https://verified-zkevm.github.io/ArkLib/ https://github.com/Verified-zkEVM/ArkLib https://verified-zkevm.github.io/ArkLib/docs

Formally Verified Arguments of Knowledge in Lean

April 24, 2025

Chapter 1

Introduction

The goal of this project is to formalize Succinct Non-Interactive Arguments of Knowledge (SNARKs) in Lean. Our focus is on SNARKs based on Interactive Oracle Proofs (IOPs) and variants thereof (i.e. Polynomial IOPs). We aim to develop a general framework for IOP-based SNARKs with verified, modular building blocks and transformations. This modular approach enables us to construct complex protocols from simpler components while ensuring correctness and soundness by construction.

Chapter 2

Oracle Reductions

2.1 Definitions

In this section, we give the basic definitions of a public-coin interactive oracle reduction (henceforth called an oracle reduction or IOR). In particular, we will define its building blocks, and various security properties.

2.1.1 Format

An **(interactive) oracle reduction** is an interactive protocol between two parties, a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$. It takes place in the following setting:

1. We work in an ambient dependent type theory (in our case, Lean).
2. The protocol flow is fixed and defined by a given *type signature*, which describes in each round which party sends a message to the other, and the type of that message.
3. The prover and verifier has access to some inputs (called the *context*) at the beginning of the protocol. These inputs are classified as follows:
   - *Public inputs*: available to both parties;
   - *Private inputs* (or *witness*): available only to the prover;
   - *Oracle inputs*: the underlying data is available to the prover, but it's only exposed as an oracle to the verifier. An oracle interface for such inputs consists of a query type, a response type, and a function that takes in the underlying input, a query, and outputs a response.
   - *Shared oracle*: the oracle is available to both parties via an interface; in most cases, it is either empty, a probabilistic sampling oracle, or a random oracle. See **??** for more information on oracle computations.
4. The messages sent from the prover may either: 1) be seen directly by the verifier, or 2) only available to a verifier through an *oracle interface* (which

specifies the type for the query and response, and the oracle's behavior given the underlying message).

5. All messages from $\mathcal{V}$ are chosen uniformly at random from the finite type corresponding to that round. This property is called *public-coin* in the literature.

We now go into more details on these objects.

Definition 1 (Type Signature of an Oracle Reduction)**.** A protocol specification $\rho : \mathsf{PSpec}\ n$ of an oracle reduction is parametrized by a fixed number of messages sent in total. For each step of interaction, it specifies a direction for the messsage (prover to verifier, or vice versa), and a type for the message. If a message from the prover to the verifier is further marked as oracle, then we also expect an oracle interface for that message. In Lean, we handle these oracle interfaces via the $\mathsf{ToOracle}$ type class.

Definition 2 (Context)**.** In an oracle reduction, its *context* (denoted Γ) consists of a list of public inputs, a list of witness inputs, a list of oracle inputs, and a shared oracle (possibly represented as a list of lazily sampled query-response pairs). These inputs have the expected visibility.

For simplicity, we imagine the context as *append-only*, as we add new messages from the protocol execution.

We define some supporting definitions for a protocol specification.

Definition 3 (Transcript & Related Definitions)**.** Given protocol specification $\rho : \mathsf{PSpec}\ n$, its *transcript* up to round i is an element of type $\rho_{[:i]} ::= \prod_{j<i} \rho\ j$. We define the type of all challenges sent by the verifier as $\rho.\mathsf{Chals} ::= \prod_{i \text{ s.t. } (\rho\ i).\mathsf{fst}=\mathsf{V2P}} (\rho\ i).\mathsf{snd}$.

Remark 4 (Design Decision)**.** We do not enforce a particular interaction flow in the definition of an interactive (oracle) reduction. This is done so that we can capture all protocols in the most generality. Also, we want to allow the prover to send multiple messages in a row, since each message may have a different oracle representation (for instance, in the Plonk protocol, the prover's first message is a 3-tuple of polynomial commitments.)

