The witness extractor produces a valid witness for the relation R

1) By stepping though the while loop of the soundness game we can show that all transcripts must be accepting by our assumptions.
2) Running all procedures of the soundness game we are left with showing:

$$\mathrm{R}\ \mathrm{h}\ ((z-z')/(e-e'))$$

Which follows by unfolding the definition of z and z' and using the automation tools of EasyCrypt to solve algebraic operations. □

Lemma 5.4.3 (Schnorr SHVZK)**.**

$$equiv[\mathrm{SHVZK(Schnorr)}.ideal \sim \Pr[\mathrm{SHVZK(Schnorr)}.real] := \{h,e\} \wedge \mathrm{R\ h\ w}^{real} \Longrightarrow = \{res\}]$$

Proof. We start by defining the simulator for Schnorr's protocol:

```
proc simulator(h : statement, e : challenge) = {
  z <$ FDistr.dt;
  a = g^z*h^(-e);
  return (a, z);
}
```

To prove SHVZK be must the prove output indistinguishability of the following procedures: To prove this we use EasyCrypt coupling functionality to show that $r^{real} \equiv z^{ideal} - e \cdot w^{real}$

```
proc real(h, w, e) = {
    r <$ FDistr.dt;
    a = g^r;
    z = r+e·w;
    return (a, e, z);
}
```

```
proc ideal(h, e) = {
  z <$ FDistr.dt;
  a = g^z*h^(-e);
  return (a, e, z);
}
```

and that $z^{ideal} \equiv r^{real} + e \cdot w^{real}$. This is easily prove, since the distribution is full and uniform, and the group is closed under addition and multiplication. All these facts follow directly from the cyclic group theory in EasyCrypt. By the coupling functionality we are then for the rest of the proof allowed to assume: $r^{real} = z^{ideal} - e \cdot w^{real}$ and $z^{ideal} = r^{real} + e \cdot w^{real}$. We then use this to show output indistinguishability:

$$\begin{aligned}
(a^{real}, e, z^{real}) &= (g^{r^{real}}, e, r^{real} + e \cdot w^{real}) \\
&= (g^{r^{real}}, e, z^{ideal} - e \cdot w^{real} + e \cdot w^{real}) \\
&= (g^{z^{ideal} - e \cdot w^{real}}, e, z^{ideal}) \\
&= (g^{z^{ideal}} \cdot g^{w^{real} - e}, e, z^{ideal}) \\
&= (g^{z^{ideal}} \cdot h^{(-e)}, e, z^{ideal}) \\
&= (a^{ideal}, e, z^{ideal})
\end{aligned}$$

Which is proven by EasyCrypt's automation tools. □

►Define generalised notation for comparing views in background chapter◄ ►The proofs have been relatively easy thanks to the strong support for algebraic groups in EC◄

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 (Section 5.4). We have also seen how it is possible to prove the security of Σ-Protocols working on composite relations like AND and OR (Section 5.2). The main problem with these solutions is that they require a specialised Σ-Protocol for every non-composite relation. In our case we have a protocol we can use for proving knowledge of the discrete logarithm relation, but what if we also want to prove knowledge for another computational problem? With our current framework we would have to define a new Σ-Protocol exclusively for this relation.

The problem with this is that 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.

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 …

We will in this chapter introduce the generalized zero-knowledge compiler, ZKBoo, by Giacomelli et al. [11]. When doing so we aim to provide an general overview of how the protocol works, whilst recalling key definitions and proofs from the paper.

6.1 ZKBOO

►Based on MPC in the head◄ ►Needs a semi-honest MPC protocol◄ ►Efficient because of semi-honest requirement◄ ►privacy implies ZK◄ ►Can only cheat verifier if he is unlucky in the view he opens◄ ►One view needs to be inconsistent to produce valid output for invalid input◄ ZKBoo protocol 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.

►define view. different from normal MPC definition◄

The principle idea of this protocol is based on a technique called "MPC in the head". Recall from section 3.4, that Multipart Computations allows us to securely compute any given function taking n inputs to an output y. We then have by definition 3.4.2 that as long as only $d \leq n$ 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 get the following:

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 $d \leq n$ 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 Σ-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*

►MPC protocol but local◄ (2,3)-Function decomposition is a general technique for computing the output of a function $f : X \to Y$ on input value $x \in X$. The decomposition works by splitting the function evaluation into three computational branches where each computation branch is a party in a MPC protocol. Each party is then allowed to communicate with each other, but observing the computation of any two of the parties will reveal no information about the input value x. Through-out this section we will simply refer the (2,3)-Function decomposition of a function f as $\mathscr{D}_f$.

We refer to the three parties of the decomposition as P_1, P_2, P_3. The decomposition then works by converting the function f into a circuit and giving each party a share, where the original input can be obtain if all three shares are acquired. Each party then evaluates the gates in the circuit to a new share based on the input they are given. party P_i is allowed to communicate with party $P_{i+1 \mod 3}$, but since every party in run locally it effectively means that party P_i has access to the entire view of $P_{i+1 \mod 3}$ for the entire duration of the protocol. The view of a party is then a list of all the shares that the party has computed so far. The view of party P_i is referred to as w_i For the rest of this section we will omit the mod 3 from the indexing. Moreover we assume that each party has access to a random tape k_i which describes what the party should do if the protocol asks for a random choice.

