Formally Verified Arguments of Knowledge in Lean

June 8, 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). We will define its building blocks, and various security properties.

2.1.1 Format

An **(interactive) oracle reduction (IOR)** is an interactive protocol between two parties, a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$. In ArkLib, IORs are defined 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 *(oracle) context*) at the beginning of the protocol. These inputs are classified as follows:
   - *Public inputs* (or *statement*) $\mathtt{x}$: available to both parties;
   - *Private inputs* (or *witness*) $\mathtt{w}$: available only to the prover;
   - *Oracle inputs* (or *oracle statement*) $\mathtt{ox}$: the underlying data is available to the prover, but it's only exposed as an oracle to the verifier. See Theorem 2 for more information.
   - *Shared oracle* $\mathcal{O}$: the oracle is available to both parties via an interface; in most cases, it is either empty, a probabilistic sampling oracle, a random oracle, or a group oracle (for the Algebraic Group Model). See Section 5.3 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).

   Currently, in the oracle reduction setting, we *only* allow messages sent to be available through oracle interfaces. In the (non-oracle) reduction setting, all messages are available directly. Future extensions may allow for mixed visibility for prover's messages.

5. $\mathcal{V}$ is assumed to be *public-coin*, meaning that its challenges are chosen uniformly at random from the finite type corresponding to that round, and it uses no randomness otherwise, except from those coming from the shared oracle.

6. At the end of the protocol, the prover and verifier outputs a new (oracle) context, which consists of:

   - The verifier takes in the input statement and the challenges, performs an *oracle* computation on the input oracle statements and the oracle messages, and outputs a new output statement.

     The verifier also outputs the new oracle statement in an implicit manner, by specifying a subset of the input oracle statements & the oracle messages. Future extensions may allow for more flexibility in specifying output oracle statements (i.e. not just a subset, but a linear combination, or any other function).
   - The prover takes in some final private state (maintained during protocol execution), and outputs a new output statement, new output oracle statement, and new output witness.

Remark 1 (Literature Comparison)**.** In the literature, our definition corresponds to the notion of *functional* IORs. Historically, (vector) IOPs were the first notion to be introduced by [3]; these are IORs where the output statement is true/false, all oracle statements and messages are vectors over some alphabet Σ, and the oracle interfaces are for querying specific positions in the vector. More recent works have considered other oracle interfaces, e.g., polynomial oracles [7, 5], generalized proofs to reductions [9, 4, 6, 2], and considered general oracle interfaces [1]. Most of the IOP theory has been distilled in the textbook [8].

We have not seen any work that considers our most general setting, of IORs with arbitrary oracle interfaces.

We now go into more details on these objects, and how they are represented in Lean. Our description will aim to be as close as possible to the Lean code, and hence may differ somewhat from "mainstream" mathematical & cryptographic notation.

Definition 2 (Oracle Interface)**.** An oracle interface for an underlying data type D consists of the following:

- A type Q for queries to the oracle,
- A type R for responses from the oracle,
- A function $\mathsf{oracle} : \mathsf{D} \to \mathsf{Q} \to \mathsf{R}$ that specifies the oracle's behavior given the underlying data and a query.

See `OracleInterface.lean` for common instances of OracleInterface.

Definition 3 (Context)**.** In an (oracle) reduction, its *(oracle) context* consists of a statement type, a witness type, and (in the oracle case) an indexed list of oracle statement types.

Currently, we do not abstract out / bundle the context as a separate structure, but rather specifies the types explicitly. This may change in the future.

Definition 4 (Protocol Specification)**.** A protocol specification for an n-message (oracle) reduction, is an element of the following type:

$$\mathsf{ProtocolSpec}\ n := \mathsf{Fin}\ n \to \mathsf{Direction} \times \mathsf{Type}.$$

In the above, $\mathsf{Direction} := \{\mathsf{P}{\to}\mathsf{V}, \mathsf{V}{\to}\mathsf{P}\}$ is the type of possible directions of messages, and $\mathsf{Fin}\ n := \{i : \mathbb{N} \,//\, i < n\}$ is the type of all natural numbers less than n.

In other words, for each step i of interaction, the protocol specification describes the *direction* of the message sent in that step, i.e., whether it is from the prover or from the verifier. It also describes the *type* of that message.

In the oracle setting, we also expect an oracle interface for each message from the prover to the verifier.

We define some supporting definitions for a protocol specification.

Definition 5 (Protocol Specification Components)**.** Given a protocol spec $\mathsf{pSpec} : \mathsf{ProtocolSpec}\ n$, we define:

- $\mathsf{pSpec.Dir}\ i := (\mathsf{pSpec}\ i).\mathsf{fst}$ extracts the direction of the i-th message.
- $\mathsf{pSpec.Type}\ i := (\mathsf{pSpec}\ i).\mathsf{snd}$ extracts the type of the i-th message.
- $\mathsf{pSpec.MessageIdx} := \{i : \mathsf{Fin}\ n \,//\, \mathsf{pSpec.Dir}\ i = \mathsf{P}{\to}\mathsf{V}\}$ is the subtype of indices corresponding to prover messages.
- $\mathsf{pSpec.ChallengeIdx} := \{i : \mathsf{Fin}\ n \,//\, \mathsf{pSpec.Dir}\ i = \mathsf{V}{\to}\mathsf{P}\}$ is the subtype of indices corresponding to verifier challenges.
- $\mathsf{pSpec.Message}\ i := (i : \mathsf{pSpec.MessageIdx}) \to \mathsf{pSpec.Type}\ i.\mathsf{val}$ is an indexed family of message types in the protocol.
- $\mathsf{pSpec.Challenge}\ i := (i : \mathsf{pSpec.ChallengeIdx}) \to \mathsf{pSpec.Type}\ i.\mathsf{val}$ is an indexed family of challenge types in the protocol.