Definition 5 (Type Signature of a Prover)**.** A prover $\mathcal{P}$ in an oracle reduction, given a context, is a stateful oracle computation that at each step of the protocol, either takes in a new message from the verifier, or sends a new message to the verifier.

Our modeling of oracle reductions only consider *public-coin* verifiers; that is, verifiers who only outputs uniformly random challenges drawn from the (finite) types, and uses no other randomness. Because of this fixed functionality, we can bake the verifier's behavior in the interaction phase directly into the protocol execution semantics.

After the interaction phase, the verifier may then run some verification procedure to check the validity of the prover's responses. In this procedure, the verifier gets access to the public part of the context, and oracle access to either the shared oracle, or the oracle inputs.

Definition 6 (Type Signature of a Verifier)**.** A verifier $\mathcal{V}$ in an oracle reduction is an oracle computation that may perform a series of checks (i.e. 'Bool'-valued, or 'Option Unit') on the given context.

An oracle reduction then consists of a type signature for the interaction, and a pair of prover and verifier for that type signature.
l

Definition 7 (Interactive Oracle Reduction)**.** An interactive oracle reduction for a given context Γ is a combination a prover and a verifier of the types specified above.

PL Formalization. We write our definitions in PL notation in **??**. The set of types Type is the same as Lean's dependent type theory (omitting universe levels); in particular, we care about basic dependent types (Pi and Sigma), finite natural numbers, finite fields, lists, vectors, and polynomials.

Using programming language notation, we can express an interactive oracle reduction as a typing judgment:

$$\Gamma := (\Psi; \Theta; \Sigma; \rho; \mathcal{O}) \vdash \mathcal{P} : \tau_{\mathsf{P}}(\Gamma),\ \mathcal{V} : \tau_{\mathsf{V}}(\Gamma)$$

where:

- Ψ represents the witness (private) inputs
- Θ represents the oracle inputs
- Σ represents the public inputs (i.e. statements)
- $\mathcal{O} : \mathsf{OSpec}\ \iota$ represents the shared oracle
- $\rho : \mathsf{PSpec}\ n$ represents the protocol type signature
- $\mathcal{P}$ and $\mathcal{V}$ are the prover and verifier, respectively, being of the given types $\tau_{\mathsf{P}}(\Gamma)$ and $\tau_{\mathsf{V}}(\Gamma)$.

To exhibit valid elements for the prover and verifier types, we will use existing functions in the ambient programming language (e.g. Lean).

We now define what it means to execute an oracle reduction. This is essentially achieved by first executing the prover, interspersed with oracle queries to get the verifier's challenges (these will be given uniform random probability semantics later on), and then executing the verifier's checks. Any message exchanged in the protocol will be added to the context. We may also log information about the execution, such as the log of oracle queries for the shared oracles, for analysis purposes (i.e. feeding information into the extractor).