Definition 6.1.4. In its most general form the decomposition is a collection of functions:

$$\mathscr{D} = \{\text{Share}, \text{Output}, \text{Rec}, \text{Update}\}$$

Where Share is a procedure for distribute an input into three shares. Moreover, it should be possible to invert the share procedure such that the original input can be recovered from the three input shares. Output is a function returning the output share from the view of a party. Rec is a function reconstructing the output of the function f based on the output values of the parties.

Lastly we have Update(view_i^j, view_{i+1}^j, k_i, k_{i+1}) $= view_i^{j+1}$ which is the function used to evaluate the j'th gate of the circuit from the point of view of P_i. Here j also refers to the size of the view, i.e. how many shares has been computed so far.

The (2,3)-Decomposition is then the three views produced by running Update on each party with input shares produced by Share until the entire circuit has been evaluated by each party.

Based on the security definitions from MPC (Section 3.4) we can then define the two necessary properties from [11] for security of our (2,3)-Function decomposition, namely, correctness and privacy.

Definition 6.1.5 (Correctness)**.** A (2,3)-decomposition $\mathscr{D}_f$ is correct if $\forall x \in X, \Pr[f(x) = \mathscr{D}_f(x)] = 1$. **►Change notation to account for randomness◄**

Definition 6.1.6 (Privacy)**.** A (2,3)-decomposition $\mathscr{D}_f$ 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

Based on the general description of the (2,3)-Decomposition from the previous section we can then define concrete procedures needed to compute the (2,3)-Decomposition of arithmetic circuits as in Giacomelli et al. [11].

We assume the circuit is expressed in some arbitrary finite field $\mathbb{Z}_q$ such that the circuit can be expressed by gates: addition by constant, multiplication by constant, binary addition, and binary multiplication. Assume that every gate in the circuit is labelled as $[1 \dots N]$ where N is the total number of gates. We then implement $\mathscr{D}_{\text{ARITH}}$ as:

- $\text{Share}(x, k_1, k_2, k_3)$: Sample random values x_1, x_2, x_3 such that $x = x_1 + x_2 + x_3$
- $\text{Output}(w_i) = y_i$: return the share corresponding to the output wire of the gate.
- $\text{Rec}(y_1, y_2, y_3) = y_1 + y_2 + y_3 = y$ where y is the value of evaluating the circuit normally.
- Update(view$_i^j$, view$_{i+1}^j$, k$_i$, k$_{i+1}$): Here we define procedures based on what type the j'th gate is. Since update only append a new share to the view of the party we only define how to compute the new share, since the old shares are immutable.
  - Addition by constant: where a is the input wire to the gate and α is the constant.

$$w[j+1]_i = \begin{cases} w_i[a] + \alpha & \text{if } i = 1 \\ w_i[a] & \text{else} \end{cases}$$

  - Multiplication by constant: where a is the input wire to the gate and $alpha$ is the constant

$$w_i[j+1] = w_i[a] \cdot \alpha$$

  - Binary addition: where a, b are the input wires.

$$w_i[j+1] = w_i[a] + w_i[b]$$

  - Binary multiplication: where a, b are the input wires.

$$w_i[j+1] = w_i[a] \cdot w_i[b] + w_{i+1}[a] \cdot w_i[b] + w_i[a] \cdot w_{i+1}[b] + R_i(j+1) - R_{i+1}(j+1)$$

    Where $R_i(j+1)$ is a uniformly random function sampling values using k_i

Here the binary multiplication gate is the most interesting since it needs the share from another party to compute. The random values are added to hide what the share of the other party where. If the random values where not added then if would be easy to deduce what the share of P_{i+1} where given access to the view of party P_i.

►ZKBoo omits implementation detail - what is the output wire of the gate◄

►Replace ϕ with f?◄

6.1.2 *ZKBoo*

Based on the (2,3)-Decomposition we are now ready to describe the Σ-Protocol for the relation R x y = $f(x) = y$.

The protocol proceeds as follows:

- The Prover run obtains the circuit representation C_f of f and uses $\mathscr{D}$ to produce three views w_1, w_2, and w_3. The Prover then commits to all random choices and the views and sends the output shares y_1, y_2, y_3 of the decomposition and the commitments to the Verifier
- The verifier pick a number $e \in \{1,2,3\}$
- The Prover sends views w_e, w_{e+1} to the Verifier
- The Verifier checks
  - The commitments corresponds to the views
  - The view w_e has been constructed by $\mathscr{D}$
  - $\text{Rec}(y_1, y_2, y_3) = y$

From this protocol we can see that if $\mathscr{D}_f$ is correct and we get access to all three views then we would be able to extract the witness of the relation, since the output of decomposition is equivalent to the result of the function it decomposes. By only revealing 2 of the three views we are ensured by the 2-privacy property of $\mathscr{D}$ that the protocol is zero-knowledge. This property is stronger than the one given by Σ-protocols, which only offers zero-knowledge if the verifier is honest. The problem, however, is that the Prover gives the Verifier access to the commitment of the last view, so if the view can be determined based on the commitment then the zero-knowledge property does not hold.

