FORMALISING SIGMA-PROTOCOLS AND COMMITMENT SCHEMES IN EASYCRYPT

NIKOLAJ SIDORENCO, 201504729

MASTER'S THESIS

June 2020

Advisor: Bas Spitters

Co-advisor: Sabine Oechsner

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

When applying cryptographic protocols in real-line applications it is paramount that we can trust them to work as specified. However, many cryptographic proofs are based on informal reasoning and implicit knowledge. Moreover, many cryptographic proofs have become so complex, that they are considered unverifiable.

Advances in formal reasoning of cryptographic protocols has provided a methodology for constructing more rigours proofs which can eliminate informal reasoning. Moreover, formal reasoning allow us to construct machine-verifiable proofs.

In this thesis we use the EasyCrypt proof assistant to formally reason about two cryptographic primitives; Commitment Schemes and Σ-Protocols. First, we consider Commitment Schemes, which allows a Committer to bind himself to a chosen message. The Committer can then later reveal the message to a Verifier, which can be ensured that the Committer revealed the same message as the one the bound himself to. Next, we consider Σ-Protocols, which allows a Prover to convince a Verifier, that he knows some secret x, without ever revealing the secret to the Verifier. Additionally, we also consider compound Σ-Protocol, which allow us to combine Σ-Protocols. To formalise these two primitives we reproduce the results of previous formalisations to develop abstract specification, which can be used to instantiate secure concrete protocols. We show the workability by instantiating Pedersen Commitment Schemes and Schnorr's Σ-Protocol.

Using the formalisations of Σ-Protocols and Commitment schemes we formalise the ZKBoo protocol, which is a generalised Σ-Protocol, based on Multi-Part Computations, for proven knowledge for the pre-image of any group homomorphism. In formalising ZKBoo we successfully formalised arithmetic circuits and an MPC protocol on arithmetic circuits called the (2,3)-Decomposition. We then use our abstract specification to prove ZKBoo to be a secure Σ-Protocol. With this work we show the discrepancy between the security proofs in cryptographic papers compared to the reasoning requires to prove a protocol secure in the formal context.

A highlight of our formalisation, is the security proofs of ZKBoo. Here we show how to formally prove a complex cryptographic protocol, which depend on the abstract specifications of Σ-Protocols, Commitment Schemes, and Multi-Part Computations.

RESUMÉ

▶in Danish. . . ◀

ACKNOWLEDGMENTS

►...◄

Nikolaj Sidorenco,
Aarhus, June 14, 2020.

CONTENTS

1 INTRODUCTION

Since their introduction zero-knowledge arguments/proofs have become an important building block for complex cryptographic protocols. Zero-knowledge has, for example, been used to enable anonymous board-room voting [17] and to ensure privacy in blockchain applications [1]. Moreover, zero-knowledge also has important theoretical applications, where it is used by the GMW-compiler to turn any passively secure protocol into an actively secure protocol.

Due to zero-knowledge being applied in critical applications like blockchains, which could have real-life financial consequences if the security is broken, it is paramount that we can offer the highest levels of assurance in the protocols performing to specification. However, many proofs of security are considered unverifiable, partly due to informal reasoning and implicit arguments. This was also remarked by Bellare and Rogaway [10]:

> In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.

To harness this complexity Bellare and Rogaway suggest the code-based game-playing approach for doing security proofs. However, game-playing security is not the end-all of security, and the approach still allows for unrigorous proofs. Many cryptographic proofs still suffer from informal arguments and human errors when proving security.

However, with the code-based approach, we can apply formal verification to help us attain an even higher level of trust in our proofs by making them machine verifiable. This approach has been made possible by the recent advances in proof assistants specialising in formal verification of probabilistic programs [7]. It is important to remark, however, that formally verified does not imply absolute security. When formally proving cryptographic protocols, we do so in the context of a cryptographic model. Such models helps us by reducing the complexity of our protocols by limiting external influences or by giving access to idealised functionalities. If the assumptions made in the model used are unsound, then the protocol would still be insecure, even though it has been formally proven. However, by applying formal reasoning to cryptographic proofs we can expel all informal reasoning and implicit knowledge, thus attaining much more precise definitions of security.

An additional benefit of using tools 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 [6]. Cryptographic protocols written in a low-level machine language can then, through EasyCrypt, be formally proven secure and extracted to assembly code.

►Studying these tools are, therefore, important because...◄

This allows researchers to take a protocol description and then transfer their definitions to a tool like EasyCrypt, which can help expose informal reasoning in their proofs.

In an attempt to bring higher assurance to the field of cryptography, formal verification of various cryptographic primitives have commenced [7–9, 12].

The goal of this thesis is to further this formalisation effort by first improving the existing formalisation of Σ-Protocols in EasyCrypt. Σ-Protocols has the desirable attribute of having code-based game-playing definitions of security, which allows us to use EasyCrypt, a proof assistant for formal verification of cryptographic protocols in order to formulate and prove their security. Moreover, Σ-Protocols can be turned into zero-knowledge proofs with limited overhead [15]. To achieve this, we reproduce the results of Butler et al. [12] to develop a rich formalisation of Σ-Protocols in EasyCrypt, which can be used to drive the formalisation effort of cryptographic protocols further.

Additionally, we reproduce the formalisation of commitment schemes by Metere and Dong [18] and improve upon the work by offering alternative, yet equivalent, security definitions.

We then use our formalisation of Σ-Protocols to formalise ZKBoo by Giacomelli et al. [15], which uses multi-part computations and commitment schemes to make a generalised Σ-Protocol. With this work we aim to demonstrate how complex cryptographic protocols can be formally verified by using formalisations of cryptographic primitives. Moreover, we show how formal verification can help expose the implicit and informal assumptions often made in cryptographic papers.

To formalise ZKBoo we also have to formalise arithmetic circuits, which can help guide future formalisations. Moreover, formalisation of a special case of multi-party computation, where all parties are simulated locally, called the (2,3)-Decomposition, is needed. **►We formalise MPC?◄**

OUTLINE In chapter 2. we introduce the code-based game playing approach to cryptographic security. Moreover, we introduce the EasyCrypt proof assistant, and how it can be used to formally prove code-based protocols. chapter 3 introduce key concepts and definitions.

In chapter 4 we reproduce the results of Metere and Dong [18] to formalise commitment schemes and a concrete instantiation in the form of Pedersen's commitment scheme. Additionally, we formulate alternative security definitions, which can help apply our formalisation in more complex protocols.

In chapter 5 we explore the results by Butler et al. [12], which were formalised in the Isabelle/CryptHOL proof assistant. These results are then reproduced in EasyCrypt.

Chapter 6 introduces generalised zero-knowledge protocols and ZKBoo as well as an informal overview of the security proofs of ZKBoo given by Giacomelli et al. [15].

Next, in chapter 7 we formalise arithmetic circuit and the (2,3)-Decomposition of arithmetic circuits. We then conclude the chapter by formally proving the security of ZKBoo.

Last, we summarise our findings and their relation to other works within the area of formal verification.

IMPLEMENTATION The EasyCrypt files of the work explored in this thesis are publicly available on Github[1]

1 https://github.com/Nsidorenco/thesis

2 EASYCRYPT AND GAME-BASED SECURITY

In this chapter, we introduce the EasyCrypt proof assistant for proving the security of cryptographic protocols. EasyCrypt allows us to reason about cryptographic protocols in the code-based game-playing paradigm.

2.1 THE GAME-PLAYING APPROACH

In the game-playing approach, security is traditionally defined as a game, where the game expresses property of the protocol, which is run with an *adversary* [10]. Most commonly, a cryptographic protocol is an interaction of two or more participants. In the example of commitment schemes the protocol is run with a committer and a verifier. The adversary is then allowed to take control of one of the participants and then tries to break the intended design of the protocol. Formally speaking, we say that the adversary wins the game if the protocol outputs some event E. The goal of the adversary is then to maximize the probability of E. We then define some target probability, which is the probability of E occurring, given that the protocol is completely secure. The difference between target probability and the actual probability of E then denotes the actual security of the protocol and is called the *advantage* of the adversary.

To bound the probability of the adversary breaking the protocol we have to reason about all the possible choices he would make. To reduce the complexity of this *game reductions* are used. By formulating intermediate games, where we reduce the set of possible choices, it becomes easier to reason about the security of the protocol.

These game reductions all have some advantage for the adversary. The probability of the adversary winning the original game is then the probability of winning the game of the last reduction plus all accumulated advantage of all the previous game reductions.

2.2 EASYCRYPT

EasyCrypt provides us with three logics, which allows us to reason about game reductions between adverbially defined games: a probabilistic relational Hoare logic (**pRHL**), a probabilistic Hoare logic (**pHL**), and a regular Hoare logic. The pHL allows us to reason about the probability of a program terminating with some output/predicate. The rPHL allows us to reason about the indistinguishability of two programs. Furthermore, EasyCrypt has a Higher-order ambient logic, in which the three previous logics are encoded. This Higher-order logic allows us to reason about mathematical constructs. This, in turn, allows us to reason about mathematical constructs used in programs since we can lift the program logics into the ambient logic. Moreover, the ambient logic offers strong tools for automation. Notably, the ambient logic can call various theorem provers, which will try to finish the current proof. This allows us to automatically prove complex mathematical statements without having to recall the specific mathematical rules needed.

To model security in the code-based game-playing approach in EasyCrypt we first formulate the original security definition as a game in EasyCrypt's embedded programming language. We then prove a number of game reductions using the pRHL. Last, we reason about the probability of the last game in the series of reduction using the pHL.

2.2.1 *Types, Operators and Procedures*

In EasyCrypt we can view types as mathematical sets containing at least one element. Types can either be *abstract* or *aliases*. An abstract type is, in essence, a new type. While type aliases work on already defined types. Moreover, EasyCrypt allows us to define inductive types and types with multiple members.

```
type t. (* Definition of new type *)

type card = [
    | spades of int
    | heart of int
    | clubs of int
    | diamonds of int
] (* Enumerated type *)
```

Functions are declared with the "op" keyword in EasyCrypt. Here functions are not allowed to make indeterministic choices, which also implies that a function will always terminate. Furthermore, functions allow us to pattern match on types with multiple members:

```
op is_red (c : card) =
    with c = spades n => false
    with c = heart n => true
    with c = diamonds n => true
    with c = clubs n => false.
```