$$
\begin{array}{rcl}
\mathsf{Type} & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathbb{N} \mid \mathsf{Fin}\ n \mid \mathbb{F}_q \mid \mathsf{List}\ (\alpha : \mathsf{Type}) \mid (i : \iota) \to \alpha\ i \mid (i : \iota) \times \alpha\ i \mid \ldots \\
\\
\mathsf{Dir} & ::= & \mathsf{P2V.Pub} \mid \mathsf{P2V.Orac} \mid \mathsf{V2P} \\
\mathsf{OI}\ (\mathrm{M} : \mathsf{Type}) & ::= & \langle \mathrm{Q}, \mathrm{R}, \mathrm{M} \to \mathrm{Q} \to \mathrm{R} \rangle \\
\mathsf{PSpec}\ (n : \mathbb{N}) & ::= & \mathsf{Fin}\ n \to (d : \mathsf{Dir}) \times (M : \mathsf{Type}) \times (\mathsf{if}\ d = \mathsf{P2V.Orac}\ \mathsf{then}\ \mathsf{OI}(M)\ \mathsf{else}\ \mathsf{Unit}) \\
\mathsf{OSpec}\ (\iota : \mathsf{Type}) & ::= & (i : \iota) \to \mathsf{dom}\ i \times \mathsf{range}\ i \\
\\
\Sigma & ::= & \emptyset \mid \Sigma \times \mathsf{Type} \\
\Omega & ::= & \emptyset \mid \Omega \times \langle \mathrm{M} : \mathsf{Type}, \mathsf{OI}(\mathrm{M}) \rangle \\
\Psi & ::= & \emptyset \mid \Psi \times \mathsf{Type} \\
\\
\Gamma & ::= & (\Psi; \Omega; \Sigma; \rho; \mathcal{O}) \\
\mathsf{OComp}^{\mathcal{O}}\ (\alpha : \mathsf{Type}) & ::= & \mid\ \mathsf{pure}\ (a : \alpha) \\
& & \mid\ \mathsf{queryBind}\ (i : \iota)\ (q : \mathsf{dom}\ i)\ (k : \mathsf{range}\ i \to \mathsf{OComp}^{\mathcal{O}}\ \alpha) \\
& & \mid\ \mathsf{fail} \\
\\
\tau_{\mathsf{P}}(\Gamma) & ::= & (i : \mathsf{Fin}\ n) \to (h : (\rho\ i).\mathsf{fst} = \mathsf{P2V}) \to \\
& & \Sigma \to \Omega \to \Psi \to \rho_{[:i]} \to \mathsf{OComp}^{\mathcal{O}}\ ((\rho\ i).\mathsf{snd}) \\
\\
\tau_{\mathsf{V}}(\Gamma) & ::= & \Sigma \to (\rho.\mathsf{Chals}) \to \mathsf{OComp}^{\mathcal{O}::\mathsf{OI}(\Omega)::\mathsf{OI}(\rho.\mathsf{Msg.Orac})}\ \mathsf{Unit} \\
\\
\tau_{\mathsf{E}}(\Gamma) & ::= & \Sigma \to \Omega \to \rho.\mathsf{Transcript} \to \mathcal{O}.\mathsf{QueryLog} \to \Psi
\end{array}
$$

Figure 2.1: Type definitions for interactive oracle reductions

Definition 8 (Execution of an Oracle Reduction)**.**

Remark 9 (More efficient representation of oracle reductions)**.** The presentation of oracle reductions as protocols on an append-only context is useful for reasoning, but it does not lead to the most efficient implementation for the prover and verifier. In particular, the prover cannot keep intermediate state, and thus needs to recompute everything from scratch for each new message.

To fix this mismatch, we will also define a stateful variant of the prover, and define a notion of observational equivalence between the stateless and stateful reductions.

2.1.2 Security properties

We can now define properties of interactive reductions. The two main properties we consider in this project are completeness and various notions of soundness. We will cover zero-knowledge at a later stage.

First, for completeness, this is essentially probabilistic Hoare-style conditions on the execution of the oracle reduction (with the honest prover and verifier). In other words, given a predicate on the initial context, and a predicate on the final context, we require that if the initial predicate holds, then the final predicate

holds with high probability (except for some *completeness* error).

Definition 10 (Completeness)**.**

Almost all oracle reductions we consider actually satisfy *perfect completeness*, which simplifies the proof obligation. In particular, this means we only need to show that no matter what challenges are chosen, the verifier will always accept given messages from the honest prover.

For soundness, we need to consider different notions. These notions differ in two main aspects:

- Whether we consider the plain soundness, or knowledge soundness. The latter relies on the notion of an *extractor*.
- Whether we consider plain, state-restoration, round-by-round, or rewinding notion of soundness.

We note that state-restoration knowledge soundness is necessary for the security of the SNARK protocol obtained from the oracle reduction after composing with a commitment scheme and applying the Fiat-Shamir transform. It in turn is implied by either round-by-round knowledge soundness, or special soundness (via rewinding). At the moment, we only care about non-rewinding soundness, so mostly we will care about round-by-round knowledge soundness.