Definition 6 (Protocol Transcript)**.** Given protocol specification $\mathsf{pSpec} : \mathsf{ProtocolSpec}\ n$, we define:

- A *transcript* up to round $k : \mathsf{Fin}\ (n+1)$ is an element of type

  $$\mathsf{Transcript}\ k\ \mathsf{pSpec} := (i : \mathsf{Fin}\ k) \to \mathsf{pSpec.Type}\ (\uparrow i : \mathsf{Fin}\ n)$$

  where $\uparrow i : \mathsf{Fin}\ n$ denotes casting $i : \mathsf{Fin}\ k$ to $\mathsf{Fin}\ n$ (valid since $k \leq n+1$).
- A *full transcript* is $\mathsf{FullTranscript\ pSpec} := (i : \mathsf{Fin}\ n) \to \mathsf{pSpec.Type}\ i$.
- The type of all *messages* from prover to verifier is

  $$\mathsf{pSpec.Messages} := \prod_{i : \mathsf{pSpec.MessageIdx}} \mathsf{pSpec.Message}\ i$$

- The type of all *challenges* from verifier to prover is

$$\mathsf{pSpec.Challenges} := \prod_{i:\mathsf{pSpec.ChallengeIdx}} \mathsf{pSpec.Challenge}\ i$$

Remark 7 (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 8 (Type Signature of a Prover)**.** A prover $\mathcal{P}$ in a reduction consists of the following components:

- **Prover State**: A family of types $\mathsf{PrvState} : \mathsf{Fin}(n+1) \to \mathsf{Type}$ representing the prover's internal state at each round of the protocol.
- **Input Processing**: A function

  $$\mathsf{input} : \mathsf{StmtIn} \to \mathsf{WitIn} \to \mathsf{PrvState}(0)$$

  that initializes the prover's state from the input statement and witness.
- **Message Sending**: For each message index $i : \mathsf{pSpec.MessageIdx}$, a function

  $$\mathsf{sendMessage}_i : \mathsf{PrvState}(i.\mathsf{val.castSucc}) \to \mathsf{OracleComp}(\mathsf{oSpec}, \mathsf{pSpec.Message}(i) \times \mathsf{PrvState}(i.\mathsf{val.succ}))$$

  that generates the message and updates the prover's state.
- **Challenge Processing**: For each challenge index $i : \mathsf{pSpec.ChallengeIdx}$, a function

  $$\mathsf{receiveChallenge}_i : \mathsf{PrvState}(i.\mathsf{val.castSucc}) \to \mathsf{pSpec.Challenge}(i) \to \mathsf{PrvState}(i.\mathsf{val.succ})$$

  that updates the prover's state upon receiving a challenge.
- **Output Generation**: A function

  $$\mathsf{output} : \mathsf{PrvState}(\mathsf{Fin.last}(n)) \to \mathsf{StmtOut} \times \mathsf{WitOut}$$

  that produces the final output statement and witness from the prover's final state.

Definition 9 (Type Signature of an Oracle Prover)**.** An oracle prover is a prover whose input statement includes the underlying data for oracle statements, and whose output includes oracle statements. Formally, it is a prover with input statement type $\mathsf{StmtIn} \times (\forall i : \iota_{\mathsf{si}}, \mathsf{OStmtIn}(i))$ and output statement type $\mathsf{StmtOut} \times (\forall i : \iota_{\mathsf{so}}, \mathsf{OStmtOut}(i))$, where:

- $\mathsf{OStmtIn} : \iota_{\mathsf{si}} \to \mathsf{Type}$ are the input oracle statement types
- $\mathsf{OStmtOut} : \iota_{\mathsf{so}} \to \mathsf{Type}$ are the output oracle statement types

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 10 (Type Signature of a Verifier)**.** A verifier $\mathcal{V}$ in a reduction is specified by a single function:

$$\mathsf{verify} : \mathsf{StmtIn} \to \mathsf{FullTranscript}(\mathsf{pSpec}) \to \mathsf{OracleComp}(\mathsf{oSpec}, \mathsf{StmtOut})$$

This function takes the input statement and the complete transcript of the protocol interaction, and performs an oracle computation (potentially querying the shared oracle oSpec) to produce an output statement.

The verifier is assumed to be *public-coin*, meaning it only sends uniformly random challenges and uses no other randomness beyond what is provided by the shared oracle.

Definition 11 (Type Signature of an Oracle Verifier)**.** An oracle verifier $\mathcal{V}$ consists of the following components:

- **Verification Logic**: A function

  $$\mathsf{verify} : \mathsf{StmtIn} \to \mathsf{pSpec.Challenges} \to \mathsf{OracleComp}(\mathsf{oSpec} +\!\!+_\mathsf{o} ([\mathsf{OStmtIn}]_\mathsf{o} +\!\!+_\mathsf{o} [\mathsf{pSpec.Message}]_\mathsf{o}), \mathsf{StmtOut})$$

  that takes the input statement and verifier challenges, and performs oracle queries to the shared oracle, input oracle statements, and prover messages to produce an output statement.

- **Output Oracle Embedding**: An injective function

  $$\mathsf{embed} : \iota_{\mathsf{so}} \hookrightarrow \iota_{\mathsf{si}} \oplus \mathsf{pSpec.MessageIdx}$$

  that specifies how each output oracle statement is derived from either an input oracle statement or a prover message.