Lastly, if the Prover is to cheat the Verifier he must produce three views where the output is y. The only way for the Prover to do this is to change some of the shares in one of the views to coerce the output. By doing so one of the views will deviate from the procedures of $\mathscr{D}_f$, which the prove can easily check if the pick the correct challenge.

To prove that the above claims holds and that the ZKBoo protocol is secure we will in the following chapter use the work laid out in this thesis to develop a formalisation of the ZKBoo protocol that captures the aforementioned security aspects.

7

FORMALISING ZKBOO

►Mention first to formalise this?◄ In this chapter we show how to use our previous formalisations of Σ-Protocols and commitment schemes to formalise the ZKBoo protocol introduced in the previous chapter. Here we will shown how to instantiate ZKBoo with our Σ-Protocol formalisation and then show how we can use our formalisation of commitment protocols to prove the security of ZKBoo.

The goal of formalising ZKBoo is two-fold. First, we show that our previous formalisations are indeed applicable to larger protocols. Second, we aim to gather insight into how EasyCrypt can help in formalising protocols larger than the usual toy examples like El-Gamal and Schnorr.

OUTLINE First, in section 7.1 we find develop a formalisation of arithmetic circuits within EasyCrypt , which allows us the reason about evaluating the circuit to a value and guide the formalisation of the decompostion. Next, in section 7.2 we formalise the (2,3)-Decomposition of arithmetic circuits as defined in section 7.1. This ultimately leads to section 7.3 where we use the formalisation of arithmetic circuits and their (2,3)-Decomposition to instantiate ZKBoo as a Σ-Protocol and then prove its security.

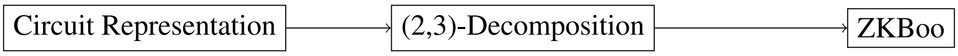

Figure 4: Outline of ZKBoo formalisation

PRELIMINARY NOTATION Throughout this chapter use the letter e to denote an integer in $\{1,2,3\}$. Moreover, we define arithmetic on e such that $3+1=1$.

To refer to the i'th index of a list w we use the notation $w[i]$.

7.1 FORMALISING ARITHMETIC CIRCUITS

In this section we will introduce the concept of arithmetic circuits and how they can be represented. Primarily we recall the usual definition of circuits as graph and discuss how to evaluate circuits programmatically. From this we introduce a number of restrictions to our formalisation which makes their easier to work with, whilst still being expressive enough to use in the ZKBoo protocol. Based on these restrictions we then formulate an alternative representation for arithmetic circuits and give a number of key definitions, which are needed for reasoning about the structure of a circuit and the evaluation of circuits.

7.1.1 *Representing an arithmetic circuit*

An arithmetic circuit is in its most general form express a function ϕ over some arbitrary field $\mathbb{Z}_q$, where $\phi : \mathbb{Z}_q^k \to \mathbb{Z}_q^l$

To express arbitrary (Arithmetic) computations in a ring of finite field we use the following four gates, addition by constant (ADDC), multiplication by constant (MULTC), addition of two wires (ADD), and multiplication of two wires (MULT).

The goal of this section is to formulate a representation of the function ϕ, which only depends on the aforementioned gate types to perform computations. Before doing so, however, we start by stating a number of simplifying assumptions about our arithmetic circuits. First, we only allow the circuit to have one input value and one output value, in other words: $k = l = 1$ in the definition of ϕ. This assumption exists pure to make it easier to reason about the inputs and output of the function. The formalisation in this section could be altered to allow for arbitrary inputs with minor alterations, which we will discuss later. **►Discuss this later◄**

Based on these simplifying assumptions we can now recall the graph representation of a circuit:

Definition 7.1.1 (Arithmetic Circuit)**.** An Arithmetic circuit is a graph C = (W, G) where W is the internal wires between the gates and G is the set of gates within the circuit. Then, we let $i \in G$ be first gate of the circuit, i.e. its only input and $o \in G$ be the output gate, for which there must exists a path from i to o in W. Specifically, o is a gate with only one in-going wire and no out-going wire. The means that the value of the circuit can be read by reading the value of the in-going wire to o.

►Define in- and out-wires?◄

Finally, for all gates $g \in G$ there must exists path in W from i to o going through g, since if this was not the case, the gate does not contribute the output of the gates and can therefore be removed from the graph without changing the semantic meaning of the circuit.

To define evaluation of a circuit we would then need compute the value of the in-wire of o, but this can only be done if we have computed all other wires in the circuit. Moreover, the value of the out-wires of a given gate, g, can only be computed if all in-wires of g has already been computed to a value. It is clear from this that we need to define an order of evaluation, such that we only try to compute the out-wire of a gate if we know that all in-wires has been computed.