2.2.2 *Theories, Abstract theories and Sections*

To structure proofs and code, EasyCrypt uses 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:

```
require Int.

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

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. Consequently, the "export" keyword can be used in place on import to require the theory under a custom prefix.

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

ABSTRACT THEORIES To model protocols that can work on many different types, we use EasyCrypt's abstract theory functionality. An abstract theory allows us to model protocols and proof over generic types. In other words, in an abstract theory, all modules, functions and lemmas are quantified over the types declared in the file. There are currently two ways of declaring an abstract theory. First, by using the "theory" keyword which declares all defined types to abstract. Second, an abstract theory file can be declared by using the ".eca" file extension.

An abstract theory can then later be "cloned", which allows the user to define concrete types for the theory before importing it.

SECTIONS Sections provide similar functionality to that of abstract theories, but instead of quantifying over types, sections allows us to quantify everything within the subsection over programs with special properties and axioms provided, which are defined by the user.

An example of this is proving a protocol secure given access to a random oracle. To model a random oracle, we can then start a subsection and declare an oracle who always sample uniformly random values. All lemmas declared within the subsection will then have access to the random oracle.

An important note is that introducing axioms in a subsection cannot break the logic of EasyCrypt, since if we axiomatise $true = false$ in a subsection, then all lemmas become on the form $true = false \implies$ lemma, which only makes the proofs impossible to realise.

Sections are particularly useful if many lemmas require the same assumptions. By using a subsection we can automatically make the assumption available in every proof without having to explicitly write it in the lemma.

2.2.3 *Modules and procedures*

To model programs in EasyCrypt, the module construct is provided. A module is a set of procedures and a record of global variables, where all procedures are written in EasyCrypt's embedded programming language, pWhile.

The syntax of pWhile is described over a set $\mathscr{V}$ of variable identifiers, a set $\mathscr{E}$ of deterministic expressions, a set $\mathscr{P}$ of procedure expressions, and a set $\mathscr{D}$ of distribution expression. The set $\mathscr{I}$ of instructions is then defined by [4]:

$$\begin{aligned}\mathscr{I} ::= \ & \mathscr{V} = \mathscr{E} \\ & | \ \mathscr{V} < \$\mathscr{D} \\ & | \ \text{if } \mathscr{E} \text{ then } \mathscr{I} \text{ else } \mathscr{I} \\ & | \ \text{while } \mathscr{E} \text{ do } \mathscr{I} \\ & | \ \mathscr{V} = \mathscr{P}(\mathscr{E},\ldots,\mathscr{E}) \\ & | \ \text{skip} \\ & | \ \mathscr{I};\mathscr{I}\end{aligned}$$

Modules are, by default, allowed to interact with all defined modules. This is due to the fact that all programs are executed within shared memory. This models a real execution where the program would have access to all memory not protected by the operating system.

From this, the set of global variables for any given module is the set of all global variables defined by the modules and all variables the procedures of the module 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 be seen as EasyCrypt's abstraction of the class construct in object-oriented programming languages.

MODULES TYPES Modules types is another feature of EasyCrypt modelling system, which enables us to procedures of modules, without having to implement them. A procedure without implementation is abstract, while an implemented one, i. e. the ones provided by modules, is 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 about an abstract modules, since there is no context to prove properties within.

This allows us to quantify over all possible implementations of an abstract module in our proofs. The implications of this are that we can then prove that no matter what choice the adversary makes during execution, he will not be able to break the security of the procedure.

►explain clone◄

2.2.4 *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.

When dealing with distribution, we have three important characteristics:

Lossless : A procedure (or distribution) is said to be lossless if it always produces an output. 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 from a distribution as x <$ dt

By introducing random assignments to 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. When sampling a random value we can provide a coupling stating that the value sampled is indistinguishable from some value, x. If it is possible to prove the two values are indistinguishable, then we can use the value of x in place of the random value, for the rest of the procedure.

►Mention common distributions like bool and interval◄

2.2.5 *Probabilistic Hoare Logic*

To formally prove a cryptographic protocol secure, we commonly have to argue that with some probability a probabilistic procedure will terminate with a certain event.

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

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

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

Alternatively, this can be stated as:

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

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. [9], which states that procedures are "distribution transformers". When running the procedure, we know that it has an input distribution satisfying Ψ. Each statement in the procedure will then change the input distribution in some way. For example, when assigning to a variable, we change the distribution of potential values for that variable. When running the whole procedure, we need to argue that the procedure transforms the input distribution in a way that makes Φ satisfiable.

2.2.6 *Probabilistic Relational Hoare Logic*

The pRHL logic allows us to reason about indistinguishability between two procedures w.r.t a pre- and postcondition. More formally, the pRHL logic allows us to determine if two procedures are perfectly indistinguishability w.r.t. to the given pre- and postcondition.

We recall from subsection 2.2.5 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 postcondition true.

In EasyCrypt we have the following notation for comparing two procedures:

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

Where Ψ is the precondition and Φ is the postcondition.

Alternatively, two procedures are indistinguishability if:

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

Informally, this can be understood as: The procedures P and Q running in respective memories m_1 and m_2 are indistinguishability w.r.t. to precondition Ψ and postcondition Φ, if both memories satisfy Ψ. 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. Alternatively, this is stated as:

$$\Phi \implies (A \iff B)$$

When dealing with pRHL statement, there are two types of deduction rules; one-sided or two-sided. The one-sided rules allow us to use the pHL deduction rules on one of the two programs we are comparing. 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 allow us to step the execution forward for one of the sides without reasoning about the other side. By doing this, we change all the term in the output distribution of the side on which the pHL deduction rule was applied.

The two-sided rules allow us to step forward both sides/procedures if they are both about to call 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 possibly non-terminate, our one-sided rules cannot be used to step the programs forward. By using the two-sided rules, this is not an issue, since if both procedures perform the same choice, then it does not matter what the choice was, or if it terminated, just that both procedures performed the same choice.

This allows us to step both procedures forward, while ensuring that both procedures made the same random choices, hence transforming their output distribution in similar ways.

2.2.7 *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:

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

Moreover, we adopt the EasyCrypt notation of **res** to signify the output value of a procedure.

We use the notation x^P to denote the value variable x w.r.t. procedure P. Likewise, we let x^Q denote the value of x when observing the run of procedure Q. To express that $x^P \sim x^Q$ we use the notation $=\{x\}$, Furthermore, we use the notation **Glob M** to express all the global variables that M has access to.

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

In this chapter, we introduce the different fields of cryptography used throughout this thesis. Most of these fields have intricate security definitions which depend on the context in which they are applied. We, therefore, do not try to give a complete introduction to the fields. Instead, we limit our introduction to the specific definitions relevant to the work of this thesis.

Notably, the section about multi-part computations (MPC) is intentionally left brief. The security definitions of an MPC protocol depends entirely on how much power the different participants of the protocol has. For the work of this thesis we only use the definitions of security where it assumed that all participants follow the protocol. This is commonly refereed to as semi-honest security.

Additionally, in cryptography, it is common practice to quantify every security definition over a "security parameter". This parameter exists to define how hard specific computation problems should be. Consider, for example, the discrete logarithm problem in a cyclic group. Naturally, the larger the group is, the harder the problem is to solve since there is a larger set of potential solutions. The security parameter would, in this case, be the size of the group. The existence of the security parameter usually offers a trade-off between security and efficiency. Considering the previous example for the discrete logarithm; any protocol operating on a group will have arithmetic operations with running time proportional to the size of the group. Therefore, the smaller the group, the faster the protocol is to run in practice.

The security parameter, however, is usually left implicit, which we have also done in the definitions below.

NOTATIONS While this thesis has been performed entirely in EasyCrypt, the code in this thesis will be given in a more pseudo-code style. For the most part, we will avoid EasyCrypt specific notation when writing procedures and solely focus on the tools EasyCrypt provides for reasoning about procedures.

Most notably we adopt the list indexing notation of $l[i]$ to mean the i'th index of the list l. Formally, this notation is not sound since it does not specify what will happen if the index does not exist. The is solved in EasyCrypt by declaring a default element should the indexing fail. We omit the default value from our code examples. Moreover, we define $x :: xs$ to mean that we prepend x to the list xs, $l_1 + +l_2$ means we concatenate two lists, and $\mathrm{rcons}\, x\, xs$ means we append x to the list xs.

Lastly, when referring to indistinguishability, we are referring the perfect indistinguishability unless explicitly 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.

A Zero-knowledge argument is a protocol run between a probabilistic polynomial time (PPT) prover P 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 he knows a pair (h,w) while only revealing h. At the end of the protocol, the verifier will then output **accept** or **reject** based on whether P convinced him or not. We require that a verifier following the protocol always output **accept** if P knew w and followed the protocol. This is known as *correctness*. Moreover, we require that a cheating adversary, who does not know w, can only make the verifier output **accept** with some negligible probability ε.

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). Formally, the *proof-of-knowledge* variant is proven by assuming the prover has infinite computational power.

Common amongst both variants is that they require that verifier learns no information, whatsoever, about w:

Definition 3.1.1 (Zero-knowledge from Damgaard [14])**.** Any proof-of-knowledge or argument with parties (P, V) is said to be zero-knowledge if for every PPT verifier V* there exists a simulator Sim$_{V^*}$ running in expected polynomial time which outputs conversations indistinguishable from a real conversation between (P, V*).

3.2 SIGMA PROTOCOLS

The following section introduce Σ-protocols, and their definition of security. The definitions given in the section is inspired by presentation of Σ-protocols by Damgaard [14]. Σ-protocols are two-party protocols on a three-move-form, with 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. This relation can also be expressed as a function, such that $(h,w) \in \mathrm{R} \iff \mathrm{R}_f\ h\ w = 1$. A Σ-protocol allows a prover who knows w to convince a verify, without revealing w to him.

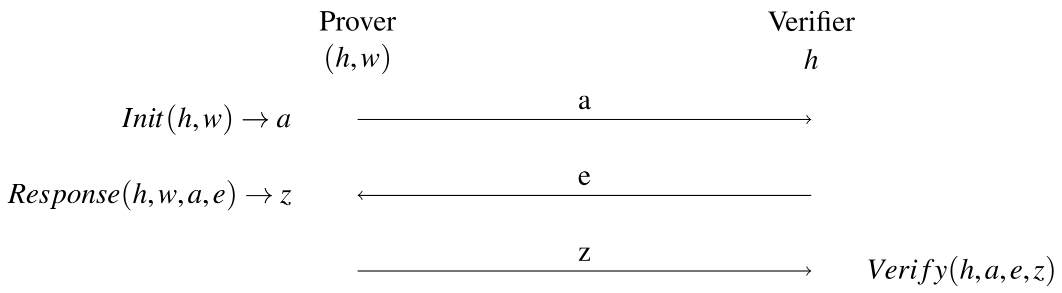


Figure 1: Σ-Protocol

A general overview of a Σ-Protocol can be seen in figure 1. Here we note that the protocol is on a three-move form since only three messages, (a,e,z), are sent between the prover and the verifier.

SECURITY A Σ-Protocol is said to be secure if it satisfy the following definitions.

Definition 3.2.1 (Completeness)**.** Assuming both P and V are honest i. e. following the protocol, then V will always **accept** at the end of the protocol.

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

Then S satisfies 2-special soundness given an efficient algorithm, called the "witness extractor" exists, that given the two accepting transcripts outputs a valid witness for the relation R.

The special soundness property is important for ensuring that a cheating prover cannot reliably convince the verifier. Given special soundness property, the cheating prover probability of convincing the verifier becomes negligible if the protocol is run multiple times. Special soundness implies that there can only exist 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 make his probability $\frac{1}{c}^n$, where n is the amount of runs.

The special soundness definition can also be generalised to s-Special Soundness. This definition requires that the witness can be constructed, given s accepting conversations.

Definition 3.2.3 (Special honest-verifier zero-knowledge)**.** 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 produces a transcript (a,e,z) indistinguishable from the transcript produced by S

Σ-Protocols have the convenient 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 while only having to prove that the protocol is zero-knowledge in the case of an 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 [14].

3.3 COMMITMENT SCHEMES

Commitment schemes are 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 [12]. A commitment scheme facilitates interaction between two parties, the committer and the verifier. The committer has a message, m, which he can use to generate a commitment and send it to the verifier. At a later point, C can then reveal the message to V, who can then use the commitment to verify the message C revealed is indeed the same message he used to generate the commitment.

Definition 3.3.1 (Commitment Schemes)**.** A commitment scheme 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 an 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 commitment schemes to be secure; it is required to satisfy three properties: **Correctness**, **Binding**, and **Hiding**.

Definition 3.3.2 (correctness)**.** A commitment scheme is said to be correct if the verification procedure will always accept a commitment made by an honest party, i. e.

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

Definition 3.3.3 (binding)**.** The binding property states that a party committing to a message cannot convince the verifier that he has committed to a message 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 is impossible to change the opening. *statistical binding* is achieved if there is a negligible probability of changing the opening and *computation binding* if producing a valid opening to a different message is equivalent to a hard computation problem.

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

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 with 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 communicate with each other over a secure channel freely. The parties then want to compute a public function, $f : (\text{input})^n \rightarrow \text{output}$, where each party contribute their input to the function and after computing the function every party will have the same output. However, none of the participant are allowed to learn about any of the inputs to the function, barring their own.

To achieve this, the parties jointly run an MPC protocol Φ_f. This protocol is defined in the term of rounds. In each round, each party P_i computes a deterministic function of all previously observed messages. The party then either sends the computed value to another party or broadcasts it to everyone. We define the collection of computed values and received values for party i as view_i.

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

Definition 3.4.1 (correctness)**.** An 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)**.** An MPC protocol Φ_f is said to have d-privacy if d parties colluding cannot obtain any information about the private inputs of the remaining $n-d$ parties.

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

4 FORMALISING COMMITMENT SCHEMES

In this section, we formalise commitment schemes and their security definitions. Our formalisation of commitment schemes provides two variants; key-based and key-less commitment schemes. The key-based formalisation requires the two parties to share a key. The other variant 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. Separating the two types, however, makes our formalisation 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 functions and procedures seen in Listing 1.

Here verification of commitments and key pairs are modelled as function, since we assume these function to be deterministic and lossless. When verifying a commitment, for example, there should be no need to sample additional randomness. A simple deterministic function on the commitment and its opening should suffice. Moreover, if the verification algorithms cannot terminate within a reasonable amount of time, then it is probably not worth studying the commitment scheme further.