- **Type Compatibility**: A proof term

  $$\mathsf{hEq} : \forall i : \iota_{\mathsf{so}}, \mathsf{OStmtOut}(i) = \begin{cases} \mathsf{OStmtIn}(j) & \text{if } \mathsf{embed}(i) = \mathsf{inl}(j) \\ \mathsf{pSpec.Message}(k) & \text{if } \mathsf{embed}(i) = \mathsf{inr}(k) \end{cases}$$

  ensuring that output oracle statement types match their sources.

This design ensures that output oracle statements are always a subset of the available input oracle statements and prover messages.

Definition 12 (Oracle Verifier to Verifier Conversion)**.** An oracle verifier can be converted to a standard verifier through a natural simulation process. The key insight is that while an oracle verifier only has oracle access to certain data (input oracle statements and prover messages), a standard verifier can be given the actual underlying data directly.

The conversion works as follows: when the oracle verifier needs to make an oracle query to some data, the converted verifier can respond to this query immediately using the actual underlying data it possesses. This is accomplished through the `OracleInterface` type class, which specifies for each data type how to respond to queries given the underlying data.

Specifically, given an oracle verifier $\mathcal{V}_{\text{oracle}}$:

- The converted verifier $\mathcal{V}_{\text{oracle}}$.toVerifier takes as input both the statement *and* the actual underlying data for all oracle statements
- When $\mathcal{V}_{\text{oracle}}$ attempts to query an oracle statement or prover message, the converted verifier uses the corresponding `OracleInterface` instance to compute the response from the actual data
- The output oracle statements are constructed according to the embedding specification, selecting the appropriate subset of input oracle statements and prover messages

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

Definition 13 (Interactive Reduction)**.** An interactive reduction for protocol specification $\mathsf{pSpec} : \mathsf{ProtocolSpec}(n)$ and oracle specification oSpec consists of:

- A **prover** $\mathcal{P} : \mathsf{Prover}(\mathsf{pSpec}, \mathsf{oSpec}, \mathsf{StmtIn}, \mathsf{WitIn}, \mathsf{StmtOut}, \mathsf{WitOut})$
- A **verifier** $\mathcal{V} : \mathsf{Verifier}(\mathsf{pSpec}, \mathsf{oSpec}, \mathsf{StmtIn}, \mathsf{StmtOut})$

The reduction establishes a relationship between input relations on $(\mathsf{StmtIn}, \mathsf{WitIn})$ and output relations on $(\mathsf{StmtOut}, \mathsf{WitOut})$ through the interactive protocol defined by pSpec.

Definition 14 (Interactive Oracle Reduction)**.** An interactive oracle reduction for protocol specification $\mathsf{pSpec} : \mathsf{ProtocolSpec}(n)$ with oracle interfaces for all prover messages, and oracle specification oSpec, consists of:

- An **oracle prover** $\mathcal{P} : \mathsf{OracleProver}(\mathsf{pSpec}, \mathsf{oSpec}, \mathsf{StmtIn}, \mathsf{WitIn}, \mathsf{StmtOut}, \mathsf{WitOut}, \mathsf{OStmtIn}, \mathsf{OStmtOut})$
- An **oracle verifier** $\mathcal{V} : \mathsf{OracleVerifier}(\mathsf{pSpec}, \mathsf{oSpec}, \mathsf{StmtIn}, \mathsf{StmtOut}, \mathsf{OStmtIn}, \mathsf{OStmtOut})$

where:

- $\mathsf{OStmtIn} : \iota_{\mathsf{si}} \to \mathsf{Type}$ are the input oracle statement types with oracle interfaces
- $\mathsf{OStmtOut} : \iota_{\mathsf{so}} \to \mathsf{Type}$ are the output oracle statement types

The oracle reduction allows the verifier to access prover messages and oracle statements only through specified oracle interfaces, enabling more flexible and composable protocol designs.

2.1.2 Execution Semantics

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).

Definition 15 (Prover Execution to Round)**.** The execution of a prover up to round $i : \mathsf{Fin}(n+1)$ is defined inductively:

$$\begin{gathered}\mathsf{Prover.runToRound}(i, \mathsf{stmt}, \mathsf{wit}) := \\ \mathsf{Fin.induction}(\\ \mathsf{pure}(\langle \mathsf{default}, \mathsf{prover.input}(\mathsf{stmt}, \mathsf{wit})\rangle), \\ \mathsf{prover.processRound}, \\ i \\ )\end{gathered}$$

where processRound handles individual rounds by either:

- **Verifier Challenge** ($\mathsf{pSpec.getDir}(j) = \mathsf{V\_to\_P}$): Query for a challenge and update prover state
- **Prover Message** ($\mathsf{pSpec.getDir}(j) = \mathsf{P\_to\_V}$): Generate message via sendMessage and update state

Returns the transcript up to round i and the prover's state after round i.

Definition 16 (Complete Prover Execution)**.** The complete execution of a prover is defined as:

$$\begin{gathered}\mathsf{Prover.run}(\mathsf{stmt}, \mathsf{wit}) := \mathsf{do}\ \{ \\ \langle \mathsf{transcript}, \mathsf{state}\rangle \leftarrow \mathsf{prover.runToRound}(\mathsf{Fin.last}(n), \mathsf{stmt}, \mathsf{wit}) \\ \langle \mathsf{stmtOut}, \mathsf{witOut}\rangle := \mathsf{prover.output}(\mathsf{state}) \\ \mathsf{return}\ \langle \mathsf{stmtOut}, \mathsf{witOut}, \mathsf{transcript}\rangle \\ \}\end{gathered}$$

Returns the output statement, output witness, and complete transcript.