Definition 11 (Soundness)**.**

A (straightline) extractor for knowledge soundness is a deterministic algorithm that takes in the output public context after executing the oracle reduction, the side information (i.e. log of oracle queries from the malicious prover) observed during execution, and outputs the witness for the input context.

Note that since we assume the context is append-only, and we append only the public (or oracle) messages obtained during protocol execution, it follows that the witness stays the same throughout the execution.

Definition 12 (Knowledge Soundness)**.**

To define round-by-round (knowledge) soundness, we need to define the notion of a *state function*. This is a (possibly inefficient) function StateF that, for every challenge sent by the verifier, takes in the transcript of the protocol so far and outputs whether the state is doomed or not. Roughly speaking, the requirement of round-by-round soundness is that, for any (possibly malicious) prover P, if the state function outputs that the state is doomed on some partial transcript of the protocol, then the verifier will reject with high probability.

Definition 13 (State Function)**.**

Definition 14 (Round-by-Round Soundness)**.**

Definition 15 (Round-by-Round Knowledge Soundness)**.**

By default, the properties we consider are perfect completeness and (straight-line) round-by-round knowledge soundness. We can encapsulate these properties into the following typing judgement:

$$\Gamma := (\Psi; \Theta; \Sigma; \rho; \mathcal{O}) \vdash \{\mathcal{R}_1\} \quad \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle \quad \{\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\}$$

2.2 A Program Logic for Oracle Reductions

In this section, we describe a program logic for reasoning about oracle reductions. In other words, we define a number of rules (or theorems) that govern how oracle reductions can be composed to form larger reductions, and how the resulting reduction inherits the security properties of the components.

The first group of rules changes relations and shared oracles.

2.2.1 Changing Relations and Oracles

Here we express the consequence rule. Namely, if we have an oracle reduction for $\mathcal{R}_1 \implies \mathcal{R}_2$, along with $\mathcal{R}'_1 \implies \mathcal{R}_1$ and $\mathcal{R}_2 \implies \mathcal{R}'_2$, then we obtain an oracle reduction for $\mathcal{R}'_1 \implies \mathcal{R}'_2$.

$$\frac{\begin{array}{c}\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P},\ \mathcal{V},\ \mathcal{E} \rangle^{\mathcal{O}} : \tau\ \{\!\{\mathcal{R}_2;\ \mathsf{St};\ \epsilon\}\!\} \\ \mathcal{R}'_1 \implies \mathcal{R}_1 \\ \mathcal{R}_2 \implies \mathcal{R}'_2\end{array}}{\Psi; \Theta; \Sigma \vdash \{\mathcal{R}'_1\}\ \langle \mathcal{P},\ \mathcal{V},\ \mathcal{E} \rangle^{\mathcal{O}} : \tau\ \{\!\{\mathcal{R}'_2;\ \mathsf{St};\ \epsilon\}\!\}} \quad \text{(Conseq)}$$

$$\frac{\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}} : \tau\ \{\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\}}{\Psi; \Theta; \Sigma \vdash \{\mathcal{R} \times \mathcal{R}_1\}\ \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}} : \tau\ \{\!\{\mathcal{R} \times \mathcal{R}_2; \mathsf{St}; \epsilon\}\!\}} \quad \text{(Frame)}$$

$$\frac{\begin{array}{c}\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}_1} : \tau\ \{\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\} \\ \mathcal{O}_1 \subset \mathcal{O}_2\end{array}}{\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}_2} : \tau\ \{\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\}} \quad \text{(Oracle-Lift)}$$

TODO: figure out how the state function needs to change for these rules (they are basically the same, but not exactly)

2.2.2 Sequential Composition

The reason why we consider interactive (oracle) reductions at the core of our formalism is that we can *compose* these reductions to form larger reductions. Equivalently, we can take a complex *interactive (oracle) proof* (which differs only in that it reduces a relation to the *trivial* relation that always outputs true) and break it down into a series of smaller reductions. The advantage of this approach is that we can prove security properties (completeness and soundness) for each of the smaller reductions, and these properties will automatically transfer to the larger reductions.