To define the order of evaluation we follow the work of [3] and introducing an alternative representation of Arithmetic circuits, which naturally gives us a well-defined evaluation order:

Definition 7.1.2 (List representation of arithmetic circuits)**.** Given an arithmetic circuit C as defined by definition 7.1.1 we define the list representation of C as computing a linear ordering, O, of G, which gives each gate in G an unique index. **►Why do we remove i and o?◄** We then let the list representation C_L be defined as:

$$C_L[j] = Enc(O(G \setminus \{i\})[j], W)$$

Where $Enc : gate \to W \to$ encoded gate, is a function taking as input a gate and the wires of the circuit and produces an encoded gate. The encoded gate contains type information about gate but also stores the indexes (From the linear ordering) of the gates, whose out-going wires are the in-going wires of the gate. The type declaration of the encoded gates can be seen in figure 10.

►Tikz example showing the two representations◄

►This representation does not allows for optimisations like parallel computations◄

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

Listing 10: Type declaration of gates

LINEAR ORDERING A linear ordering O is a function that when applied to G assigns a unique index to each gate in G. One example of such function defining a linear ordering is a breadth-first search, where each gate in the circuit graph is labelled according to at which time the BFS reached the gate. This labelling would start at gate i and end at o.

A special property of the linear ordering induced by a BFS labelling is that a gate can only be visited when all nodes that the gates computation can depend on has already been visited. This ensure that for a node indexed i it only depends on out-wires of nodes with index less-than i.

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 by the linear ordering. However, since the gates i performs no computation and only exists to add the input value to the graph we exclude it from the list representation, and shift every index one down.

ENCODED GATES 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.

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 10.

One important aspect of the list representation of circuits is that it allows us to easily define an evaluation order, where we are ensure that we are not computing the value of a gate before the previous gates has been computed. To capture this notion of a valid evaluation order we give the following definition:

►Define notation for list indexing?◄

Definition 7.1.3 (Valid circuit)**.** An arithmetic circuit in list representation C_L 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.

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 11. The evaluation is broken into two parts: First we have a function for evaluating one gate to an intermediate values. Second, we have a procedure for evaluating the entire circuits which calls the former function. To evaluate a single gate, we first need to determine which gate it is. This can be done by utilizing the power of the EasyCrypt type system, which allows us to pattern match on the type of the gate as seen in listing 11.

```
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 11: Circuit evaluation function

Then, if the circuit is valid the evaluation order is the indexes of the list representation. We know that if we are computing index i of the circuit, then indices $[0 \dots i-1]$ have already been computed. Perform the appropriate function then reduces to looking up the values of the previously computed gates and applying them to the function appropriate for the type of the gate.

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.

Definition 7.1.4 (State of list representation)**.** For a list representation of a circuit C_L we give the following recursive definition of the state:

$$\text{state}[0] = \text{input value}$$
$$\text{state}[i > 0] = \text{eval\_gate } C_L[i-1]\ state[i-1]$$

Here we recall that input gate has been removed from the list representation and $C_L[0]$ is the first non-input gate in the circuit. From this it also follows that

$$\text{size state} = \text{size } C_L + 1 \tag{12}$$

We then have that any valid circuit c can be compute to a value y as eval_circuit(c, [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 7.1.5 (Function/Procedure relation)**.** ∀ f, inputs, output: f(inputs) = output $\iff$ $\Pr[f(inputs) = output] = 1$.

Proof.

- “⇒”: Trivial
- “⇐”: by contradiction?

□

►ZKBoo paper gives no notion of a valid evaluation order - needed for security◄ ►Graph representation isomorphic to list representation?◄

7.2 (2,3) DECOMPOSITION OF CIRCUITS

In this section we ... formalisation based on the description of arithmetic circuits... **►mention MPC◄**

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. **►rewrite? - Random tapes = set of random choices◄** More specifically the decomposition work by incrementally 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 12, where it is assumed access to a function eval_gate, which has signature: $eval\_gate : circuit \Rightarrow party \Rightarrow (view * view) \Rightarrow (random\_tape * random\_tape) \Rightarrow share$, **►explain eval gate better?◄** 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 aforementioned procedure. More formally the output is:

$$\text{Output}(w1, w2, w3) = \sum_{i \in \{1,2,3\}} \text{last } w_i \tag{13}$$

Definition 7.2.1 (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:

$$\forall 0 \leq i < \text{size } c, \sum_{p \in \{1,2,3\}} w_p[i] = s[i] \tag{14}$$

where s is the list of intermediate values produces by calling eval_circuit_aux in figure 11.

Additionally a share is only valid, if it has been produced by functions used by the decomposition. This property also ensures that the views are consistent wrt. each other. Namely, if p_i computes a share s then s is the share that the other parties received from p_i during the execution of the protocol.

$$\forall 0 \leq i < \text{size } c - 1, w_e[i+1] = \text{eval\_gate } c[i] \; w_e \; w_{e+1} \tag{15}$$

To express that the views satisfy the above definition we use the notation **Valid**$(c, w1, w2, w3)$ to express that $w1, w2, w3$ are valid views for the decomposition of 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 12: Incremental decomposition procedure

HANDLING RANDOMNESS Looking at compute we see that compute we make three random choices for each gate and then save those choices to some random tapes. These tapes are then returned. They are used to keep track of the random choices made thoughout the protocol, such that views can be verified. For most proof the specific values contained within the random tapes are not imporant, since the random values are only there to cancel eachother out whilst making the shares look randomly distributed. We therefore omit the random tapes for most procedures and proofs in this section, since they are static? When the random tapes are imporant for the security we will mentioned how.

The reason for sampling random values at each iteration of the while loop instead of all at once is to be able to reason about the specific random choices made at this time in the computation. This is especially important to be able to relate two procedures running at the same iteration of the while loop... **►Show this in section about privacy◄**

SECURITY To prove security we...

7.2.1 *Correctness*

Ultimately want to prove:

Lemma 7.2.2 (Decomposition correctness)**.**

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

i.e. The output distribution of the two programs are perfectly indistinguishable. From lemma 7.1.5 we have that circuit evaluation always succeeds. This lemma, therefore, also implies that the decomposition always succeeds.

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:

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

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 7.2.2. 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 13. simulator_eval is a function that evaluates a single gate from the point of view of party "p". In 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 from the other 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 his own shares 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. This is true by equation **►Make sim mult function in zkboo chapter◄**, where the difference between two random values are added to the share, effectively making the share appear random too. **►rewrite this◄** 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.

►Where we use that compute and simulate sample randomness at the time of computation◄