```
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 1: Key-Based commitment specification

The committer is modelled as a module with two procedures; key_gen for generating key pairs and commit for committing to messages. This allows the Committer to keep state and make random choice, while verify can be kept as a deterministic function given access to the public key.

By separating the verification functions from the committers procedures we get a formalisation closer to the real world, where the verifiers functions should not be able to read/alter any of the state of the committer. This could alternatively have been modelled with the verifier being a separate module. However, if verify is expressed as a procedure then we allow verify to read/alter state. By introducing state, we unnecessarily complicate reasoning about verification of multiple messages. Consider the case where the verifier is given two commitment and two openings; if verify is allowed to alter state, then the order of which we verify the commitments matter. It is, therefore, not possible to change to order of verifying commitments, unless it is explicitly proven that verify does not access the state of the module. Implementing verify as a function alleviates the problem of reasoning about state. This is in contrast to previous work by Metere and Dong [18], which implemented the functionality of the verifier as procedures in the same module as the committers functionality.

We define a commitment scheme to be an implementation of the functions and procedures in listing 1.

4.2 KEY-LESS COMMITMENT SCHEMES

Furthermore, we formalise a variant of commitment schemes that is key-less. This is formalised separately from the key-based commitment schemes, since the change in function/procedure signatures makes it incompatible with the key-based formalisation. They could 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 reason about the sampling and distribution of the keys. Ideally, it should be proven that the two formalisation are compatible w.r.t. security, and one can be used in place of the other, but this is 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 1 for the key-based commitment schemes except that 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.

Consequently, 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 commitment schemes we use the same definitions of security, which is based on the work of Metere and Dong [18].

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 in Listing 2

```
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;
  }
}.
```

Listing 2: Correctness Game

Definition 4.3.2 (Hiding)**.** A Commitment scheme C can have the 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 the following degrees of binding: *perfect binding:* $\forall Adv.\Pr[BindingGame(C,Adv).main() = true] = 0$ *computational binding:* $\forall Adv.\Pr[BindingGame(C,Adv).main() = true] = \varepsilon$

The adversary Adv and the procedure BindingGame is defined in Listing 3.

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 [9].

4.4 ALTERNATIVE DEFINITIONS OF SECURITY

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

```
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');
  }
}.
```

Listing 3: Binding Game

Lemma 4.4.1 (Alternative correctness)**.**

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

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 semantically equivalent.

We are then left with showing:

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

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 in EasyCrypt since pRHL statement be used to prove equivalence of calls to sub-procedures in other proofs.

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 to reproduce the results of Metere and Dong [18]. 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 $(\mathbb{G}, q, g)$, 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 $h \in_R \mathbb{G}$ and send it to him
- He then samples a random opening $d \in_R \mathbb{Z}_q$ 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 randomness, (m', d') to R, which 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 performs a deterministic computations. This fits perfectly within our formalisation of the Receiver. We, therefore, instantiate our commitment scheme framework as seen in listing 4.

```
clone export Commitment as Com with
  type public_key <- group
  type secret_key <- group
  type commitment <- group
  type message    <- F.t  (* Finite field element *)
  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) = $g^r \cdot pk^m = c$,

module Pedersen : Committer = {
  proc key_gen() = {
    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 4: 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 the uniform distributions of messages and randomness by the provided distributions of field elements.

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

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 the message 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) $\forall Adv.\Pr[\text{HidingGame}(\text{Pedersen, Adv}).main = true] = \Pr[\text{HidingIdeal}(\text{Adv}).main = true]$. We prove that for any choice of b the two procedures are indistinguishable. We start by proving indistinguishability with $b = 0$. To prove this we have to prove that $g^r \sim g^{r'} \cdot \text{sk}^m$ Here we use EasyCrypt's coupling functionality to prove that $r \sim r' \cdot \text{sk}^m$ 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) $\forall Adv.\Pr[\text{HidingIdeal}(\text{Adv}).main = true] = \frac{1}{2}$

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

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 an 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:

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

Fist we show that if DLogPedersen if given a 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 $\mathrm{EasyCrypt}$'s automation tools.

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

5 FORMALISING Σ-PROTOCOLS

In this chapter, we formalise Σ-Protocols along with different constructions based on Σ-Protocols. Our formalisation is driven by the definitions given by Damgaard [14] and already existing formalisation of Σ-Protocols in the EasyCrypt source code [2]. Our work improves on previous work by providing more generalised definitions, namely s-special soundness instead of the usual 2-special soundness definition given. Additionally, we also provide an alternative definition of completeness. Moreover, we reproduce the work of compound Σ-Protocols by Butler et al. [12] by formalising their results in EasyCrypt. We then introduce the Fiat-Shamir transformation, which can turn any Σ-Protocol into a non-interactive zero-knowledge protocol in the random oracle model. We then explore how this transformation applies to our formalisation. Last, we show that our formalisation can be used to prove the security of Schnorr's Σ-Protocol.

5.1 DEFINING Σ-PROTOCOLS

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

$$\begin{aligned}\textbf{type: } & \text{statement}\\ & \text{witness}\\ & \text{message}\\ & \text{challenge}\\ & \text{response}\end{aligned}$$

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 witness to true/false $R : (\text{statement} \times \text{witness}) \to \{0,1\}$. Moreover, we fix a lossless, full, and uniform distribution over challenges space. This distribution is used to model an honest verifier which will always generate a uniformly random challenge.

We define a Σ-protocol to be a series of probabilistic procedures, based on the outline of Σ-Protocols given in Figure 1. The procedures can be seen in Listing 5.

```
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 5: 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 6: Completeness game for Σ-Protocols

Here all procedures are defined in the same module. This allows the Verifier procedure to access the global state of the prover, which could lead to invalid proofs of security. It is, therefore, paramount to implement verify such that it never accesses the 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, however, it is easier to quantify over one module containing all relevant procedures than quantifying over a prover and a verifier module and then reasoning about the two modules being part of the same Σ-Protocol. Ultimately, we decided on defining everything within the same module.

An instantiation of a Σ-Protocol is then an implementation of the procedures in listing 6.

SECURITY 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 listing 6 outputs 1 with probability 1, i. e.

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

One problem with definition 5.1.1 is that quantification over challenges is implicitly done when sampling from the random distribution of challenges. This means 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:

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

Where the procedure "Completeness(S).special" is defined in listing 7.

However, since the alternative procedure no longer samples from a random distribution, it is not possible to prove equivalence between the two procedure. Instead, we show the alternative definition still captures what it means for a protocol to be complete:

Lemma 5.1.3.

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

```
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;
}
```

Listing 7: Special Completeness

Proof. 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 inlining the procedures and moving the sampling to the first line of each program. This will make the two programs equivalent.

Next, we prove the lemma by showing:

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

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

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

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

Definition 5.1.4 (Special Soundness)**.** A Σ-Protocol S has s-special soundness if given a list of challenges c and a list of responses z with size c = size $z = s$, it holds that:

$$\begin{aligned}&\forall(i \neq j).c[i] \neq c[j]\\ \wedge&\forall(i \in [1,\ldots,s]).\Pr[S.verify(h,a,c[i],z[i])] = 1\\ &\implies \Pr[SpecialSoundness(\text{ANDProtocol}(S)).main(h,a,c,z)] = 1\end{aligned}$$

With 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, which can be seen in figure 2. We then say a Σ-Protocol S is special honest verifier zero-knowledge if:

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

Definition 5.1.6. S is said to be a secure Σ-Protocol if it implements the procedures in figure 5 and satisfies the definitions of completeness, special soundness, and special honest-verifier zero-knowledge.

```
module SpecialSoundness(S : SProtocol) = {
  proc main(h : statement, a : message, c : challenge list, z :
   response list) : bool = {
    w = S.witness_extractor(h, m, c, z);

    valid = true;

    while (c <> []) {
      c' = c[0];
      z' = z[0];
      valid = valid /\ S.verify(h, m, c', z');
      c = behead c;
      z = behead z;
    }

    return valid /\ R h (oget w);
}.
```

Listing 8: 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

We note that our definition of special honest-verifier zero-knowledge is not expressive enough for all use cases. Since we are using EasyCrypt's pRHL to distinguish the two procedures, our definition only captures perfect zero-knowledge. If we wanted a definition capable of expressing computational zero-knowledge, we would have to use an adversary to compare the two procedures. We opted to restrict SHVZK to perfect zero-knowledge.

5.2 COMPOUND PROTOCOLS

Given our formalisation of Σ-Protocols we now show that our formalisation composes in various ways. More specifically, we show that it is possible to prove knowledge of relations compounded by the logical operators "AND" and "OR".

Formalisations of compound Σ-Protocols already exists in other proof assistants [8, 12], which we will also use as a basis for our EasyCrypt formalisation. Primarily, we aim to reproduce the results of Butler et al. [12]. By reproducing formalisations within a new proof assistant, we can gain valuable insight into how EasyCrypt compares to other proof assistants while reflecting on how to improve previous work.

HIGHER-ORDER INSTANCES OF THEORIES Before formalising the constructions we remark on how to express modules depending on other modules in EasyCrypt. We note that this section is not important for the proofs presented in this section, but outlines a workaround need to express this type of construction in EasyCrypt.

In EasyCrypt modules can be parametrised by abstract modules, i. e. we can define the module of the AND construction as:

```
module ANDProtocol(S_1 : SProtocol, S_2 : SProtocol) : SProtocol =
    { ... }
```

The problem with this, however, is our formalisation of Σ-Protocols is more than just a module; it is a theory abstracted over the types of Σ-Protocols and a fixed distribution of challenges.

The above notation of parametrising modules does encapsulate the functionality of the Σ-Protocol theory and only exposes the procedures of the modules. This in turn, means that the above code listing will not type check, since we have no way of knowing that the types of S_1 and S_2 will be.

To overcome this limitation, we let the AND construction be an abstract theory quantified over the types of two Σ-Protocols, and instantiations with those types. This mean that the types of S_1 and S_2 will be fixed at the time when declaring the module ANDProtocol.

The code for this can be seen in listing 9. Similarly, the construction of OR follows the same outline.

5.2.1 *AND*

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 is itself 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 following the steps of the previous section. We then overload S_1 and S_2 to also mean any Σ-Protocol with types matching S_1 or S_2. Therefore, when we write $P_i : S_i$ we are referring to any protocol P_i, which satisfy the types of S_i.

We then define the procedures of the AND construction as seen in listing 10. The AND construction works by running both protocols in parallel. The first message, therefore, becomes the product of the messages produced by the underlying protocols. The responses are similarly created.

SECURITY Given that the AND constructions is a Σ-Protocols we 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 $\mathrm{ANDProtocol}(P_1, P_2)$ satisfy completeness definition 5.1.1

Proof. By inlining the procedures of $\mathrm{ANDProtocol}(P_1, P_2)$ in $\mathrm{Completeness}(\mathrm{ANDProtocol}).\mathrm{special}$ we see that it is equivalent first running the completeness game of P_1 and then the completeness game of P_2, i. e. $\mathrm{Completeness}(P_1).\mathrm{special};\ \mathrm{Completeness}(P_2).\mathrm{special}$. We then concluded that the special definition of completeness must return true with probability one by our assumption of completeness of P_1 and P_2.

We need to use the special definition of completeness since the challenge e is given by a Verifier running the AND construction. Were we to use the original definition of completeness, then the AND construction would need to randomly sample two challenges to prove equivalence with the binding games, which is not the case.