This section is devoted to the composition of interactive (oracle) reductions, and proofs that the resulting reductions inherit the security properties of the two (or more) constituent reductions.

Sequential composition can be expressed as the folowing rule:

$$\frac{\begin{array}{c}\Psi;\Theta;\Sigma \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P}_1, \mathcal{V}_1, \mathcal{E}_1\rangle^{\mathcal{O}} : \tau_1\ \{\!\{\mathcal{R}_2; \mathsf{St}_1; \epsilon_1\}\!\} \\ \Psi;(\Theta :: \tau_1);\Sigma \vdash \{\mathcal{R}_2\}\ \langle \mathcal{P}_2, \mathcal{V}_2, \mathcal{E}_2\rangle^{\mathcal{O}} : \tau_2\ \{\!\{\mathcal{R}_3; \mathsf{St}_2; \epsilon_2\}\!\}\end{array}}{\Psi;(\Theta :: \tau_1 :: \tau_2)\ ;\Sigma \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P}_1 \circ \mathcal{P}_2, \mathcal{V}_1 \circ \mathcal{V}_2, \mathcal{E}_1 \circ_{\mathcal{V}_2} \mathcal{E}_2\rangle^{\mathcal{O}} : \tau_1 \oplus \tau_2\ \{\!\{\mathcal{R}_3; \mathsf{St}_1 \oplus \mathsf{St}_2; \epsilon_1 \oplus \epsilon_2\}\!\}} \quad \text{(Seq-Comp)}$$

2.2.3 Virtualization

Another tool we will repeatedly use is the ability to change the context of an oracle reduction. This is often needed when we want to adapt an oracle reduction in a simple context into one for a more complex context.

See the section on sum-check **??** for an example.

Definition 16 (Mapping into Virtual Context)**.** In order to apply an oracle reduction on virtual data, we will need to provide a mapping from the current context to the virtual context. This includes:

- A mapping from the current public inputs to the virtual public inputs.
- A simulation of the oracle inputs for the virtual context using the public and oracle inputs for the current context.
- A mapping from the current private inputs to the virtual private inputs.
- A simulation of the shared oracle for the virtual context using the shared oracle for the current context.

Definition 17 (Virtual Oracle Reduction)**.** Given a suitable mapping into a virtual context, we may define an oracle reduction via the following construction:

- The prover first applies the mappings to obtain the virtual context. The verifier does the same, but only for the non-private inputs.
- The prover and verifier then run the virtual oracle reduction on the virtual context.

We will show security properties for this virtualization process. One can see that completeness and soundness are inherited from the completeness and soundness of the virtual oracle reduction. However, (round-by-round) knowledge soundness is more tricky; this is because we must extract back to the witness of the original context from the virtual context.

$$\frac{\begin{array}{c}\Psi';\Theta';\Sigma' \vdash \{\mathcal{R}_1\}\ \langle \mathcal{P}, \mathcal{V}, \mathcal{E}\rangle^{\mathcal{O}} : \tau\ \{\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\} \\ f : (\Psi, \Theta, \Sigma) \to (\Psi', \Theta', \Sigma') \\ g : \Psi' \to \Psi \\ f.\mathsf{fst} \circ g = \mathsf{id}\end{array}}{\Psi;\Theta;\Sigma \vdash \{\mathcal{R}_1 \circ f\}\ \langle \mathcal{P} \circ f, \mathcal{V} \circ f, \mathcal{E} \circ (f, g)\rangle^{\mathcal{O}} : \tau\ \{\!\{\mathcal{R}_2 \circ f; \mathsf{St} \circ f; \epsilon\}\!\}} \quad \text{(Virtual-Ctx)}$$

2.2.4 Substitution

Finally, we need a transformation / inference rule that allows us to change the message type in a given round of an oracle reduction. In other words, we substitute a value in the round with another value, followed by a reduction establishing the relationship between the new and old values.