Definition 17 (Verifier Execution)**.** The execution of a verifier is simply the application of its verification function:

$$\mathsf{Verifier.run}(\mathsf{stmt}, \mathsf{transcript}) := \mathsf{verifier.verify}(\mathsf{stmt}, \mathsf{transcript})$$

This takes the input statement and full transcript, and returns the output statement via an oracle computation.

Definition 18 (Oracle Verifier Execution)**.** The execution of an oracle verifier is defined as:

$$\begin{gathered}\mathsf{OracleVerifier.run}(\mathsf{stmt}, \mathsf{oStmtIn}, \mathsf{transcript}) := \mathsf{do}\ \{ \\ \mathsf{f} := \mathsf{simOracle2}(\mathsf{oSpec}, \mathsf{oStmtIn}, \mathsf{transcript.messages}) \\ \mathsf{stmtOut} \leftarrow \mathsf{simulateQ}(\mathsf{f}, \mathsf{verifier.verify}(\mathsf{stmt}, \mathsf{transcript.challenges})) \\ \mathsf{return}\ \mathsf{stmtOut} \\ \}\end{gathered}$$

This simulates the oracle access to input oracle statements and prover messages, then executes the verification logic.

Definition 19 (Interactive Reduction Execution)**.** The execution of an interactive reduction consists of running the prover followed by the verifier:

$$\mathsf{Reduction.run}(\mathsf{stmt}, \mathsf{wit}) := \mathsf{do}\ \{$$
$$\langle \mathsf{prvStmtOut}, \mathsf{witOut}, \mathsf{transcript} \rangle \leftarrow \mathsf{reduction.prover.run}(\mathsf{stmt}, \mathsf{wit})$$
$$\mathsf{stmtOut} \leftarrow \mathsf{reduction.verifier.run}(\mathsf{stmt}, \mathsf{transcript})$$
$$\mathsf{return}\ ((\mathsf{prvStmtOut}, \mathsf{witOut}), \mathsf{stmtOut}, \mathsf{transcript})$$
$$\}$$

Returns both the prover's output (statement and witness) and the verifier's output statement, along with the complete transcript.

Definition 20 (Oracle Reduction Execution)**.** The execution of an interactive oracle reduction is similar to a standard reduction but includes logging of oracle queries:

$$\mathsf{OracleReduction.run}(\mathsf{stmt}, \mathsf{wit}, \mathsf{oStmt}) := \mathsf{do}\ \{$$
$$\langle \langle \mathsf{prvStmtOut}, \mathsf{witOut}, \mathsf{transcript} \rangle, \mathsf{proveQueryLog} \rangle \leftarrow$$
$$(\mathsf{simulateQ}(\mathsf{loggingOracle}, \mathsf{reduction.prover.run}(\langle \mathsf{stmt}, \mathsf{oStmt} \rangle, \mathsf{wit}))).\mathsf{run}$$
$$\langle \mathsf{stmtOut}, \mathsf{verifyQueryLog} \rangle \leftarrow$$
$$(\mathsf{simulateQ}(\mathsf{loggingOracle}, \mathsf{reduction.verifier.run}(\mathsf{stmt}, \mathsf{oStmt}, \mathsf{transcript}))).\mathsf{run}$$
$$\mathsf{return}\ ((\mathsf{prvStmtOut}, \mathsf{witOut}), \mathsf{stmtOut}, \mathsf{transcript}, \mathsf{proveQueryLog}, \mathsf{verifyQueryLog})$$
$$\}$$

Returns the same outputs as a standard reduction, plus logs of all oracle queries made by both the prover and verifier.

2.1.3 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 21 (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 22 (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 23 (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 24 (State Function)**.**

Definition 25 (Round-by-Round Soundness)**.**

Definition 26 (Round-by-Round Knowledge Soundness)**.**

2.2 Composition of Oracle Reductions

In this section, we describe a suite of composition operators for building secure oracle reductions from simpler secure components. In other words, we define a number of definitions that govern how oracle reductions can be composed to form larger reductions, and how the resulting reduction inherits the security properties of the components.

2.2.1 Sequential Composition

Sequential composition allows us to chain together oracle reductions where the output context of one reduction becomes the input context of the next reduction. This is fundamental for building complex protocols from simpler components.

Composition of Protocol Specifications

We begin by defining how to compose protocol specifications and their associated structures.

Definition 27 (Protocol Specification Append)**.** Given two protocol specifications $\mathsf{pSpec}_1 : \mathsf{ProtocolSpec}\ m$ and $\mathsf{pSpec}_2 : \mathsf{ProtocolSpec}\ n$, their sequential composition is:

$$\mathsf{pSpec}_1 +\!\!+_{\mathsf{p}} \mathsf{pSpec}_2 : \mathsf{ProtocolSpec}\ (m + n)$$

Definition 28 (Full Transcript Append)**.** Given full transcripts $T_1 : \mathsf{FullTranscript}\ \mathsf{pSpec}_1$ and $T_2 : \mathsf{FullTranscript}\ \mathsf{pSpec}_2$, their sequential composition is:

$$T_1 +\!\!+_{\mathsf{t}} T_2 : \mathsf{FullTranscript}\ (\mathsf{pSpec}_1 +\!\!+_{\mathsf{p}} \mathsf{pSpec}_2)$$