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

```
theory ANDProtocol.
  type statement1. (* type of statement in protocol 1 *)
  type statement2. (* type of statement in protocol 2 *)
  type witness1.
  type witness2.
  type message1.
  type message2.
  type challenge.
  type response1.
  type response2.

  (* define the relations *)
  op R1 (x : statement1, w : witness1) : bool.
  op R2 (x : statement2, w : witness2) : bool.
  op R = fun x w => (R1 (fst x) (fst w)) /\ (R2 (snd x) (snd w))
   .

  (* fix the challenge distribution *)
  op dchallenge : {challenge distr | is_lossless dchallenge /\
   is_funiform dchallenge} as dchallenge_llfuni.

clone SigmaProtocols as S1 with
  type statement <- statement1,
  type witness <- witness1,
  type message <- message1,
  type challenge <- challenge,
  type response <- response1,
  op R = R1,
  op dchallenge = dchallenge

(* Define AND construction as Sigma-Protocol with the types of
    the underlying protocols *)
clone SigmaProtocols as S2 with
  type statement <- statement2,
  type witness <- witness2,
  type message <- message2,
  type challenge <- challenge,
  type response <- response2,
  op R = R2,
  op dchallenge = dchallenge

clone export SigmaProtocols as Sigma with
  type statement <- (statement1 * statement2),
  type message   <- (message1 * message2),
  type witness   <- (witness1 * witness2),
  type response  <- (response1  * response2),

  op R = R,
  op dchallenge = dchallenge

module ANDProtocol (P1 : S1.SProtocol, P2 : S2.SProtocol) :
    Sigma.SProtocol = { ... }
```

Listing 9: AND construction theory

```
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 10: AND construction

```
proc witness_extractor(h : statement, a : message, e : challenge
    list, z : response list) = {
  (h1, h2) = h;
  (a1, a2) = a;
  e = e[0];
  e' = e[1];
  (z1, z2) = z[0];
  (z1', z2') = z[1];
  w1 = P1.witness_extractor(h1, a1, [e;e'], [z1;z1']);
  w2 = P2.witness_extractor(h2, a2, [e;e'], [z2;z2']);

  return Some(oget w1, oget w2);
}
```

Listing 11: Witness extractor for AND construction

```
proc simulator(h : statement, e : challenge) : message *
   response = {
  (h1, h2) = h;

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

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

Listing 12: Simulator for the AND construction

Lemma 5.2.2 (AND special soundness)**.** Given secure Σ-Protocols P1 and P2 the AND-Protocol(P1, P2) satisfy definition 5.1.4 with $s = 2$.

The witness extractor is defined in listing 11

Proof. We start by showing that

$$\begin{aligned} & \text{AND.verify}((h_1,h_2),(a_1,a_2),e,(z_1,z_2)) && (7)\\ \iff & \text{P1.verify}(h_1,a_1,e,z_1) \wedge \text{P2.verify}(h_2,a_2,e,z_2) && (8)\end{aligned}$$

Which follows directly from the definition of AND.verify.

Then, since relation of AND requires a witness for both R_1 and R_2, we can use the 2-special soundness property of P_1 and P_2 to conclude the proof. □

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

The simulator for the AND construction is given in listing 12.

Proof. Since the simulator for AND is running both simulator for P1 and P2, we use 7 to apply the SHVZK property of P1 and P2. From this, we can conclude that the transcript of the simulator is indistinguishable from the transcript of running honest versions of P1 and P2. By correctness of AND the proof is then complete. □

5.2.2 *OR*

For the OR construction we use the definition by Damgaard [14], 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 OR construction to be a Σ-Protocol proving knowledge of the relation

$$R((h_1, h_2), w) = R_1(h_1, w) \vee R_2(h_2, w)$$

.

However, this relation makes it impossible to realise our security definitions [12]. If the public statement h_i, for which there is no witness, is not in the domain of statements with a possible witness making the relation true, then it is possible for the verifier to guess which of the two underlying relations held. To fix this we use the alternative relation:

$$\begin{aligned} R((h_1, h_2), w) = & R_1(h_1, w) \wedge h_2 \in \textbf{Domain } R_2 \\ & \vee R_2(h_2, w) \wedge h_1 \in \textbf{Domain } R_1 \end{aligned}$$

The main idea behind the OR construction, is that by 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 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 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.

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

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

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

We then define the OR construction as a Σ-Protocol like in section 5.2.1. The procedures can be seen in listing 13.

SECURITY Given the OR constructions is a Σ-Protocols we need to prove the security definitions given in section 5.1 with regards to the module $\mathrm{ORProtocol}$

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 into the P1 completeness game. We then need to prove that $\mathrm{S2.verify}$ output accept on the transcript generated by $\mathrm{S2.simulator}$, which is true by the assumption of SHVZK of P2. The proof when R2 $h2$ w holds follows similarly. □

```
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 13: OR construction

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 pre- and post-condition given by definition 5.1.5.

The simulator for the OR construction is given is listing 5.2.5.

```
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 further split the proof into two steps:
1) we show that $e1^{real} \sim e1^{ideal}$ and $e2^{real} \sim e2^{ideal}$. This follows trivially since we assume both procedures make the same random choices, and since the order in which the challenges are sampled is the same, then 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 to conclude that the procedures are indistinguishable, 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. More specially, $e1$ is sampled at random and $e2 = s \oplus e1$ in this case. To prove this we instead use EasyCrypt's coupling functionality and 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 $s = 2$ and 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 Some(oget 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 being that Σ-Protocols only guarantee zero-knowledge in the presence of a honest verifier. This is stated by the special honest-verifier zero-knowledge property. However, if we can alter the protocol slightly 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, thus making it non-interactive. The verifier is replaced by an random oracle, which generates a uniformly random challenge based on the first message of the prover. The random oracle, therefore, works exactly like an honest verifier in the interactive protocol. However, since the prover is responsible for obtaining the challenge from random oracle, and there is no interaction with the verifier, he is free to call the oracle as many times as he wants. The only limitations is that the time complexity of the protocol must still be polynomial. Therefore, the prover can only make polynomially many calls to the oracle. The problem with allowing the prover to call the oracle multiple times is he can keep sampling new challenges from the oracle until he gets a challenge that he can answer.

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 is used to determine the random choices made by the oracle. Moreover, it exposes the procedure sample which samples new challenges based on messages.

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 RandomOracle : Oracle = {
  global variable : h = (message ↦ challenge)

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

  proc sample (m : message) : challenge = {
    if ($m \notin \mathrm{Domain}(h)$) {
      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 difference is that underlying protocol waits for the verifier to sample a challenge for him. Since an honest verifier will never fail to send a challenge (i. e. he is lossless) and the challenge is always 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. Here, the simulator is allowed to dictate the random choices made by the oracle in 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 forking lemma, which depends on rewinding and is still an open research topic to formalise within EasyCrypt [9].

5.4 CONCRETE INSTANTIATION: SCHNORR PROTOCOL

To show the workability of our formalisation, we show that it can be used to instantiate Schnorr's protocol. Schnorr's protocol is run between a Prover P and a Verifier V. Both parties, before running the protocol, agree 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 P wants to prove knowledge of w to V he starts by constructing a message $a = g^r$ for some random value r. The Verifier will generate a random challenge, e, which is a bit-string of some arbitrary length. 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 our formalisation of Σ-Protocol procedures. We then define the appropriate types and instantiate the protocol using our Σ-Protocol formalisation, which can be seen in listing 14.

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

To prove that the protocol is secure, we show that it satisfies the security definitions from section 5.1.

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

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

1. The procedure always terminates

2. The output of the procedure is always true.

1) Since all instructions of the game, bar the random sampling, are arithmetic operations they can never fail and the random sampling was proven to be lossless when instantiating the Σ-Protocol formalisation we can conclude that the procedure always terminates.

```
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 $r + e \cdot w$;
  }

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

Listing 14: Schnorr instantiation

2) After running all sub-procedures of the correctness game, the output of the procedure is

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

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

Lemma 5.4.2 (Schnorr 2-special 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]);
}
```

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 iterating through 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:

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

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)**.**

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

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

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

To prove SHVZK be must the prove output indistinguishability of the procedures in figure 3. To prove this we use EasyCrypt coupling functionality to show that $r^{real} \sim z^{ideal} - e \cdot w^{real}$ and that $z^{ideal} \sim r^{real} + e \cdot w^{real}$. This follows from the distribution being full and uniform,

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

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

Figure 3: SHVZK procedures

and the group is closed under addition and multiplication. All these facts follow from the cyclic group theory in EasyCrypt. We then use this to show output indistinguishability:

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

Which can easily be proven by EasyCrypt's automation tools. □

6 GENERALISED ZERO-KNOWLEDGE PROTOCOLS

In this chapter, we introduce generalised zero-knowledge protocols. In the previous section, we have 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).

However, since the need for zero-knowledge is widespread, it is unlikely that a Σ-Protocol for proving knowledge of a discrete logarithm suffices. Consider the example of an authentication system, where each user has a certificate consisting of his name, income, and birthday. This certificate also has some additional information to verify the certificate. A user of this system would then like to prove that he is above the age of 18. However, the user is not willing to reveal his birthday. To alleviate this, we could define a Σ protocol for proving knowledge of the birthday matching a valid certificate with the name and income. We could then image another user being interesting in proving their wealth is above 5000, but not their precise income. Here, we have the same problem as before, but with a slightly different relation.

To solve this problem, we could pre-determine all the relations for which we need zero-knowledge proofs. However, another approach is that of generalised zero-knowledge protocols. A generalised zero-knowledge protocol is, like the name suggests, a protocol able to prove zero-knowledge for an entire class of relations. One example of such a class of relations is every relation that can be described as the pre-image of a group homomorphism, i. e. that the user knows the input to some function which results in a specific output.

To explore the field of generalised zero-knowledge protocols, we introduce ZKBoo by Giacomelli et al. [15]. ZKBoo is a Σ-Protocol for any relation expressed as the pre-image of a group homomorphism.

6.1 ZKBOO

In this section we introduce ZKBoo, a generalised Σ-Protocol for relations of the form:

$$(x,y) \in R \iff f(x) = y$$

Where y is the public input, x is the witness and f is any function defined as a circuit over a finite group.

The principle idea of this protocol is based on a technique called "MPC in the head". The idea of "MPC in the head" is to run a secure n party MPC protocol for computing f, but with every party being locally simulated, rather than run as part of the protocol. By running every party locally, we also remove any communication overhead that the MPC protocol might incur. Moreover, since the protocol is run locally, we can assume every party of the MPC protocol to be semi-honest. If the parties where not semi-honest, then it might be easier to extract the secret input of the parties, which is not in the interest of the person running the protocol. These two facts contribute to the "MPC in the head" approach being significantly more efficient than a normal MPC protocol.

We then distribute x into shares $x_1, \dots x_n$ for each of the parties. x should be distributed in such a way, that we can reconstruct x from the shares. This allows us to compute $f(x_1, \dots x_n) = y$ with the MPC protocol. In a sense, this corresponds to splitting the computation into n branches. Consequently, if the branches are then later recombined we get the actual evaluation of f. Moreover, if we only reveal d of the computational branches, then the inputs cannot be recovered by the privacy property of the MPC protocol.

In the following section, we introduce the concept of the $(2,3)$-Decomposition, which is an "MPC in the head" protocol for splitting the evaluation of a circuit into three branches with 2-privacy and correctness. We then introduce the actual ZKBoo protocol and describe how the views of an MPC protocol can be used to prove knowledge of x.

6.1.1 *(2,3)-Function Decomposition*

The (2,3)-Function decomposition is a general technique for computing the output of a circuit $f : X \to Y$ 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 an MPC protocol. The $(2,3)$-Decomposition is essentially a 3 party MPC protocol with 2-privacy that has been altered such that every party is run locally. Throughout this section, we will 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 splitting the input into an input share for each party, where the original input can be obtained if all three input shares are acquired. Each party then evaluates all the gates of the circuit to a value, as described by the MPC protocol. Here, we mimic the communication of the parties by letting them access each other's views. As a general rule, we only allow party P_i to communicate with party $P_{i+1 \mod 3}$. The view of a party is a list of values that the party has computed so far. The view of party P_i is referred to as w_i. For the rest of this chapter, we will omit the "mod 3" from the indexing for the sake of brevity. 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 6.1.1. In its most general form the decomposition is a collection of functions:

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

Share is a procedure for computing the three input shares based on an input to f. Conceptually, Share takes a value x and splits into n shares, called $x_1, \dots, x_n$. The n shares can the be used to reconstruct x by applying the inverse function of share: $\mathrm{Share}^{-1}(x_1, \dots, x_n) = x$. 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.