```
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 13: Simulator

The procedure simulate is simply a wrapper around simulator_eval, which is responsible for constructing the views and sampling randomness incrementally for each gate in the circuit, much like how compute is a wrapper around eval_gate.

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 14.

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.

►Why must h be in the domain of R here, but not in correctness?◄

This can be stated in rPHL as:

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

To prove this lemma we first prove 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 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 ...

```
proc real((c,y) : statement, w : witness, e : challenge) = {
     ($y_1,y_2,y_3,w_1,w_2,w_3$) = compute(c);
     return ($w_e,w_{e+1},y_{e+2}$)
}

proc simulated((c, y) : statement, e : challenge) = {
     ($w_e,w_{e+1}$) = simulate(c, e);
     $y_e$ = last $w_e$;
     $y_{e+1}$ = last $w_{e+1}$;
     $y_{e+2}$ = $y-(y_e+y_{e+1})$
     return ($w_e,w_{e+1},y_{e+3}$)
}
```

Listing 14: Real/Simulated view of decomposition

This is stated as the following lemma:

►Define notation for referring the views from protocol one and the views from protocol 2◄

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 w_1, w_2, w_3 satisfies the correctness property from equation 14.

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 [6]?

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

- Possible due to inlined randomness sampling
- If tape has been sampled before the run we cannot use EC's coupling functionality to reason about the random choices.

□

Proof of lemma 7.2.4. By applying lemma 7.2.5 we have that the views output by both procedures are indistinguishable. All we have left to prove is that $y_{e+2}^{real} \sim y_{e+2}^{simulated}$. To prove this we use the equation 14, which states that the shares of the real views always sum to the intermediate values of computing the circuit to conlude

$$y = y_1^{real} + y_2^{real} + y_3^{real} \iff y_{e+2}^{real} = y - (y_e^{real} + y_{e+1}^{real})$$

Then by $(y_e^{real} + y_{e+1}^{real}) \sim (y_e^{real} + y_{e+1}^{real})$ it follows that

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

□

7.3 ZKBOO

►Throughout this section we will introduce the instantiated sigma protocol...◄ Since the ZKBoo protocol is an instantiation of a Σ-Protocol we start by defining the types as specified in section 5.

```
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).
```

The relation is then all tuples of circuits outputs and inputs, where it holds that evaluating the circuit with the input returns the output. We formally encode this as

$$\mathrm{R} = \{((c,y),w) \,|\, \text{eval\_circuit } c\, w = y\}. \tag{16}$$

We then add the restriction, that the challenge is always a integer in $\{1,2,3\}$. Moreover, we recall from section 6.1, that ZKBoo depends on a commitment scheme. Here we follow [11] and use a key-less commitment scheme. We therefore assume the existence of a commitment scheme, Com, which is an instantiation of the key-less commitment scheme formalisation from section 7.3. Furthermore, we simply our proof burden by requiring Com to satisfy the alternative perfect hiding property from definition 4.4.2 as well as the alternative binding property from definition 4.4.3 with probability *binding_prob*.

With these preliminaries in place we are now ready formalise the ZKBoo protocol. First, we start by defining the sub-procedures needed for the verify procedure. Recall from section 6.1, that the Verifier accepts a transcript (a,e,z) if z is a valid opening of the views w_e and w_{e+1} commitment to in a and that every entry in w_e has been produced by the procedure defining the decomposition. This step of validating that w_e has been produced in accordance with the decomposition is given by equation 15. This equation can be encoded within EasyCrypt as a predicate: **►Change the procedure to use the naming from the report◄**

```
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).
```

Predicates allows us to use quantifiers to assert properties within EasyCrypt, 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. Having a predicate reasoning quantifying over all integers, for example, is perfectly legal, but this is obviously not possible to express as a computation, since it would take indefinitely many computations to verify a property for indefinitely many integers. A predicate, therefore, cannot be used within procedures, since they are not required to be computable. The quantification in equation 15, however, only need finitely many computations to verify the property, since it is bounded by the size of the circuit. We can, therefore, define a computable function which for each entry check if the property holds and then returns if the property holds for all entries. This can clearly be computed in time proportional to the size of the circuit and the time it takes to compute one share of the decomposition. This function is given by:

```
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))).
```