Composition of Provers and Verifiers

Definition 29 (Prover Append)**.** Given provers $P_1 : \mathsf{Prover}\ \mathsf{pSpec}_1\ \mathsf{oSpec}\ \mathsf{StmtIn}_1\ \mathsf{WitIn}_1\ \mathsf{StmtOut}_1\ \mathsf{WitOut}_1$ and $P_2 : \mathsf{Prover}\ \mathsf{pSpec}_2\ \mathsf{oSpec}\ \mathsf{StmtOut}_1\ \mathsf{WitOut}_1\ \mathsf{StmtOut}_2\ \mathsf{WitOut}_2$, their sequential composition is:

$$P_1.\mathsf{append}\ P_2 : \mathsf{Prover}\ (\mathsf{pSpec}_1 +\!\!+_{\mathsf{p}} \mathsf{pSpec}_2)\ \mathsf{oSpec}\ \mathsf{StmtIn}_1\ \mathsf{WitIn}_1\ \mathsf{StmtOut}_2\ \mathsf{WitOut}_2$$

The composed prover works by:

- Running P_1 on the input context to produce an intermediate context
- Using this intermediate context as input to P_2
- Outputting the final context from P_2

Definition 30 (Verifier Append)**.** Given verifiers $V_1 : \mathsf{Verifier}\ \mathsf{pSpec}_1\ \mathsf{oSpec}\ \mathsf{StmtIn}_1\ \mathsf{StmtOut}_1$ and $V_2 : \mathsf{Verifier}\ \mathsf{pSpec}_2\ \mathsf{oSpec}\ \mathsf{StmtOut}_1\ \mathsf{StmtOut}_2$, their sequential composition is:

$$V_1.\mathsf{append}\ V_2 : \mathsf{Verifier}\ (\mathsf{pSpec}_1 +\!\!+_{\mathsf{p}} \mathsf{pSpec}_2)\ \mathsf{oSpec}\ \mathsf{StmtIn}_1\ \mathsf{StmtOut}_2$$

The composed verifier first runs V_1 on the first part of the transcript, then runs V_2 on the second part using the intermediate statement from V_1.

Definition 31 (Reduction Append)**.** Sequential composition of reductions combines the corresponding provers and verifiers:

$$R_1.\mathsf{append}\ R_2 : \mathsf{Reduction}\ (\mathsf{pSpec}_1 +\!\!+_{\mathsf{p}} \mathsf{pSpec}_2)\ \mathsf{oSpec}\ \mathsf{StmtIn}_1\ \mathsf{WitIn}_1\ \mathsf{StmtOut}_2\ \mathsf{WitOut}_2$$

Definition 32 (Oracle Reduction Append)**.** Sequential composition extends naturally to oracle reductions by composing the oracle provers and oracle verifiers.

General Sequential Composition

For composing an arbitrary number of reductions, we provide a general composition operation.

Definition 33 (General Protocol Specification Composition)**.** Given a family of protocol specifications $\mathsf{pSpec} : \forall i : \mathsf{Fin}(m+1), \mathsf{ProtocolSpec}\ (n\ i)$, their composition is:

$$\mathsf{compose}\ m\ n\ \mathsf{pSpec} : \mathsf{ProtocolSpec}\ (\sum_i n\ i)$$

Definition 34 (General Prover Composition)**.**

Definition 35 (General Verifier Composition)**.**

Definition 36 (General Reduction Composition)**.**

Security Properties of Sequential Composition

The key insight is that security properties are preserved under sequential composition.

Theorem 37 (Completeness Preservation under Append)**.** *If reductions R_1 and R_2 satisfy completeness with compatible relations and respective errors ϵ_1 and ϵ_2, then their sequential composition $R_1.\mathsf{append}\ R_2$ satisfies completeness with error $\epsilon_1 + \epsilon_2$.*

Theorem 38 (Perfect Completeness Preservation under Append)**.** *If reductions R_1 and R_2 satisfy perfect completeness with compatible relations, then their sequential composition also satisfies perfect completeness.*

Theorem 39 (Soundness Preservation under Append)**.** *If verifiers V_1 and V_2 satisfy soundness with respective errors ϵ_1 and ϵ_2, then their sequential composition satisfies soundness with error $\epsilon_1 + \epsilon_2$.*

Theorem 40 (Knowledge Soundness Preservation under Append)**.** *If verifiers V_1 and V_2 satisfy knowledge soundness with respective errors ϵ_1 and ϵ_2, then their sequential composition satisfies knowledge soundness with error $\epsilon_1 + \epsilon_2$.*

Theorem 41 (Round-by-Round Soundness Preservation under Append)**.** *If verifiers V_1 and V_2 satisfy round-by-round soundness, then their sequential composition also satisfies round-by-round soundness.*

Theorem 42 (Round-by-Round Knowledge Soundness Preservation under Append)**.** *If verifiers V_1 and V_2 satisfy round-by-round knowledge soundness, then their sequential composition also satisfies round-by-round knowledge soundness.*

Similar preservation theorems hold for the general composition of multiple reductions:

Theorem 43 (General Completeness Preservation)**.**

Theorem 44 (General Soundness Preservation)**.**

Theorem 45 (General Knowledge Soundness Preservation)**.**

2.2.2 Lifting Contexts

Another essential tool for modular oracle reductions is the ability to adapt reductions from one context to another. This allows us to apply reductions designed for simple contexts to more complex scenarios.