Last, we have $\mathrm{Update}(\mathrm{w}_i^j, \mathrm{w}_{i+1}^j, \mathrm{k}_i, \mathrm{k}_{i+1}) = w_i^{j+1}$ 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.

SECURITY Based on the security definitions from MPC (Section 3.4) we can then define the two fundamental properties from [15] for showing security of our (2,3)-Function decomposition, namely correctness and privacy.

Definition 6.1.2 (Correctness)**.** A (2,3)-decomposition $\mathscr{D}_f$ is correct if $\forall x \in X, \Pr[f(x) = \mathscr{D}_f(x)] = 1$.

Definition 6.1.3 (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

6.1.2 *(2,3)-Function Decomposition for Arithmetic circuits*

Based on the general description of the (2,3)-Decomposition from the previous section we can now define a concrete (2,3)-Decomposition of arithmetic circuits as in Giacomelli et al. [15].

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. Moreover, we 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:

- Share(x, k_1, k_2, k_3): Sample random values x_1, x_2, x_3 such that $x = x_1 + x_2 + x_3$
- Output$(w_i) = y_i$: return the output share of party i.
- 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.
  $$w_i[j] = \begin{cases} w_i[a] + \alpha & \text{if } i = 1 \\ w_i[a] & \text{else} \end{cases}$$
  - Multiplication by constant: where a is the input wire to the gate and $alpha$ is the constant
  $$w_i[j] = w_i[a] \cdot \alpha$$
  - Binary addition: where a, b are the input wires.
  $$w_i[j] = w_i[a] + w_i[b]$$
  - Binary multiplication: where a, b are the input wires.
  $$w_i[j] = w_i[a] \cdot w_i[b] + w_{i+1}[a] \cdot w_i[b] + w_i[a] \cdot w_{i+1}[b] + R_i(j) - R_{i+1}(j)$$
  Where $R_i(j)$ 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 value of the share supplied by the other party. If the random values were not added then it would be easy to deduce what the share of P_{i+1} is, given access to the view of party P_i.

6.1.3 *ZKBoo*

Based on the (2,3)-Decomposition we are now ready to ZKBoo.

The protocol proceeds as follows:

- The prover obtains the circuit representation C_f of f and uses $\mathscr{D}_{C_f}$ 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 the verifier gets access to all three views and $\mathscr{D}_{C_f}$ is correct, then we would be able to extract the witness from 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 $\mathscr{D}_{C_f}$ 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 option for the prover is if he changes some of the shares in one of the views, thereby coercing the output to the value he wants. By doing so, one of the views will deviate from the procedures of $\mathscr{D}_f$, and the prove can quickly detect this if he picked the challenge exposing the view the prover coerced. If not, then the prover has successfully cheated him. It is, therefore, necessary to run the protocol multiple times.

To prove that the above claims hold and that the ZKBoo protocol is secure, we will, in the following chapter, formalise ZKBoo by utilising our formalisation of Σ-Protocols and Commitment Schemes.

7 FORMALISING ZKBOO

In this chapter, we formalise the ZKBoo protocol along with the security proofs by Giacomelli et al. [15]. To formalise this, we utilise our formalisations of Σ-Protocols and commitment schemes. The definitions and proofs formalised in the chapter all come from Giacomelli et al. [15]. In order to formalise these, we have had to make definitions more precise and prove certain predicates about the different procedures of the protocol in order to prove it secure. These changes have been necessary since crucial assumptions have been kept implicit in the original paper. By formalising the proofs of security, we make these assumptions explicit, which can help improve implementation-level security by exposing edge-cases where the security definitions could break down.

The goal of formalising ZKBoo is two-fold. First, we show that our previous formalisations are indeed applicable to more extensive protocols. Second, we aim to gather insight into the security of the ZKBoo protocol itself, and how formal verification can help us find pitfalls in informal proofs.

To formalise the work of ZKBoo, we first start by developing a formalisation of arithmetic circuits within EasyCrypt. This formalisation allows us the reason about evaluating circuits to a value while also enabling us to reason about the structure of arithmetic circuits. Next, in section 7.2, we formalise the (2,3)-Decomposition of arithmetic circuits as defined in section 7.1. Ultimately, we use the formalisation of arithmetic circuits and their (2,3)-Decomposition to instantiate ZKBoo as a Σ-Protocol and then prove it secure.

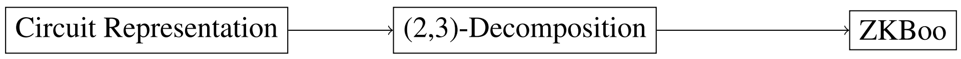


Figure 4: Outline of ZKBoo formalisation

PRELIMINARY NOTATION Throughout this chapter, we use the letter e to denote a challenge expressed as an integer in $\{1,2,3\}$. Moreover, we define the views after 3 to be 1. Logically the challenges could be denoted $\mathbb{Z}_3$, but we chose to follow the notation of the original paper here.

7.1 FORMALISING ARITHMETIC CIRCUITS

Before we can reason about the decomposition of an arithmetic circuit, we need a clear definition of what an arithmetic circuit is, and how we can compute values from it. Primarily, we recall the definition of circuits as graphs which is also used in the original paper by Giacomelli et al. [15] and discuss how to evaluate circuits programmatically. Lastly, we introduce an alternative representation of arithmetic circuits and provide several definitions, enabling us to reason about the structure of the circuit and its evaluation.

7.1.1 *Representing an arithmetic circuit*

An arithmetic circuit in its general form expresses a function f over some arbitrary finite field $\mathbb{Z}_q$, where $f : \mathbb{Z}_q \to \mathbb{Z}_q$

To express arbitrary (arithmetic) computations in a 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 f, which only depends on the aforementioned gate types to perform computations.

We now recall the graph representation of a circuit:

Definition 7.1.1 (Arithmetic Circuit)**.** An Arithmetic circuit is a directed acyclic graph C = (W, G) where W is the wires between the gates and G is the set of gates within the circuit. Then, we let $in \in G$ be the first gate of the circuit, i. e. the input and $o \in G$ be the final gate of the circuit, for which there must exist a path from *in* to o in W. Specifically, o is a gate with only in-going wires and no out-going wire. The value of the circuit is then obtained by computed the value of o.

Finally, for all gates, $g \in G$ there must exist a path in W from *in* to o going through g. If this was not the case, then the gate could be removed from the graph without changing the semantic meaning of the circuit.

To define the evaluation of the circuit, we would then need to compute the values of the in-going wires of o, but this requires all other wires in the circuit to be computed first. Consequently, the value of the out-going wires of a gate can only be computed if all in-going wires of the gate have already been assigned a value. It is clear from this that we need to define an evaluation order of the circuit, such that we only try to compute the value of a gate if we know that all in-going wires have been assigned a value.

To define the order of evaluation, we follow the work of [5] and introducing an alternative representation of Arithmetic circuits. However, before doing so, we first introduce linear orderings and encoded gates:

LINEAR ORDERING A linear ordering O is a function that applied to G assigns an unique index to each gate in G. One example of such a linear ordering is the breadth-first search (BFS) of a graph. Here each gate in the graph is labelled according to when the BFS visited the gate/node. This labelling would start at the input gate and end at the output gate.

A particularly useful property of the linear ordering induced by a BFS, is that a gate can only be visited if the BFS has already visited all nodes with wires going to that gate. This ordering ensures that, for any given node, with the index i will only depend on computations of gates with an 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 input gate does not perform any computations, except for adding the input value to the graph, we exclude it from the list representation and shift every index one down.

ENCODED GATES An encoded gate is a type, where the type defines the operation which the gate computes along with a tuple (l,r) where l is the index node corresponding to the left input wire and r is for the right input wire. In the case of unary gates like ADDC and MULTC the tuple is (c,l) where l is the input wire, and c is the constant used in the

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

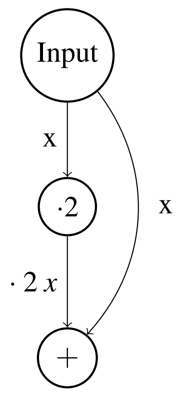
Listing 15: Type declaration of gates

computation. Here, the tuple (l,r) is the wires of W encoded directly into the gates. The type definition can been seen in Listing 15.

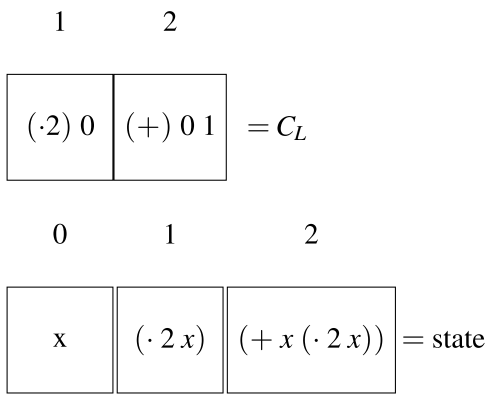
Definition 7.1.2 (List representation of arithmetic circuits)**.** Given an arithmetic circuit C (Theorem 7.1.1) we define the list representation of C by computing a linear ordering, O, of G, which assigns each gate in G an unique index. We then let the list representation C_L be defined as:

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

Where $Enc : gate \rightarrow W \rightarrow$ encoded gate, is a function taking as input a gate and the wires of the circuit and produces an encoded gate. Encoded gates contain type information about the 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. e. g. If we have an addition gate with wires coming from the gate with index i and a gate with index j we express this as an encoded gate $MULT(i,j)$.



(a) Graph representation of circuit



(b) List representation of circuit

One important aspect of the list representation of circuits is that it gives us a well-defined evaluation order. For specific linear orderings we can ensure that we are not trying to compute the value of a gate before all its inputs wires has been assigned a value. To capture this notion of a valid evaluation order, we give the following definition:

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 gate corresponding to the input wires of C[i] have index j where $0 \leq j < i$.

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

Listing 16: Circuit evaluation function

It is then possible to define the semantic meaning of the list representation of a circuit by defining an evaluation function, which can be seen in Listing 16. The evaluation is broken into two parts: First, we have a function for evaluating one gate to an intermediate value. Second, we have a procedure for evaluating the entire circuits which call the former. To evaluate a single gate, we first need to determine the type of the gate. 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 16.

Then, by the validity of the circuit, we know that if computing index i of the circuit, then indices $[0 \dots i-1]$ have already been computed. Performing the computation expressed by the gate 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.

To maintain the list of computed gates, we define the "state", which at index $i+1$ will store the value for performing gate evaluation of the gate at index i

For example, when computing the gate $MULT(i, j)$, we simply look up the valued of index i and j in "state" and multiply them together. When every single gate of the circuit has then been computed and saved in "state", then the output of the circuit will be in the last entry of the state.

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:

$$\begin{aligned}\text{state}[0] &= \text{input value}\\ \text{state}[i > 0] &= \text{eval\_gate } C_L[i-1] \; state[i-1]\end{aligned}$$

Here we recall that the 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

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

This representation can then be extended to allow for multiple inputs. If we want the circuit to have n input gates, then we reserve the first n entries of the state to be inputs. Consequently, we then shift every index in our encoded gates by n. Moreover, if want to support multiple output we need to fix the ordering such that the m output gates are always computed last. By computing the output gates last we are ensured that the last m entries of the state will be the outputs.

We then have that any valid circuit c can be computed to a value y as $\mathrm{eval\_circuit}(\mathrm{c},\ [\text{input}]) = \mathrm{y}$. This can also be stated as a probabilistic procedure as $\Pr[\mathrm{eval\_circuit}(\mathrm{c},\ [\text{input}]) = y] = 1$.

To reason about functions and procedures based on the same function we have the following lemma:

Lemma 7.1.5 (Function/Procedure relation)**.** $\forall$ f, inputs, output: f(inputs) = output $\iff$ $\Pr[f(inputs) = output] = 1$.

Proof.

- "$\Rightarrow$": trivial
- "$\Leftarrow$": We split the prove into two parts:

  $output = f(inputs)$: trivial.

  $output \neq f(inputs)$: Here we prove that

  $$output \neq f(inputs) \implies \Pr[f(inputs) = output] = 0.$$

  Which is trivially true. We then use this to derive a contradiction in our assumptions, thus completing the proof.