This function allows us to computationally validate the property from equation 15, but it is harder to reason about, since we have to reason about every computational step of the function before we can verify the property holds. We would therefore want our function to use our function in the implementation of ZKBoo, but use the predicate whenever we need to reason about the security of the protocol. To achieve this use introduce the following lemma by Almeida et al. [3], which allows us to replace the result of the function with the predicate:

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 Σ-Protocol in our formalisation by implementing the algorithms from figure 4, which can be seen in figure 15. **►Assumes existence of decomposition protocol◄**

7.3.1 *Security*

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

ASSUMPTIONS We assume that ZKBoo is given access to a secure key-less commitment scheme Com which is not allowed to alter the global state of the ZKBoo module. Moreover we assume that that Com satisfy the perfect hiding definition 4.4.2 and can win the alternative binding game given in definition 4.4.3 with probability *binding_prob* and that the commit to a message using Com can never fail.

The requirement of Com not being able to access the global state of ZKBoo is an important one, since it none of the below proofs would be true without it. This assumption for security is especially important to remember when implementing the protocol in a programming language where all variables are stored in a global state like Python.

Furthermore, we assume that ZKBoo is given access to a secure (2,3)-Decomposition of the circuit.

Lemma 7.3.2. ZKBoo satisfy Σ-Protocol completeness definition 5.1.1.

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.

```
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 15: ZKBoo Σ-Protocol instantiation

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;

  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 16: 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 $e = 1$: By our assumption of the committing to a message never failing we can remove the line committing to view w_{e+2} since it does not influence the output of the procedure. Next, since the commitment scheme and the decomposition are correct are we left with showing that valid_view_op return true. To reason about this we use lemma 7.1.5. From this it follows that the predicate must be true by correctness of the decomposition.
case $e = 2 \wedge e = 3$ follow the same steps as above. □

Lemma 7.3.3. Assuming perfect hiding from definition 4.4.2 then ZKBoo satisfy Special Honest Verifier Zero-knowledge definition 5.1.5

Proof. ►**Would have to be expressed as an adversary game if we used original definitions from commitment schemes**◄ 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 7.2.1. This is enough to make the verify procedure return true.

```
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 = (y_1,y_2,y_3,c_1,c_2,c_3)
  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
    w_1);
     k_{e+2} = dlist dinput (size k_1
    );
     y_e = last 0 w_e;
     y_{e+1} = last 0 w_{e+1};
     c_i = commit((w_i,k_i));
     a = (y_1,y_2,y_3,c_1,c_2,c_3);
     z = (w_e,w_{e+1});

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

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. To prove this formally we use perfect hiding definition (4.4.2) which states that two programs run in parallel and making the same choices will be indistinguishable. Since we assume perfect hiding we can then by the perfect hiding definition assume $c^{real}_{e+2} = c^{ideal}_{e+2}$ for the rest of the proof.

The rest of the output values are then indistinguishable by the 2-Privacy property. □

►Change order so this lemma comes last◄ ►Problem: compute sample random values - tape is complete already in this instance◄

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. **►If we had used to original definition of hiding for Com then this lemma would have to be restated◄** 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 To check if the views are valid we use the verify procure, which check that valid_view_op return true. By lemma 7.1.5 we know that this is equivalent to equation 15. From this we can define the following procedure for checking validity and consistency of the openings:

```
proc extract_views(h : statement, m : message, z1 z2 z3 :
    response) = {

  v1 = verify(h, m, 1, z1);
  v2 = verify(h, m, 2, z2);
  v3 = verify(h, m, 3, z3);

  (w1, w2) = z1;
  (w2', w3) = z2;
  (w3', w1') = z3;
  (y1, y2, y3, c1, c2, c3) = m;
  cons = bind_three(c1, c2, c3, (w1, k1), (w1', k1'), (w2, k2),
    (w2', k2'), (w3, k3), (w3', k3'));

  return v1 /\ v2 /\ v3;
}
```

Listing 17: Consistency procedure

Here we are given two potential openings for each view, namely w_e and w'_e. Ideally $w_e = w'_e$ but if it is possible for the adversary to win the binding game for three commitments then the openings might be inconsistent. We therefore let the procedure call bind_three, which returns true if the adversary has broken the binding game. This is bound to a variable cons, which is not returned. This allows us to encode the consistency check as auxiliary information such that the return value of the procedure is still equivalent to only calling verify on the three responses.