Examples include:

1. Substituting an oracle input by a public input:
   - Often by just revealing the underlying data. This has no change on the prover, and for the verifier, this means that any query to the oracle input can be locally computed.
   - A variant of this is when the oracle input consists of a data along with a proof that the data satisfies some predicate. In this case, the verifier needs to additionally check that the predicate holds for the substituted data.
   - Another common substitution is to replace a vector with its Merkle commitment, or a polynomial with its polynomial commitment.
2. Substituting an oracle input by another oracle input, followed by a reduction for each oracle query the verifier makes to the old oracle:
   - This is also a variant of the previous case, where we do not fully substitute with a public input, but do a "half-substitution" by substituting with another oracle input. This happens e.g. when using a polynomial commitment scheme that is itself based on a vector commitment scheme. One can cast protocols like Ligero / Brakedown / FRI / STIR in this two-step process.

Chapter 3

Proof Systems

3.1 Simple Oracle Reductions

We start by introducing a number of simple oracle reductions.

3.1.1 Polynomial Equality Testing

Context: two univariate polynomials $P, Q \in \mathbb{F}[X]$ of degree at most d, available as polynomial evaluation oracles

Input relation: $P = Q$ as polynomials

Protocol type: a single message of type $\mathbb{F}$ from the verifier to the prover.

Honest prover: does nothing

Honest verifier: checks that $P(r) = Q(r)$

Output relation: $P(r) = Q(r)$

Extractor: trivial since there is no witness

Completenes: trivial

Round-by-round state function: corresponds precisely to input and output relation

Round-by-round error: $d/|\mathbb{F}|$

Round-by-round knowledge soundness: follows from Schwartz-Zippel

To summarize, we have the following judgment:

$$\Psi := (); \ \Theta := (P, Q); \ \Sigma := (); \ \tau := (\mathsf{V2P}, \mathbb{F}) \vdash$$

$$\{P = Q\} \quad \left(\begin{array}{l} \mathcal{P} := (), \\ \mathcal{V} := (P, Q, r) \mapsto [P(r) \stackrel{?}{=} Q(r)], \\ \mathcal{E} := () \end{array} \right)^{\emptyset} \quad \{\!\{P(r) = Q(r); \mathsf{St}_{P,Q}; \tfrac{d}{|\mathbb{F}|}\}\!\}$$

where $\mathsf{St}(i) = \begin{cases} P \stackrel{?}{=} Q \text{ if } i = 0 \\ P(r) \stackrel{?}{=} Q(r) \text{ if } i = 1 \end{cases}$

3.1.2 Batching Polynomial Evaluation Claims

Context: n-tuple of values $v = (v_1, \ldots, v_n) \in \mathbb{F}^n$

Protocol type: one message of type $\mathbb{F}^k$ from the verifier to the prover, and another message of type $\mathbb{F}$ from the prover to the verifier

Auxiliary function: a polynomial map $E : \mathbb{F}^k \to \mathbb{F}^n$

Honest prover: given $r \leftarrow \mathbb{F}^k$ from the verifier's message, computes $\langle E(r), v\rangle := E(r)_1 \cdot v_1 + \cdots + E(r)_n \cdot v_n$ and sends it to the verifier

Honest verifier: checks that the received value v' is equal to $\langle E(r), v\rangle$

Extractor: trivial since there is no witness

Security: depends on the degree & non-degeneracy of the polynomial map E

3.2 The Sum-Check Protocol

In this section, we describe the sum-check protocol [**?**] in a modular manner, as a running example for our approach to specifying and proving properties of oracle reductions (based on a program logic approach).