□

7.2 (2,3) DECOMPOSITION OF CIRCUITS

In this section, we formalise the (2,3)-Decomposition of arithmetic circuits based on the description given in section 6.1.2.

In its most general form, we define the decomposition as a procedure compute, which takes an input three views and three random tapes along with the circuit it needs to compute. The results of running compute is then three new views and random tapes, which express the decomposed computation. More specifically, the decomposition works by incrementally evaluating a gate based on previously computes views, which yields new shares that can be appended to the view. We then repeat the process of evaluating a single gate based on the view of evaluating the previous gate until all gates have been computed. The arithmetic decomposition, however, requires that we can split the witness into three uniformly random shares. To do this, we fix a distribution dinput which is full and uniformly distributed over the shares; in this case, field elements. Moreover, we note that the function Update is the one defined in section 6.1.2, except that it only returns the newest share and not the entire view. Additionally, Update takes an extra parameter p, which is used to decide which party is computing the share. This is use for the ADD gate to determine if $p = 1$. Moreover, when calling Update on gate g, then it will use randomness $k_i[j]$ where j is the index of g in the circuit.

The reconstructed output of the decomposition can then be defined as summing the output share from each view that has been computed by the aforementioned procedure.

```
proc compute(c : circuit , w1 w2 w3 : view , k1 k2 k3 :
  random_tape) = {
   while (c <> []) {
     g = c[0];
     r1 <$ dinput;
     r2 <$ dinput;
     r3 <$ dinput;
     k1 = (rcons k1 r1);
     k2 = (rcons k2 r2);
     k3 = (rcons k3 r3);
     v1 = Update g 1 w1 w2 k1 k2;
     v2 = Update g 2 w2 w3 k2 k3;
     v3 = Update g 3 w3 w1 k3 k1;
     w1 = (rcons w1 v1);
     w2 = (rcons w2 v2);
     w3 = (rcons w3 v3);
     c = behead c; (* remove first entry from c *)
   }
   return (k1, k2, k3, w1, w2, w3);
}
```

Listing 17: Incremental decomposition procedure

Here we use the evaluation function for arithmetic circuits defined in the previous section, which tells us that the last computed share is the output share. More formally the reconstructed output is:

$$\sum_{i\in\{1,2,3\}} \text{last } w_i \tag{13}$$

Based on procedure defined in Listing 17 we then define what it means to correctly decompose the circuit into three computational branches:

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 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 Listing 16.

Consequently, a share is only valid if it has been produced by decomposition. Namely, if p_i computes a share s then for any other party using s in their computation e. g. when computing a MULT gate, then the share used by the other parties was indeed s.

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

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

Based on the above definitions and listings, we can then define the decompose procedure that computes the three input shares based on the witness and returns the reconstructed output of the decomposition. The procedure can be seen in listing 18

```
proc decompose(h : input, c : circuit) = {
  (c, x) = h;
  x1 <$ dinput;
  x2 <$ dinput;
  x3 = x - x1 - x2;
  k1 = [];
  k2 = [];
  k3 = [];
  (k1, k2, k3, w1, w2, w3) = compute(c, [x1], [x2], [x3], k1, k2
    , k3);
  y1 = last w1;
  y2 = last w2;
  y3 = last w3;
  y = y1 + y2 + y3;
  return y;
}
```

Listing 18: Decompose procedure

HANDLING RANDOMNESS Looking at compute we see that it makes three random choices for each iteration of the while-loop and then save those choices in the random tapes. These tapes are then returned alongside the views and are used to keep track of the random decisions made throughout the protocol. The purpose of this is to be able to re-run the protocol, thus enabling any outside party to verify that each share has in-fact been computed by the decomposition and not adversarially chosen.

In most proofs, it is not essential to verify the shares of previously computed decomposition. Specifically, when considering the correctness and 2-privacy properties it is only important that the views produced by the decomposition, which we are currently computing, is going to be valid. This can be stated as an invariant on the decomposition rather than verifying it by re-running the decomposition.

Moreover, if we observe Update as defined in section 6.1.2 we see the random tapes are only used to add randomness to the evaluation of MULT gates. When summing the three shares from the multiplication gate, we see that all the randomness cancels out. From the randomness cancelling out, we can conclude that the reconstructed output of the decomposition will be the same regardless of the random tapes.

For this reason, we omit the random tapes from our proofs when we only care about the result of the computation or when we assume two procedures to make the same random choices. When the contents of the random tapes are essential for security, we will explicitly mention them.

Last, since compute samples randomness at the time it is needed by the computation, rather than before the protocol is run, it cannot be used to re-run the protocol and verify the views w.r.t. The random tapes. To this end, we define an alternative procedure, which does not sample any randomness and instead relies on the contents of the random tapes. This procedure can be seen in Listing 19 and works precisely like compute except that it assumes all randomness has been sampled before-hand.

We then give the following lemmas for describing the relation between compute and compute_fixed:

```
proc compute_fixed(c : circuit, w1 w2 w3 : view, k1 k2 k3 :
    random_tape) = {
  var g, v1, v2, v3;
  while (c <> []) {
    g = c[0];
    v1 = Update g 1 w1 w2 k1 k2;
    v2 = Update g 2 w2 w3 k2 k3;
    v3 = Update g 3 w3 w1 k3 k1;
    w1 = (rcons w1 v1);
    w2 = (rcons w2 v2);
    w3 = (rcons w3 v3);
    c = behead c;
  }
  return (w1, w2, w3);
}.
```

Listing 19: Compute with fixed randomness

Lemma 7.2.2.

$$\textit{equiv}[\text{compute} \sim \text{compute\_fixed} : ={c, w_1, w_2, w_3}$$
$$\implies \forall j. \left(\sum_{i \in \{1,2,3\}} w_i^{compute}[j] = \sum_{i \in \{1,2,3\}} w_i^{compute\_fixed}[j] \right)]$$

Proof. We show that while-loop will never break the invariant given by the postcondition. We prove this by induction on the number of step iterated through the while-loop. Now, since we know by definition of Update that the randomness has no influence on the output shares, the proof is trivially true. □

SECURITY To prove security, we state the MPC security definitions as procedures based on the above listings and definitions.

7.2.1 *Correctness*

Lemma 7.2.3 (Decomposition correctness)**.**

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

i.e. The output distributions 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 7.2.4 (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 $c = c_1 + +c_2$. 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:

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

Proof. The proof proceeded by induction on the list c.
Base case $c = []$**:** trivially true.
Induction step $c = c' + +[g]$**:** We start by splitting the procedure of compute into two calls:

```
(w1', w2', w3') = compute(c', w1, w2, w3);
(w1'', w2'', w3'') = compute([g], w1, w2, w3);
return (w1'', w2'', w3'');
```

To use the induction hypothesis, we must show that

$$\textbf{Valid circuit } c' + +[g] \implies \textbf{Valid circuit } c'.$$

Which is true by the definition of Valid Circuit. We can then apply our induction hypothesis to the compute call with circuit c'.

We are then left with showing:

$$\textbf{Valid}(c', w_1', w_2', w_3') \wedge \text{Valid circuit } c \implies \Pr[\text{compute}([g], w_1', w_2', w_3') : \textbf{Valid}(c, w_1'', w_2'', w_3'')] = 1$$

First, we note that Valid circuit c implies that g has no input wires from gates that have not already been computed.

To prove that the predicate holds after computing gate g we split the prove based on the type of g. The proof is then complete by inlining the definition of Update. □

Proof of lemma 7.2.3. 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 7.2.4 □

In some sense, our work imposes stricter restrictions on the correctness of decomposition than the proof by Giacomelli et al. [15] since we require that every share computed can be proven correct by decomposition, while the original proof of the correctness only requires that the output shares sum to the output of the circuit evaluation. This additional restriction on the views become essential for proving the security of ZKBoo.

7.2.2 *2-Privacy*

To prove 2-Privacy, we define a simulator capable of producing two views, which are indistinguishable from two views produced by the decomposition. The simulator is given by the procedure simulate and function simulator_eval in Listing 20. 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, and MULTC gates, the simulator simply calls the Update function just like compute. When evaluating MULT gates shares needs to be distributed amongst the parties, as seen in section 6.1.2. However, when computing the next share of w_e, then the computation only depends on w_e and w_{e+1}. Since the simulator simulates the view of party e and $e+1$ the view of party e can be computed normally with the Update function. For simulating the view of party $e+1$, we use the fact that shares should be uniformly random distributed, and sample a random value.

simulate is the a wrapper around simulator_eval, which is responsible for calling simulator_eval on each gate in the circuit and appending the newly computed shares to the views.

To compare the views produced by the simulator and the ones produced by the decomposition we fix two procedures real and simulated, where the former return two views

```
op simulator_eval (g : gate, p : int, e : int, w w' : view, k1
   k2 k3: int list) =
with g = MULT inputs =>
  (* Use Update to compute party e*)
  (* and sample random value for e+1*)
  if (p <> e) then k3[size w1 - 1] else Update g p w w' k1 k2
with g = ADDC inputs =>
    Update g p w w' k1 k2
with g = MULTC inputs => Update g p w w' k1 k2
with g = ADD inputs => Update g p w w' k1 k2.

proc simulate(c : circuit, e : int, w w' : view, k1 k2 k3 :
   random_tape) = {
  while (c <> []) {
    g = c[0];
    r1 <$ dinput;
    r2 <$ dinput;
    r3 <$ dinput;
    k1 = (rcons k1 r1);
    k2 = (rcons k2 r2);
    k3 = (rcons k3 r3);
    v1 = simulator_eval g e e w w' k1 k2 k3;
    v2 = simulator_eval g (e+1) e w' w k1 k2 k3;
    w1 = (rcons w1 v1);
    w2 = (rcons w2 v2);
    c = behead c;
  }
  return (w, w');
```

Listing 20: Simulator

```
proc real((c,y) : statement, x : witness, e : challenge) = {
  $x_1$ <$ dinput;
  $x_2$ <$ dinput;
  $x_3 = x - x_1 - x_2$
  $(w_1, w_2, w_3)$ = compute($c, [x_1], [x_2], [x_3]$);
  $y_1 = \text{last } w_1$;
  $y_2 = \text{last } w_2$;
  $y_3 = \text{last } w_3$;
  return $(w_e, w_{e+1}, y_{e+2})$
}

proc simulated((c, y) : statement, e : challenge) = {
    $x_e$ <$ dinput;
    $x_{e+1}$ <$ dinput;
    $(w_e, w_{e+1})$ = simulate($c, e, [x_e], [x_{e+1}]$);
    $y_e$ = 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 21: Real/Simulated view of decomposition

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 third view. These procedures can be seen in figure 21. Here, simulated is given access to the output of the decomposition as by the definition of privacy of MPC protocols.

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

Lemma 7.2.5 (Decomposition 2-Privacy)**.**

$$equiv[real \sim simulated : =\{e,x,c\} \wedge y^{simulated} = \text{eval\_circuit}\ c\ x^{real} \implies = \{w_e, w_{e+1}, y_{e+2}\}].$$

To prove this lemma, we first show that running compute and simulate 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, which is necessary to be able to reconstruct the output share y_{e+2}.

This is stated as the following lemma:

Lemma 7.2.6. Given a valid arithmetic circuit, c, in list representation with challenge e and witness x:

$$\begin{aligned} &equiv[\text{compute} \sim \text{simulated} : = \{c, e, w_e, w_{e+1}\} \\ &\qquad \implies = \{w_e, w_{e+1}\} \wedge \sum_{i \in \{1,2,3\}} \textbf{last } w_i^{compute} = \text{eval\_circuit}\ c\ x] \end{aligned}$$

Moreover, we require that the input views w_1, w_2, w_3 satisfies the correctness property from equation 14.

Figure 6: expanded procedures

```
$(w_1,w_2,w_3)$ = compute($[g],[x_1],[x_2],[x_3]$)
    ;
$(w_1,w_2,w_3)$ = compute($c',w_1,w_2,w_3$);
return $(w_1,w_2,w_3)$
```

(a) expanded compute procedure