►bind_three => (a = a') and valid views◄ ►Proven by phoare split◄

From this we can state the following:

Lemma 7.3.5. $\Pr[\text{extract\_views}(h,m,z_1,z_2,z_3) : v_1 \wedge v_2 \wedge v_3 \wedge w_i = w'_i] = (1 - \mathit{binding\_prob})$

Proof. Special soundness assumes that all transcripts are accepting, we can therefore conclude that $v_1 \wedge v_2 \wedge v_3$ must hold. We are then left with showing that bind_three proves that the views are consistent with probability $(1 - \mathit{binding\_prob})$. This is true by our assumption of Com having binding. □

WITNESS EXTRACTION Given that all openings correspond to the same call of compute and $\mathbf{Valid}(c, w_1, w_2, w_3)$ we must then show that $w = w_1[0] + w_2[0] + w_3[0] \implies y = \text{eval\_circuit}(c, w)$ i.e. the witness is the sum of all the input shares to the parties of the decomposition.

$$\begin{aligned}
&\text{eval\_circuit}(c, w_1[0] + w_2[0] + w_3[0]) = y \\
\iff &\Pr[\text{eval\_circuit}(c, w_1[0] + w_2[0] + w_3[0]) = y] \\
= &\Pr[(w'_1, w'_2, w'_3) \leftarrow \text{compute}(c, w_1[0], w_2[0], w_3[0]); \left(\sum_{i \in \{1,2,3\}} \text{last } w'_i\right) = y]
\end{aligned}$$

We then show that we can traverse the computations of the decomposition in reverse:

Lemma 7.3.6. $\Pr[(w'_1, w'_2, w'_3) \leftarrow \text{compute}(c, w_1[0], w_2[0], w_3[0]); (\sum_{i \in \{1,2,3\}} \text{last } w'_i) = y]$

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 alt_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;
  }
}
```

Lemma 7.3.7. The above procedure has output distribution indistinguishable from the soundness game from definition 5.1.4 instantiated with ZKBoo, i.e. $\Pr[\text{alt\_soundness}] = \Pr[\text{soundness(ZKBoo)}]$

Proof. By inlining all sub-procedure calls from both procedures we have equivalent calls to verify and witness_extractor. The only differences between the two procedures is that consistent_views call the binding game, but the value from the binding game is never returned, so it does it affect the output distribution. □

We can then show conclude the proof of the main lemma by applying lemma 7.3.7. From this we need to show: $\Pr[\text{alt\_soundness} : \mathit{true}] = (1 - \mathit{binding\_prob})$. Which follows from applying lemma 7.3.5 and 7.3.6. □

►Formal verification does not tell us about efficiency◄

CONCLUSION In this chapter we have seen how to apply our formalisations of Σ-Protocols and commitment schemes to a MPC based protocol...

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...

definitions are annoying to work with.

►Have to take care to ensure that the decomposition can be backtraced◄

```
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 18: ZKBoo witness extractor

8 REFLECTIONS AND CONCLUSION

8.1 RELATED WORKS

This work exists in the field of formal verification of cryptographic protocols. Notably our work has been heavily influenced by similar formalisations [2, 3, 6, 8, 13]

Butler et al. [8] managed to formalise both Σ-Protocols and commitment schemes within Isabelle/CryptoHOL. Additionally, they have managed to prove that commitment schemes can be build directly from Σ-Protocols. Their formalisation of Σ-Protocols also include various concrete instantiations. The main difference between the results obtained in their work compared to our has been the tool usage. Isabelle/CryptHOL is a tool similar to EasyCrypt that offers a higher-order logic for dealing with cryptographic game-based proofs. The fundamental difference between the two tools is that Isabelle/CryptHOL programs are written in a functional style, where as EasyCrypt allows the user to write programs in an imperative style. This ultimately leads to the same understanding of programs as distribution transformers as discussed in chapter 2.

Other formalisations of Σ-Protocols also exists. Barthe et al. [6] successfully formalised Σ-Protocols with CertiCrypt. They work includes a formalisation of Σ-Protocols where the relation is the pre-image of a homomorphism with certain restrictions or a claw-free permutation. This has allowed them to define and prove the security for a whole class of Σ-Protocols. This result is similar to the one we achieved with out formalisation of ZKBoo. ZKBoo, however, defines a more general class of Σ-Protocols than the one defined in the paper.

Moreover, commitment schemes has been formalised in EasyCrypt by Metere and Dong [13]. Their work differs from our by offering less definitions of security, which we described the need for in chapter 4

Notable work also exists for formalising generalised zero-knowledge compilers. Almeida et al. [2] developed a fully verified zero-knowledge compiler in CertiCrypt which uses the generalised Schnorr protocol to produce zero-knowledge proofs of any relation defined by the pre-image of a group homomorphism, just like ZKBoo. The generalised Schnorr protocol, however, is a fundamentally different protocol than ZKBoo, in the sense that it does not use MPC or commitment scheme.

Last, secure function evaluation has been studied by Almeida et al. [3], which formalised Yao protocol in EasyCrypt. This work also included a formalisation of circuits.

8.2 DISCUSSION

For all of the work laid out in this thesis we have used the EasyCrypt proof assistant to formally verify all the proofs shown. By using EasyCrypt we have had to formulate our proofs ...

EasyCrypt tries to capture the models in which cryptographers create and prove protocols. For the most part we feel like EasyCrypt has managed to capture these models quite well both in its *pWhile* language for implementing protocols and its different logics for proving properties about programs.