The sum-check protocol, as described in the original paper and many expositions thereafter, is a protocol to reduce the claim that

$$\sum_{x \in \{0,1\}^n} P(x) = c,$$

where P is an n-variate polynomial of certain individual degree bounds, and c is some field element, to the claim that

$$P(r) = v,$$

for some claimed value v (derived from the protocol transcript), where r is a vector of random challenges from the verifier sent during the protocol.

In our language, the initial context of the sum-check protocol is the pair (P, c), where P is an oracle input and c is public. The protocol proceeds in n rounds of interaction, where in each round i the prover sends a univariate polynomial s_i of bounded degree and the verifier sends a challenge $r_i \leftarrow \mathbb{F}$. The honest prover would compute

$$s_i(X) = \sum_{x \in \{0,1\}^{n-i-1}} P(r_1, \ldots, r_{i-1}, X, x),$$

and the honest verifier would check that $s_i(0) + s_i(1) = s_{i-1}(r_{i-1})$, with the convention that $s_0(r_0) = c$.

Theorem 18. *The sum-check protocol is complete.*

We now proceed to break down this protocol into individual message, and then specify the predicates that should hold before and after each message is exchanged.

First, it is clear that we can consider each round in isolation. In fact, each round can be seen as an instantiation of the following simpler "virtual" protocol:

1. In this protocol, the context is a pair (p, d), where p is now a *univariate* polynomial of bounded degree. The predicate / relation is that $p(0) + p(1) = d$.

2. The prover first sends a univariate polynomial s of the same bounded degree as p. In the honest case, it would just send p itself.

3. The verifier samples and sends a random challenge $r \leftarrow \mathbb{F}$.

4. The verifier checks that $s(0) + s(1) = d$. The predicate on the resulting output context is that $p(r) = s(r)$.

The reason why this simpler protocol is related to a sum-check round is that we can *emulate* the simpler protocol using variables in the context at the time:

- The univariate polynomial p is instantiated as $\sum_{x \in \{0,1\}^{n-i-1}} P(r_1, \ldots, r_{i-1}, X, x)$.
- The scalar d is instantiated as c if $i = 0$, and as $s_{i-1}(r_{i-1})$ otherwise.

It is "clear" that the simpler protocol is perfectly complete. It is sound (and since there is no witness, also knowledge sound) since by the Schwartz-Zippel Lemma, the probability that $p \neq s$ and yet $p(r) = s(r)$ for a random challenge r is at most the degree of p over the size of the field.

Note that there is no witness so knowledge soundness follows trivially from soundness. Moreover, we can define the following state function for the simpler protocol:

1. The initial state funtion is the same as the predicate on the initial context, namely that $p(0) + p(1) = d$.

2. The state function after the prover sends s is the predicate that $p(0) + p(1) = d$ and $s(0) + s(1) = d$. Essentially, we add in the verifier's check.

3. The state function for the output context (after the verifier sends r) is the predicate that $s(0) + s(1) = d$ and $p(r) = s(r)$.

Seen in this light, it should be clear that the simpler protocol satisfies round-by-round soundness.

In fact, we can break down this simpler protocol even more: consider the two sub-protocols that each consists of a single message. Then the intermediate state function is the same as the predicate on the intermediate context, and is given in a "strongest post-condition" style where it incorporates the verifier's check along with the initial predicate. We can also view the final state function as a form of "canonical" post-condition, that is implied by the previous predicate except with small probability.

3.3 The Spartan Protocol

3.4 The Ligero Polynomial Commitment Scheme

Chapter 4

Commitment Schemes

4.1 Definitions

4.2 Merkle Trees

Chapter 5

Supporting Theories

5.1 Polynomials

This section contains facts about polynomials that are used in the rest of the library, and also definitions for computable representations of polynomials.

Definition 19 (Multilinear Extension)**.**

Theorem 20 (Multilinear Extension is Unique)**.**

We note that the Schwartz-Zippel Lemma is already in Mathlib.

Theorem 21 (Schwartz-Zippel Lemma)**.**

We also define the type of computable univariate & multilinear polynomials using arrays to represent their coefficients (or dually, their evaluations at given points).