Context Lenses

The fundamental abstraction for context adaptation is a *context lens*, which provides bidirectional mappings between outer and inner contexts.

Definition 46 (Statement Lens)**.** A statement lens between outer context types $(\mathsf{StmtIn}_{\mathsf{outer}}, \mathsf{StmtOut}_{\mathsf{outer}})$ and inner context types $(\mathsf{StmtIn}_{\mathsf{inner}}, \mathsf{StmtOut}_{\mathsf{inner}})$ consists of:

- $\mathsf{projStmt} : \mathsf{StmtIn}_{\mathsf{outer}} \to \mathsf{StmtIn}_{\mathsf{inner}}$ (projection to inner context)
- $\mathsf{liftStmt} : \mathsf{StmtIn}_{\mathsf{outer}} \times \mathsf{StmtOut}_{\mathsf{inner}} \to \mathsf{StmtOut}_{\mathsf{outer}}$ (lifting back to outer context)

Definition 47 (Witness Lens)**.** A witness lens between outer witness types $(\mathsf{WitIn}_{\mathsf{outer}}, \mathsf{WitOut}_{\mathsf{outer}})$ and inner witness types $(\mathsf{WitIn}_{\mathsf{inner}}, \mathsf{WitOut}_{\mathsf{inner}})$ consists of:

- $\mathsf{projWit} : \mathsf{WitIn}_{\mathsf{outer}} \to \mathsf{WitIn}_{\mathsf{inner}}$ (projection to inner context)
- $\mathsf{liftWit} : \mathsf{WitIn}_{\mathsf{outer}} \times \mathsf{WitOut}_{\mathsf{inner}} \to \mathsf{WitOut}_{\mathsf{outer}}$ (lifting back to outer context)

Definition 48 (Context Lens)**.** A context lens combines a statement lens and a witness lens for adapting complete reduction contexts.

Definition 49 (Oracle Context Lens)**.** For oracle reductions, we additionally need lenses for oracle statements that can simulate oracle access between contexts.

Lifting Reductions

Given a context lens, we can lift reductions from inner contexts to outer contexts.

Definition 50 (Prover Context Lifting)**.** Given a prover P for the inner context and a context lens, the lifted prover works by:

- Projecting the outer input to the inner context
- Running the inner prover
- Lifting the output back to the outer context

Definition 51 (Verifier Context Lifting)**.**

Definition 52 (Reduction Context Lifting)**.**

Conditions for Security Preservation

For lifting to preserve security properties, the context lens must satisfy certain conditions.

Definition 53 (Completeness-Preserving Context Lens)**.** A context lens preserves completeness if it maintains relation satisfaction under projection and lifting.

Definition 54 (Soundness-Preserving Statement Lens)**.** A statement lens preserves soundness if it maps invalid statements to invalid statements.

Definition 55 (RBR Soundness-Preserving Statement Lens)**.** For round-by-round soundness, we need a slightly relaxed soundness condition.

Definition 56 (Knowledge Soundness-Preserving Context Lens)**.** A context lens preserves knowledge soundness if it maintains witness extractability.

Security Preservation Theorems for Context Lifting

Theorem 57 (Completeness Preservation under Context Lifting)**.** *If a reduction satisfies completeness and the context lens is completeness-preserving, then the lifted reduction also satisfies completeness.*

Theorem 58 (Soundness Preservation under Context Lifting)**.** *If a verifier satisfies soundness and the statement lens is soundness-preserving, then the lifted verifier also satisfies soundness.*

Theorem 59 (Knowledge Soundness Preservation under Context Lifting)**.** *If a verifier satisfies knowledge soundness and the context lens is knowledge soundness-preserving, then the lifted verifier also satisfies knowledge soundness.*

Theorem 60 (RBR Soundness Preservation under Context Lifting)**.** *If a verifier satisfies round-by-round soundness and the statement lens is RBR soundness-preserving, then the lifted verifier also satisfies round-by-round soundness.*

Theorem 61 (RBR Knowledge Soundness Preservation under Context Lifting)**.** *If a verifier satisfies round-by-round knowledge soundness and the context lens is knowledge soundness-preserving, then the lifted verifier also satisfies round-by-round knowledge soundness.*

Extractors and State Functions

Context lifting also applies to extractors and state functions used in knowledge soundness and round-by-round soundness.

Definition 62 (Straightline Extractor Lifting)**.**

Definition 63 (Round-by-Round Extractor Lifting)**.**

Definition 64 (State Function Lifting)**.**

These composition and lifting operators provide the essential building blocks for constructing complex oracle reductions from simpler components while preserving their security properties.

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

This section documents our formalization of the sum-check protocol. We first describe the sum-check protocol as it is typically described in the literature, and then present a modular description that maximally relies on our general oracle reduction framework.

3.2.1 Standard Description

Protocol Parameters

The sum-check protocol is parameterized by the following:

- R: the underlying ring (for soundness, required to be finite and an integral domain)
- $n \in \mathbb{N}$: the number of variables (and the number of rounds for the protocol)
- $d \in \mathbb{N}$: the individual degree bound for the polynomial
- $\mathcal{D} : \{0, 1, \ldots, m-1\} \hookrightarrow R$: the set of m evaluation points for each variable, represented as an injection. The image of $\mathcal{D}$ as a finite subset of R is written as $\text{Image}(\mathcal{D})$.
- $\mathcal{O}$: the set of underlying oracles (e.g., random oracles) that may be needed for other reductions. However, the sum-check protocol itself does *not* use any oracles.