The benefit of using an imperative language like *pWhile* over a functional language used by cryptographic proof assistants like Isabelle/CryptHOL is that most cryptographic protocols are described in a pseudo-code mimicking imperative languages. This makes it relatively easy to convert protocols described in papers into EasyCrypt implementations. This is clear from the code examples provided in this thesis, which closely resemble the actual EasyCrypt implementations whilst being relatively similar to the protocol descriptions seen in cryptographic papers.

Ultimately, the tool offers the possibility of writing programs both in a functional style and in an imperative style. It is, however, only programs written in the imperative style that is allowed to make random choices.

We feel...

Our main problems with using this tool has been the schism between computation and perfect indistinguishability and the tools steep learning curve.

In particular EasyCrypt offers its rPHL for proving procedures to be perfect indistinguishable. If, however, computational indistinguishability is need then the rPHL logic cannot directly be used, and we instead have to deal with adversaries comparing procedures. This is a completely sound method for dealing with indistinguishability of procedures. In EasyCrypt, however, the technique of using an adversary to compare procedures then none of the tactics for dealing with procedures provided by EasyCrypt can be used, unless all proofs are converted into adversarial games. **►rewrite◄**

The steep leaning curve is primarily caused by the lack of documentation of new tactics. At the time of writing this thesis the last update to the EasyCrypt reference manual [1] was in 2018. Moreover, the deduction rules by the different logics that EasyCrypt provides are not documented anywhere, but instead have to be found in the papers describing **CertiCrypt** which is the Coq-based proof assistant antecedent to EasyCrypt.

►How has EC been to work with◄ ►What is the future for cryptographers using EC◄ ►Possible code extraction?◄ ►Two styles of proof: adversary as indis◄ ►Differences?◄

►functions in EC are bound to theories not modules◄

8.3 FUTURE WORK

In this thesis we has created a formalisation and Σ-Protocols and commitment schemes that is applicable to larger cryptographic protocols, as show by our formalisation of ZKBoo. Various improvement has then been made to the ZKBoo protocol to mainly reduce to proof size but also to provide zero-knowledge in a post-quantum context [9].

With our formalisation we have intentionally focused on the ZKBoo protocol in isolation but in real applications it would be part of a larger tool chain. Mainly, ZKBoo requires a circuit with a definable execution order to be secure. In our formalisation we have assumed the input to be a circuit and defined an execution order but to complete the tool chain we would need a formalisation of a procedure converting functions to circuits and a formal proof of the induced execution order in section 7.1.1 being semantic preserving.

Moreover we saw in section 5.3 that there is a need for formalising the rewinding lemma to reason about soundness of the Fiat-Shamir transformation. Moreover, rewinding is a common technique for proving soundness of zero-knowledge protocols. Formalising the rewinding lemma would then allows us to reason about be general zero-knowledge protocols than the sub-class of Σ-Protocol which we have explored in this thesis.

►Prove connection between Σ and pok or arg ZK◄ ►Commitment: connection between original and alternative definitions◄

8.4 CONCLUSION

In this thesis we have successfully managed to develop a rich formalisation of Σ-Protocols and commitment schemes, whilst reproducing some of the key results of formalisation done in other proof assistant [6, 8]. From this formalisation we have managed to take MPC-based zero-knowledge compiler for general relations and managed to prove it to be secure in a formal setting by using our formalisations of both Σ-Protocols and commitments schemes. In doing so we showed how important details for achieving security is often glossed over in cryptographic literature...

The main contributions of this work has been recreating key results form other proof assistants and showing the workability of EasyCrypt, whilst also showing how our formalisation can be used to fuel future work by showing how it is possible to prove security of a more complex cryptographic protocol. Moreover, we have gained key insights into how EasyCryptworks and how to develop workable formalisations

Particular we have seen in section 7.3.1 how important small assumption are for security of implementations of cryptographic protocols. If one procedures is allowed to observe the state of another running on the system all proofs in the aforementioned section would not hold. These assumption are often left out when discussing cryptographic protocol design, but are important when reasoning about the security of the protocols when implemented in a programming language.

BIBLIOGRAPHY

[1] *EasyCrypt Reference Manual*. February 2018. URL https://www.easycrypt.info/documentation/refman.pdf.

[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] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, FranÃ§ois Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. Cryptology ePrint Archive, Report 2017/821, 2017. https://eprint.iacr.org/2017/821.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. *CoRR*, abs/1904.04606, 2019. URL http://arxiv.org/abs/1904.04606.

[5] 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.

[6] 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.

[7] Gilles Barthe, Benjamin GrÃ©goire, Sylvain Heraud, and Santiago Zanella-Beguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology, CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, January 2011. ISBN 978-3-642-22791-2. URL https://www.microsoft.com/en-us/research/publication/computer-aided-security-proofs-for-the-working-cryptographer/. Best Paper Award.

[8] 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.

[9] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. pages 1825–1842, 10 2017. doi: 10.1145/3133956.3133997.

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

[11] 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.

[12] Patrick McCorry, Siamak Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. 01 2017.

[13] 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.

[14] Rui Zhang, Rui Xue, and Ling Liu. Security and privacy on blockchain. *ACM Comput. Surv.*, 52(3), July 2019. ISSN 0360-0300. doi: 10.1145/3316481. URL https://doi.org/10.1145/3316481.