Definition 22 (Computable Univariate Polynomials)**.**

Definition 23 (Computable Multilinear Polynomials)**.**

5.2 Coding Theory

This section contains definitions and theorems about coding theory as they are used in the rest of the library.

Definition 24 (Code Distance)**.**

Definition 25 (Distance from a Code)**.**

Definition 26 (Generator Matrix)**.**

Definition 27 (Parity Check Matrix)**.**

Definition 28 (Interleaved Code)**.**

Definition 29 (Reed-Solomon Code)**.**

Definition 30 (Proximity Measure)**.**

Definition 31 (Proximity Gap)**.**

5.3 The VCVio Library

This library provides a formal framework for reasoning about computations that make *oracle queries*. Many cryptographic primitives and interactive protocols use oracles to model (or simulate) external functionality such as random responses, coin flips, or more structured queries. The VCVio library "lifts" these ideas into a setting where both the abstract specification and concrete simulation of oracles may be studied, and their probabilistic behavior analyzed.

The main ingredients of the library are as follows:

Definition 32 (Specification of Oracles)**.** An oracle specification describes a collection of available oracles, each with its own input and output types. Formally, it's given by an indexed family where each oracle is specified by:

- A domain type (what inputs it accepts)
- A range type (what outputs it can produce)

The indexing allows for potentially infinite collections of oracles, and the specification itself is agnostic to how the oracles actually behave - it just describes their interfaces.

Some examples of oracle specifications (and their intended behavior) are as follows:

- `emptySpec`: Represents an empty set of oracles
- `singletonSpec`: Represents a single oracle available on a singleton index
- `coinSpec`: A coin flipping oracle that produces a random Boolean value
- `unifSpec`: A family of oracles that for every natural number $n \in \mathbb{N}$ chooses uniformly from the set $\{0, \ldots, n\}$.

We often require extra properties on the domains and ranges of oracles. For example, we may require that the domains and ranges come equipped with decidable equality or finiteness properties .

Definition 33 (Oracle Computation)**.** An oracle computation represents a program that can make oracle queries. It can:

- Return a pure value without making any queries (via `pure`)
- Make an oracle query and continue with the response (via `queryBind`)

- Signal failure (via `failure`)

The formal implementation uses a free monad on the inductive type of oracle queries wrapped in an option monad transformer (i.e. `OptionT(FreeMonad(OracleQuery spec))`).

Definition 34 (Handling Oracle Queries)**.** To actually run oracle computations, we need a way to handle (or implement) the oracle queries. An oracle implementation consists a mapping from oracle queries to values in another monad. Depending on the monad, this may allow for various interpretations of the oracle queries.

Definition 35 (Probabilistic Semantics of Oracle Computations)**.** We can view oracle computations as probabilistic programs by considering what happens when oracles respond uniformly at random. This gives rise to a probability distribution over possible outputs (including the possibility of failure). The semantics maps each oracle query to a uniform distribution over its possible responses.

Once we have mapped an oracle computation to a probability distribution, we can define various associated probabilities, such as the probability of failure, or the probability of the output satisfying a given predicate (assuming it does not fail).

Definition 36 (Simulating Oracle Queries with Other Oracles)**.** We can simulate complex oracles using simpler ones by providing a translation mechanism. A simulation oracle specifies how to implement queries in one specification using computations in another specification, possibly maintaining additional state information during the simulation.

Definition 37 (Logging & Caching Oracle Queries)**.** Using the simulation framework, we can add logging and caching behaviors to oracle queries:

- Logging records all queries made during a computation
- Caching remembers query responses and reuses them for repeated queries

These are implemented as special cases of simulation oracles.

Definition 38 (Random Oracle)**.** A random oracle is implemented as a caching oracle that uses lazy sampling:

- On first query: generates a uniform random response and caches it
- On repeated queries: returns the cached response

Chapter 6

References