Input and Output Statements

For the standard description of the sum-check protocol, we specify the complete input and output data:

Input Statement. The claimed sum $T \in R$.

Input Oracle Statement. The polynomial $P \in R[X_0, X_1, \ldots, X_{n-1}]_{\leq d}$ of n variables with bounded individual degrees d.

Input Witness. None (the unit type).

Input Relation. The sum-check relation:

$$\sum_{x \in (\text{Image}(\mathcal{D}))^n} P(x) = T$$

Output Statement. The claimed evaluation $e \in R$ and the challenge vector $(r_0, r_1, \ldots, r_{n-1}) \in R^n$.

Output Oracle Statement. The same polynomial $P \in R[X_0, X_1, \ldots, X_{n-1}]_{\leq d}$.

Output Witness. None (the unit type).

Output Relation. The evaluation relation:

$$P(r_0, r_1, \ldots, r_{n-1}) = e$$

Protocol Description

The sum-check protocol proceeds in n rounds of interaction between the prover and verifier. The protocol reduces the claim that a multivariate polynomial P sums to a target value T over the domain $(\text{Image}(\mathcal{D}))^n$ to the claim that P evaluates to a specific value e at a random point $(r_0, r_1, \ldots, r_{n-1})$.

In each round, the prover sends a univariate polynomial of bounded degree, and the verifier responds with a random challenge. The verifier performs consistency checks by querying the polynomial oracle at specific evaluation points. After n rounds, the protocol terminates with an output statement asserting that $P(r_0, r_1, \ldots, r_{n-1}) = e$, where the challenges $(r_0, r_1, \ldots, r_{n-1})$ are the random values chosen by the verifier during the protocol execution.

The protocol is described as an oracle reduction, where the polynomial P is accessed only through evaluation queries rather than being explicitly represented.

Security Properties

We prove the following security properties for the sum-check protocol:

Theorem 65 (Perfect Completeness)**.** *The sum-check protocol satisfies perfect completeness. That is, for any valid input statement and oracle statement satisfying the input relation, the protocol accepts with probability 1.*

Theorem 66 (Knowledge Soundness)**.** *The sum-check protocol satisfies knowledge soundness. The soundness error is bounded by $n \cdot d/|R|$, where n is the number of rounds, d is the degree bound, and $|R|$ is the size of the field.*

Theorem 67 (Round-by-Round Knowledge Soundness)**.** *The sum-check protocol satisfies round-by-round knowledge soundness with respect to an appropriate state function (to be specified). Each round maintains the security properties compositionally, allowing for modular security analysis.*

Implementation Notes

Our formalization includes several important implementation considerations:

Oracle Reduction Level. This description of the sum-check protocol stays at the **oracle reduction** level, describing the protocol before being compiled with concrete cryptographic primitives such as polynomial commitment schemes for P. The oracle model allows us to focus on the logical structure and security properties of the protocol without being concerned with the specifics of how polynomial evaluations are implemented or verified.

Abstract Protocol Description. The protocol description above does not consider implementation details and optimizations that would be necessary in practice. For instance, we do not address computational efficiency, concrete polynomial representations, or specific algorithms for polynomial evaluation. This abstraction allows us to establish the fundamental security properties that any concrete implementation must preserve.

Degree Constraints. To represent sum-check as a series of Interactive Oracle Reductions (IORs), we implicitly constrain the degree of the polynomials via using subtypes such as $R[X]_{\leq d}$ and appropriate multivariate polynomial degree bounds. This is necessary because the oracle verifier only gets oracle access to evaluating the polynomials, but does not see the polynomials in the clear.

Polynomial Commitments. When this protocol is compiled to an interactive proof (rather than an oracle reduction), the corresponding polynomial commitment schemes will enforce that the declared degree bounds hold, by letting the (non-oracle) verifier perform explicit degree checks.

Formalization Alignment. TODO: Align the sum-check protocol formalization in Lean to use n variables and n rounds (as in this standard description) rather than $n+1$ variables and $n+1$ rounds. This should be achievable by refactoring the current implementation to better match the standard presentation.

Future Extensions

Several generalizations are considered for future work:

- **Variable Degree Bounds:** Generalize to $d : \{0, 1, \ldots, n+1\} \to \mathbb{N}$ and $\mathcal{D} : \{0, 1, \ldots, n+1\} \to (\{0, 1, \ldots, m-1\} \hookrightarrow R)$, allowing different degree bounds and summation domains for each variable.
- **Restricted Challenge Domains:** Generalize the challenges to come from suitable subsets of R (e.g., subtractive sets), rather than the entire domain. This modification is used in lattice-based protocols.
- **Module-based Sum-check:** Extend to sum-check over modules instead of just rings. This would require extending multivariate polynomial evaluation to modules, defining something like evalModule : $(R^n \to M) \to R[X_0, \ldots, X_{n-1}] \to M$ where M is an R-module.

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 $s_0(r_0) = c$.

3.2.2 Modular Description

Round-by-Round Analysis

Our modular approach breaks down the sum-check protocol into individual rounds, each of which can be analyzed as a two-message Interactive Oracle Reduction. This decomposition allows us to prove security properties compositionally and provides a more granular understanding of the protocol's structure.

Round-Specific Statements. For the i-th round, where $i \in \{0, 1, \ldots, n\}$, the statement contains:

- target $\in R$: the target value for sum-check at this round
- challenges $\in R^i$: the list of challenges sent from the verifier to the prover in previous rounds

The oracle statement remains the same polynomial $P \in R[X_0, X_1, \ldots, X_{n-1}]_{\leq d}$.