```
$(w_e,w_{e+1})$ = simulate($[g],e,[x_e],[x_{e+1}]$)
    ;
$(w_e,w_{e+1})$ = simulate($c',e,w_e,w_{e+1}$);
return $(w_e,w_{e+1})$
```

(b) expanded simulate procedure

Figure 8: inlined procedures

```
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);

return (k1, k2, k3, w1, w2, w3)
    ;
```

(a) inlined compute procedure

```
$r_e$ <$ dinput;
$r_{e+1}$ <$ dinput;
$r_{e+2}$ <$ dinput;
$k_e$ = (rcons $k_e$ $r_e$);
$k_{e+1}$ = (rcons $k_{e+1}$ $r_{e+1}$);
$k_{e+2}$ = (rcons $k_{e+2}$ $r_{e+2}$);
$v_e$ = simulator_eval g e $w_e$ $w_{e+1}$
    $k_e$ $k_{e+1}$ $k_{e+2}$;
$v_{e+1}$ = simulator_eval g $e+1$ $w_{e+1}$
    $w_e$ $k_e$ $k_{e+1}$ $k_{e+2}$;
$w_e$ = (rcons $w_e$ $v_e$);
$w_{e+1}$ = (rcons $w_{e+1}$ $v_{e+1}$);
return $(k_e,k_{e+1}k_{e+2},w_e,w_{e+1})$;
```

(b) inlined simulate procedure

Proof. We proceed by induction on the list representation of the circuit c:
Base case $c = []$: trivial.
Induction step $c = g :: c'$: We then start by splitting the computation of compute and simulate into two separate calls; one for computing g and another for computing the circuit c' as seen in Figure 6. We then show that calling compute and simulate will satisfy $=\{w_e, w_{e+1}\}$. and that the output shares of compute sums to the output.

We proceed by inlining both procedures and explicitly stating the random tapes on which the procedures operate. This can be seen in Figure 8. We then start by applying the induction hypothesis, which states that compute and simulate called on c' is indistinguishable given that the view from running the procedures on the gate g are indistinguishable. The proof is then split based on the value of the challenge:
$e = 1$: We then further split the proof based on the type of the gate. For every gate except MULT, the computation for compute and simulate are the same for parties e and $e+1$. This is true, since simulator_eval, when called on every gate type, except MULT, will call Update. This equivalent to compute, which calls Update directly. We, therefore, only need to look at the case of MULT.

When simulating MULT view w_1 is constructed by calling Update and is therefore trivially indistinguishable. For the construction of w_2 the value is sampled at random by the simulator. More specially, the value of r_3 is used as the share of party 2. The reason why we need to use the value of r_3 is that r_1 and r_2 must be sampled equivalently between the two procedures, otherwise w_1 would not be construed correctly. r_3, however, is not used for any computation relating to the output. We are, therefore, free to manipulate the value of it.

Since the randomness is sampled at the beginning of both procedures we can use EasyCrypt's coupling functionality to reason about indistinguishability of the sampled values. In this case, we prove that the computation perform for MULT by compute is indistinguishable from the share $r_3^{simulate}$:

$$\begin{aligned}&(w_2^{compute}[l] \cdot w_2^{compute}[r] \\ &\quad + w_3^{compute}[l] \cdot w_2^{compute}[r] \\ &\quad + w_2^{compute}[l] \cdot w_3^{compute}[r] \\ &\quad + r_2^{compute} - r_3^{compute}) \sim r_3^{simulate}.\end{aligned}$$

where l and r are the indexed of the left and right input wire, respectively. Indistinguishability here follows from the fact that dinput is full and uniform, and that field, from which the shares are sampled, is closed under arithmetic operations.

Had the randomness been sampled before running the protocol, this would not have been possible, since EasyCrypt can only reason about indistinguishability of sampled values at the time of sampling.

From this we can conclude that views w_1 and w_2 are indistinguishable between the two protocols. Moreover, we have that $\sum_{i \in \{1,2,3\}} \textbf{last } w_i^{compute} = \text{eval\_circuit } [g]\ x$, by the correctness property of compute.

$e = 2 \land e = 3$ The rest of the cases proceeded like the case for $e = 1$ except when proving MULT to be indistinguishable we show indistinguishability between the random value and the view of w_3 and w_1 respectively.

□

Proof of lemma 7.2.5. By applying lemma 7.2.6 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 equation 14, which states that the shares of the real views always sum to the intermediate values of computing the circuit to conclude

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

Then we have $(y_e^{real} + y_{e+1}^{real}) \sim (y_e^{simulated} + y_{e+1}^{simulated})$ since the views of the two procedures are indistinguishable. We can then conclude:

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

□

7.3 ZKBOO

Having formalised both arithmetic circuits and their (2,3)-Decomposition we are now ready to formalise the ZKBoo protocol. Since ZKBoo is a Σ-Protocol we start by defining its types as specified in section 5.

```
type statement = circuit × int.
type witness   = int.
type statement = share × share × share × commitment ×
   commitment
× commitment.
type challenge = int.
type response  = random_tape × view × random_tape × view
```

The relation is then all tuples of (circuits, outputs, inputs), for which it holds that evaluating the circuit with the input returns the output. We formalise this as:

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

We then define the functions needed to verify the views from the decomposition, as outlined by the verification step in section 6.1.3. Here, we recall 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 share in w_e has been produced by the decomposition. To verify that w_e has been produced by the decomposition we use the following predicate:

```
pred valid_view p (w w' : view) c (k k' : random_tape) =
  $(\forall i. 0 \leq i \wedge i+1 < \mathrm{size}\ w \implies w[i+1] = \mathrm{Update}(c[i], p, w, w', k, k')$)
```

Predicates allow us to use quantifiers to assert properties within EasyCrypt, which are useful to reason about in pre- and postcondition of procedures. Predicates, however, have no computation aspect to them and are purely logical. Having a predicate quantify over all integers, for example, is perfectly legal, but 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 the size of the circuit bounds the number of computations. We can, therefore, define a computable function which checks for each entry in the circuit if the property holds and then returns if the property held for all entries. This can 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 (w w' : view) c (k k' : random_tape) =
    (foldr (fun (i, acc), acc ∧ w[i+1] = Update(c[i], p, w, w',
   k, k'))
             true (range 0 (size w - 1))).
```

This function allows us to validate the property from equation 15 computationally, but it is harder to reason about since we have to reason about every computational step of the function before we can assert the truthiness of the property. We, therefore, need to use valid_view_op in our procedures to check validity, but we would much rather use valid_view in our pre/postconditions. To achieve this we introduce a specialized version of a lemma proven by Almeida et al. [5], which allows us to replace the result of the function with the predicate:

Lemma 7.3.1 (valid_view predicate/op equivalence)**.** ∀ 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 5, which can be seen in figure 22.

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

proc init(h : statement, w : witness) = {
  x1 <$ dinput;
  x2 <$ dinput;
  x3 = x - x1 - x2;
  $(k_1, k_2, k_3, w_1, w_2, w_3)$ = Compute(c, [x1], [x2], [x3]);
  $c_1$ = Commit(($w_1, k_1$));
  $c_2$ = Commit(($w_2, k_2$));
  $c_3$ = Commit(($w_3, k_3$));
  $y_1$ = last $w_1$;
  $y_2$ = last $w_2$;
  $y_3$ = last $w_3$;
  return $(y_1, y_2, y_3, c_1, c_2, c_3)$;
}

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) = {
  $(y_1, y_2, y_3, c_1, c_2, c_3)$ = m;
  (c, y) = h;

  $(k'_e, w'_e, k'_{e+1}, w'_{e+1})$ = z;
  valid_com1 = Com.verify $(w'_e, k'_e)$ $c_e$;
  valid_com2 = Com.verify $(w'_{e+1}, k'_{e+1})$ $c_{e+1}$;
  valid_share1 = last $w'_e$     = $y_e$;
  valid_share2 = last $w'_{e+1}$ = $y_{e+1}$;
  valid = valid_view_op 1 $w'_e$ $w'_{e+1}$ c $k'_e$ $k'_{e+1}$;
  valid_length = size c = size $w'_e - 1$ /\ size $w'_e$ = size $w'_{e+1}$;

  return $y = y_1 + y_2 + y_3$ /\ valid_com1 /\ valid_com2 /\ valid_share1
     /\ valid_share2 /\ valid /\ valid_length
}
```

Listing 22: ZKBoo Σ-Protocol instantiation

7.3.1 *Security*

Given that ZKBoo is a Σ-Protocol we simply need to prove the security definitions given in section 5.1 with regards to the module ZKBoo

ASSUMPTIONS We assume that ZKBoo is given access to a secure key-less commitment scheme Com which is not allowed to access nor alter the state of the ZKBoo module. Moreover, we assume Com satisfies 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 procedure is lossless.

The requirement of Com not being able to access the state of ZKBoo is an important, yet subtle, assumption. If we did not assume this, then none of the proofs in the following section would hold. This assumption 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 then verifying the commitment in verify is equivalent to the correctness game for commitment schemes defined in chapter 4.

We, therefore, replace the calls to the commitment procedures with the correctness game which can be seen in Listing 23. To do so, we need to swap the order of the procedures in the completeness game. Most importantly, we need to move the verification of the commitments. Since we have formalised the verification of commitment as a function, i. e. it holds no state we are free to do so. If, however, Com.verify had been a procedure we could not change the order of the calls, since one verification could potentially change the state of another.

```
proc intermediate_main(h : statement, x : witness, e : challenge
    ) = {
  (c, y) = h;
  x1 <$ dinput;
  x2 <$ dinput;
  x3 = x - x1 - x2;
  (k1, k2, k3, w1, w2, w3) = Phi.compute(c, [x1], [x2], [x3]);
  $y_e$ = last $w_e$;
  $y_{e+1}$ = last $w_{e+1}$;

  valid_com1 = Correctness(Com).main(($w_e$,$k_e$));
  valid_com2 = Correctness(Com).main(($w_{e+1}$,$k_{e+1}$));
  commit(($w_{e+2}$,$k_{e+2}$));
  valid_share1 = (last $w_e = y_e$);
  valid_share2 = (last $w_{e+1} = y_{e+1}$);
  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 23: 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 Commit being lossless, we can remove the commitment to view w_{e+2} from the procedure since it does not influence the output of the procedure. Next, since the commitment scheme and the decomposition is correct 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 the correctness of the decomposition. Here the additional restrictions put on the correctness property becomes important. If the correctness of the decomposition did not ensure that the decomposition has computed every share, there would be no way to conclude the truthiness of valid_view_op.
case $e = 2, e = 3$ 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. To prove shvzk we show that running the real and the ideal procedures with the same inputs and identical random choices produce indistinguishable output values. The proof the is then split based on the value of the challenge e. The proof for the different values of e are identical so we only show the case of $e = 1$. When $e = 1$ the two procedures are:

```
proc real(h, x, e) = {
  (c, y) = h;
  x1 <$ dinput;
  x2 <$ dinput;
  x3 = x - x1 - x2;
  (k1, k2, k3, w1, w2, w3) =
    compute(c, [x1], [x2], [x3
    ]);
  $c_i$ = Commit(($w_i$,$k_i$));
  $y_i$ = last $w_i$;

  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) = {
  (c, y) = h;
  $x_e$ <$ dinput;
  $x_{e+1}$ <$ dinput;
  ($k_e$,$k_{e+1}$,$w_e$,$w_{e+1}$,$y_{e+2}$) = simulated
    ($c$,[$x_e$],[$x_{e+1}$]);

  (* Generate random list of
    shares *)
  $w_{e+2}$ = dlist dinput (size $w_e$)
    ;
  $k_{e+2}$ = dlist dinput (size $k_e$);
  $y_e$ = last $w_e$;
  $y_{e+1}$ = last $w_{e+1}$;
  $c_1$ = Commit(($w_1$,$k_1$));
  $c_2$ = Commit(($w_2$,$k_2$));
  $c_3$ = Commit(($w_3$,$k_3$));
  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;
}
```

From the 2-Privacy of the decomposition we have that $\mathrm{compute}$ and $\mathrm{simulate}$ are indistinguishable, buy only when observing the views w_e, w_{e+1} and output share y_{e+2}. By this indistinguishability we can then apply the correctness lemma to both sides, thus making both procedures return true.