Round-Specific Relations. The sum-check relation for the i-th round checks that:

$$\sum_{x \in (\text{Image}(\mathcal{D}))^{n-i}} P(\text{challenges}, x) = \text{target}$$

Note that when $i = n$, this becomes the output statement of the sum-check protocol, checking that $P(\text{challenges}) = \text{target}$.

Individual Round Protocol

For $i = 0, 1, \ldots, n-1$, the i-th round of the sum-check protocol consists of the following:

Step 1: Prover's Message. The prover sends a univariate polynomial $p_i \in R[X]_{\leq d}$ of degree at most d. If the prover is honest, then:

$$p_i(X) = \sum_{x \in (\text{Image}(\mathcal{D}))^{n-i}} P(\text{challenges}_0, \ldots, \text{challenges}_{i-1}, X, x)$$

Here, $P(\text{challenges}_0, \ldots, \text{challenges}_{i-1}, X, x)$ is the polynomial P evaluated at the concatenation of:

- the prior challenges $\text{challenges}_0, \ldots, \text{challenges}_{i-1}$
- the i-th variable as the new indeterminate X
- the remaining values $x \in (\text{Image}(\mathcal{D}))^{n-i}$

In the oracle protocol, this polynomial p_i is turned into an oracle for which the verifier can query evaluations at arbitrary points.

Step 2: Verifier's Challenge. The verifier sends the i-th challenge r_i sampled uniformly at random from R.

Step 3: Verifier's Check. The (oracle) verifier performs queries for the evaluations of p_i at all points in $\text{Image}(\mathcal{D})$, and checks that:

$$\sum_{x \in \text{Image}(\mathcal{D})} p_i(x) = \text{target}$$

If the check fails, the verifier outputs `failure`. Otherwise, it outputs a statement for the next round as follows:

- target is updated to $p_i(r_i)$
- challenges is updated to the concatenation of the previous challenges and r_i

Single Round Security Analysis

Definition 68 (Single Round Protocol)**.** The i-th round of sum-check consists of:

1. **Input:** A statement containing target value and prior challenges, along with an oracle for the multivariate polynomial
2. **Prover's message:** A univariate polynomial $p_i \in R[X]_{\leq d}$
3. **Verifier's challenge:** A random element $r_i \leftarrow R$
4. **Output:** An updated statement with new target $p_i(r_i)$ and extended challenges

Theorem 69 (Single Round Completeness)**.** *Each individual round of the sum-check protocol is perfectly complete.*

Theorem 70 (Single Round Soundness)**.** *Each individual round of the sum-check protocol is sound with error probability at most $d/|R|$, where d is the degree bound and $|R|$ is the size of the field.*

Theorem 71 (Round-by-Round Knowledge Soundness)**.** *The sum-check protocol satisfies round-by-round knowledge soundness. Each individual round can be analyzed independently, and the soundness error in each round is bounded by $d/|R|$.*

Virtual Protocol Decomposition

We now proceed to break down this protocol into individual messages, 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:

Definition 72. 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 $r \leftarrow R$.

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 $\sum_{x \in (\text{Image}(\mathcal{D}))^{n-i-1}} P(r_0, \ldots, r_{i-1}, X, x)$.
- The scalar d is instantiated as T 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.

Theorem 73. *The virtual sum-check round protocol is sound.*

Note that there is no witness, so knowledge soundness follows trivially from soundness.

Theorem 74. *The virtual sum-check round protocol is knowledge sound.*

Moreover, we can define the following state function for the simpler protocol

Definition 75 (State Function)**.** The state function for the virtual sum-check round protocol is given by:

1. The initial state function 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.

Theorem 76. *The virtual sum-check round protocol is round-by-round sound.*

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 77 (Multilinear Extension)**.**

Theorem 78 (Multilinear Extension is Unique)**.**

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

Theorem 79 (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 80 (Computable Univariate Polynomials)**.**

Definition 81 (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 82 (Code Distance)**.**

Definition 83 (Distance from a Code)**.**

Definition 84 (Generator Matrix)**.**

Definition 85 (Parity Check Matrix)**.**

Definition 86 (Interleaved Code)**.**

Definition 87 (Reed-Solomon Code)**.**

Definition 88 (Proximity Measure)**.**

Definition 89 (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 90 (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 91 (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 92 (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 93 (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 94 (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 95 (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 96 (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

Bibliography

[1] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. Whir: Reed–solomon proximity testing with super-fast verification. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 214–243. Springer, 2025.

[2] Anubhav Baweja, Pratyush Mishra, Tushar Mopuri, and Matan Shtepel. Fics and facs: Fast iopps and accumulation via code-switching. *Cryptology ePrint Archive*, 2025.

[3] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*, pages 31–60. Springer, 2016.

[4] Benedikt Bünz, Alessandro Chiesa, Giacomo Fenzi, and William Wang. Linear-time accumulation schemes. *Cryptology ePrint Archive*, 2025.

[5] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 677–706. Springer, 2020.

[6] Benedikt Bünz, Pratyush Mishra, Wilson Nguyen, and William Wang. Arc: Accumulation for reed–solomon codes. *Cryptology ePrint Archive*, 2024.

[7] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 738–768. Springer, 2020.

[8] Alessandro Chiesa and Eylon Yogev. *Building Cryptographic Proofs from Hash Functions*. 2024.

[9] Abhiram Kothapalli and Bryan Parno. Algebraic reductions of knowledge. In *Annual International Cryptology Conference*, pages 669–701. Springer, 2023.