We, therefore, need to argue that $c_{e+2}^{ideal} \sim c_{e+2}^{real}$. In the real case c_{e+2} is a commitment to the view produced by the decomposition. In the ideal case, however, it is a commitment to a list of random values but due to our assumption of perfect hiding, these two commitments are identically distributed. To prove this formally, we use perfect hiding definition (4.4.2) to step both procedures forward. This allows us to conclude that the two procedures are indistinguishable. □

For the proof of SHVZK, we depend on the alternative hiding property of the underlying commitment scheme. The reason for this is when trying to prove indistinguishable between the two programs using $\mathrm{EasyCrypt}$'s pRHL we ultimately have to prove a statement of the form:

$$equiv[\mathrm{Com.commit}(m_1) \sim \mathrm{Com.commit}(m_2) : =\{\textbf{glob}\ Com\} \implies =\{res\}]$$

If we were given the original hiding definition based on an adversary, it is not immediately clear how to apply this notion of indistinguishability in relation to the pRHL statement above.

Otherwise, to use the original definition of hiding (definition 4.3.2) we would have to change the SHVZK definition to be an adversary-based game. Proving this, however, would require more intermediate steps, since we would have to construct an adversary breaking the SHVZK property based on one that violates the hiding property.

Lastly, we want to prove the 3-special soundness property of ZKBoo. To do so, we first inline the procedures of the soundness game. We then group the procedures into a procedure, $\mathrm{extract\_views}$, which verifies that the views are accepting. To verify the views it uses the binding game, which only checks that the responses does not break the binding property. This is seen in Listing 24

Here, we replace the calls to $\mathrm{Com.verify}$ with the alternative binding game from definition 4.4.3. This replacement does not change the output distribution of the soundness game since the output distribution of verifying a commitment and the alternative binding game is identical.

From the new soundness game, we then prove two helper lemmas. First we introduce a lemma that allows us to reason about openings given by:

$$\begin{aligned} z_1 &= (w_1, w_2) \\ z_2 &= (w_2', w_3) \\ z_3 &= (w_3', w_1') \end{aligned}$$

For each view we have there are two potential openings w_i and w_i'. Now, since the two potential openings corresponds to a commitment in message a, it should hold that $w_i = w_i'$. Otherwise, the prover has managed to break the binding property of the commitment scheme. When $\forall i. w_i = w_i'$ we refer to the responses as begin *consistent*.

Lemma 7.3.4. Assuming that all three transcripts are accepting and that the probability of breaking the binding game with three attempts is *binding_prob* then:

$$\Pr[\mathrm{extract\_views}(h, m, z_1, z_2, z_3) : v_1 \wedge v_2 \wedge v_3 \wedge w_i = w_i'] = (1 - binding\_prob)$$

```
local module SoundnessInter = {
  proc extract_views(h : statement, m : message, z1 z2 z3 :
   response) = {
    v1 = ZK.verify(h, m, 1, z1);
    v2 = ZK.verify(h, m, 2, z2);
    v3 = ZK.verify(h, m, 3, z3);

    (k1, w1, k2, w2) = z1;
    (k2', w2', k3, w3) = z2;
    (k3', w3', k1', w1') = z3;
    (y1, y2, y3, c1, c2, c3) = m;
    cons1 = alt_binding(c1, w1, w1');
    cons2 = alt_binding(c2, w2, w2');
    cons3 = alt_binding(c3, w3, w3');

    return v1 /\ v2 /\ v3 /\ cons1 /\ cons2 /\ cons3;
  }

  proc main(h : statement, m : message, z1 z2 z3 : response) = {
    v = extract_views(h, m, z1, z2, z3);
    x = witness_extractor(h, m, [1;2;3], [z1;z2;z3]);

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

Listing 24: Soundness game for ZKBoo

Proof. This follows directly from inlining all procedures and applying our assumptions. □

Consequently, since $\mathrm{extract\_views}$ returns true for all calls to verify we can conclude $\mathbf{Valid}(c, w_1, w_2, w_3)$ and $\sum_{i \in \{1,2,3\}}$ last $w_i = y$

Next, we prove that if the responses are consistent then we can extract a witness satisfying R.

Lemma 7.3.5. Given consistent responses with openings w_1, w_2, w_3 and randomness k_1, k_2, k_3 then

$$\mathbf{Valid}(c, w_1, w_2, w_3) \implies \Pr[\mathrm{witness\_extractor} : \mathrm{R\ h}\ [w_1[0] + w_2[0] + w_3[0]]] = 1$$

Proof. We start by unfolding the relation:

$$\begin{aligned}
& \mathrm{R\ h}\ [w_1[0] + w_2[0] + w_3[0]] & \\
&= \mathrm{eval\_circuit}\ c\ [w_1[0] + w_2[0] + w_3[0]] = y & \\
\iff & \Pr[\mathrm{eval\_circuit}(c, [w_1[0] + w_2[0] + w_3[0]]) = y] = 1 & \text{lemma 7.1.5} \\
&= \Pr[\mathrm{decomposition}(c, [w_1[0]], [w_2[0]],\ [w_3[0]]) = y] & \text{Decomposition correctness} \\
&= \Pr[\mathrm{decomposition\_fixed}(c, x, [w_1[0]], [w_2[0]], [w_3[0]], k_1, k_2, k_3) = y] & \text{lemma 7.2.2}
\end{aligned}$$

From this we use $\mathbf{Valid}(c, w_1, w_2, w_3)$ to prove that the computation starting with the input shares $w_i[0]$ will result in output y

To do so we prove that after running the while-loop of procedure $\mathrm{decomposition\_fixed}$ producing views w_i' then $w_i' = w_i$.

We prove this by showing that if this invariant holds before one iteration of the while-loop then it will also hold after the iteration.

This follows directly from $\mathbf{Valid}(c, w_1, w_2, w_3)$ since at any iteration of the while-loop it will perform exactly the same computation as the ones performed to compute views w_1, w_2, w_3. Next, we have to show that the invariant is true after the first iteration of the while-loop. This is again true by $\mathbf{Valid}(c, w_1, w_2, w_3)$.

After having executed every iteration of the while-loop we are left with views matching w_1, w_2, w_3, which by our assumption of $\mathbf{Valid}(c, w_1, w_2, w_3)$ have output shares summing to y. We can, therefore, conclude that the decomposition starting with values $w_i[0]$ produce the correct output y; hence summing the input shares must be a valid witness for the relation. □

Based on these two lemmas are now ready to prove special soundness for ZKBoo.

Lemma 7.3.6. Given accepting transcripts $(a, e_i, z_i)_{i=\{1,2,3\}}$ we have:

$$\Pr[\mathrm{SpecialSoundness(ZKBoo)}.main = true] = (1 - binding\_prob)$$

Proof. First we proceed with replacing $\mathrm{SpecialSoundness(ZKBOO).main}$ with $\mathrm{SoundnessInter.main}$

We then split the execution of the procedure into thee parts:

1) First we show that we can execute $\mathrm{extract\_views}$ with output true with probability $(1 - binding\_prob)$

2) If $\mathrm{extract\_views}$ outputs true then the rest of the procedures will output true with probability 1

2) If extract_views outputs then the rest of procedure will also output false with probability 1.

To prove **1)** we apply lemma 7.3.4. Next, we show **2)**. Since we assume extract_views to have output true, we know that the openings must be consistent. We then use lemma 7.3.5 to conclude that we can produce a valid witness. The result of the procedure will then output true. Last, we show 3) which follows directly from the fact that if extract_views output false the rest of the procedure immediately fails by definition.

Combining the three above facts, we can conclude that the procedure will output true with probability $1 - binding\_prob$. □

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 [4, 5, 8, 12, 18]

Butler et al. [12] formalises both Σ-Protocols and commitment schemes within Isabelle/CryptoHOL. Additionally, they formalise the proof 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 ours 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. [8] successfully formalised Σ-Protocols with CertiCrypt. Their work include 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 without formalisation of ZKBoo. ZKBoo, however, defines a more general class of Σ-Protocols than the one defined in this paper.

Moreover, commitment schemes have been formalised in EasyCrypt by Metere and Dong [18]. Their work differs from ours by offering fewer 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. [4] 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, numerous formalisation of Multi-Part Computations exists. Almeida et al. [5] formalised secure function evaluation with Yao's protocol in EasyCrypt. Their work also included a formalisation of circuits. Butler et al. [11] formalised a two party MPC protocol based on secret sharing against a semi-honest adversary. Their work also manage to formally prove the privacy property with a simulator, akin to our work on the (2,3)-Decomposition in section 7.2. Haagh et al. [16] then formalised security of secret-sharing based MPC, but in the presence of an active adversary in EasyCrypt.

8.2 DISCUSSION

Throughout the work of this thesis we have used the EasyCrypt proof assistant to formally verify the proofs presented herein. The work has been an iterative process between formulating a lemma and then trying to prove it within EasyCrypt. This process was then repeated until the lemma would be formally proven. Consequently, EasyCrypt has been a instrumental part of the work formulated in this thesis. Moreover, the powerful automation tools offered by EasyCrypt has allowed us to discharge trivial proofs, thus enabling us to spend more time working on the complex lemmas seen within this thesis.

Overall, we feel that EasyCrypt captures the models used by cryptographers quite well; our formalisation of ZKBoo has a structure akin to the one presented in the original paper by Giacomelli et al. [15]. This has enabled us to spend less time formulating the protocol and more time on formalising important security criteria. This has been capacitated, in part, by EasyCrypt *pWhile* language for implementing procedures. This language follow a structure closely resembling the pseudo-code 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.

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

In particular EasyCrypt offers its rPHL for proving procedures to be perfectly indistinguishable. If, however, computational indistinguishability is needed then the rPHL logic cannot directly be used, and we instead have to deal with adversaries comparing procedures.

This problem is part of a more general problem where EasyCrypt in essence offers two techniques for dealing with cryptographic proofs. The first is the traditional adversaries game-hopping technique where we reason about an adversary being able to break the security of the protocol. These adversaries can then be used to construct new adversaries that can break the security of other protocols. The other techniques is showing indistinguishability of the output distributions of the programs with EasyCrypt's relational logic. Both of these techniques are perfectly valid for proving security of cryptographic protocols. However, at the time of writing this thesis it is not clear to us how to formally prove the relation between the two techniques. The gap between the two techniques became apparent in chapter 7 where the adversarial-based security definitions of our commitment schemes did not conform to the goals needed to prove security of our Σ-Protocol formalisation.

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 [3] 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.

8.3 FUTURE WORK

In this thesis we have formalised Σ-Protocols and commitment schemes that is applicable to larger cryptographic protocols, as shown by our formalisation of ZKBoo. However, various improvement has since been made to the ZKBoo protocol. Notably, the ZB++ protocol, which offers a reduction in the size of messages/responses sent to the verifier from the prover. Moreover, it also provides zero-knowledge in a post-quantum context [13]. An interesting next step could, therefore, be to use our existing formalisation of ZKBoo to formally verify the improvements made by ZB++.

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 function to represented as a circuit and then defined an execution order. However, 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 allow us to reason about more general zero-knowledge protocols than the sub-class of Σ-Protocol which we have explored in this thesis.

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 [8, 12]. 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 from other proof assistants and showing the workability of EasyCrypt, whilst also showing how our formalisation can be used to fuel future works 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

Particularly we have seen in section 7.3.1 how important small assumption are for security of implementations of cryptographic protocols. If one procedure is allowed to observe the state of another running on the system all proofs in the aforementioned section would not hold. These assumptions 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] Concordium white paper. URL https://concordium.com/wp-content/uploads/2020/04/Concordium-White-Paper-Vol.-1.0-April-2020-1.pdf.

[2] *Easycrypt source code*. URL https://github.com/EasyCrypt/easycrypt.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptology ePrint Archive*, 2004:331, 01 2004.

[11] David Butler, David Aspinall, and Adrià Gascón. How to simulate it in isabelle: Towards formal proof for secure multi-party computation. *CoRR*, abs/1805.12482, 2018. URL http://arxiv.org/abs/1805.12482.

[12] 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.

[13] 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.

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

[15] 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.

[16] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. *CoRR*, abs/1806.07197, 2018. URL http://arxiv.org/abs/1806.07197.

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

[18] 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.