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*) $\mathbb{x}$: available to both parties;
   - *Private inputs* (or *witness*) $\mathbb{w}$: available only to the prover;
   - *Oracle inputs* (or *oracle statement*) $\mathbb{O}\mathbb{x}$: 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 specification $\mathsf{pSpec} : \mathsf{ProtocolSpec}\ n$, 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 $k \le n+1$).

- A *full transcript* is $\mathsf{FullTranscript}\ \mathsf{pSpec} := (i : \mathsf{Fin}\ n) \to \mathsf{pSpec.Type}\ i$.

- 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, 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.

Definition 9 (Type Signature of an Oracle Prover)**.**

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 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.

Definition 11 (Type Signature of an Oracle Verifier)**.**

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

Definition 12 (Interactive Reduction)**.** An interactive reduction is a combination of a type signature `ProtocolSpec`, a prover for `ProtocolSpec`, and a verifier for `ProtocolSpec`.

Definition 13 (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.

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 14 (Execution of an Oracle Reduction)**.**

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 15 (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 16 (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 17 (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 18 (State Function)**.**

Definition 19 (Round-by-Round Soundness)**.**

Definition 20 (Round-by-Round Knowledge Soundness)**.**

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 $\mathcal{R}_1' \implies \mathcal{R}_2'$.

$$\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\} \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}} : \tau \{\!\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\!\}$$

$$\mathcal{R}_1' \implies \mathcal{R}_1$$

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

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

$$\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\} \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}_1} : \tau \{\!\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\!\}$$

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

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 following rule:

$$\Psi; \Theta; \Sigma \vdash \{\mathcal{R}_1\} \langle \mathcal{P}_1, \mathcal{V}_1, \mathcal{E}_1 \rangle^{\mathcal{O}} : \tau_1 \{\!\!\{\mathcal{R}_2; \mathsf{St}_1; \epsilon_1\}\!\!\}$$

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

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 3.2 for an example.

Definition 21 (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 22 (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.

$$\Psi'; \Theta'; \Sigma' \vdash \{\mathcal{R}_1\} \langle \mathcal{P}, \mathcal{V}, \mathcal{E} \rangle^{\mathcal{O}} : \tau \{\!\!\{\mathcal{R}_2; \mathsf{St}; \epsilon\}\!\!\}$$

$$f : (\Psi, \Theta, \Sigma) \to (\Psi', \Theta', \Sigma')$$

$$g : \Psi' \to \Psi$$

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

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:

8

- 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:

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

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

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:

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

11

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

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

Output Witness. None (the unit type).

Output Relation. The evaluation relation:

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

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 $(\mathrm{Image}(\mathcal{D}))^n$ to the claim that P evaluates to a specific value e at a random point $(r_0, r_1, \dots, r_{n-1})$.

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 $P(r_0, r_1, \dots, r_{n-1}) = e$, where the challenges $(r_0, r_1, \dots, r_{n-1})$ 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 23 (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 24 (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 25 (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 $\text{evalModule} : (R^n \to M) \to R[X_0, \ldots, X_{n-1}] \to M$ 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

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

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 $i \in \{0, 1, \dots, n\}$, 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 $P \in R[X_0, X_1, \dots, X_{n-1}]_{\leq d}$.

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

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

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 $i = 0, 1, \dots, n-1$, 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:

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

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

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

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 26 (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 27 (Single Round Completeness)**.** *Each individual round of the sum-check protocol is perfectly complete.*

Theorem 28 (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 29 (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 30.   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 $\sum_{x \in (\text{Image}(\mathcal{D}))^{n-i-1}} P(r_0, \dots, r_{i-1}, X, x)$.

- 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 31. *The virtual sum-check round protocol is sound.*

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

Theorem 32. *The virtual sum-check round protocol is knowledge sound.*

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

Definition 33 (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 34. *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 35 (Multilinear Extension)**.**

Theorem 36 (Multilinear Extension is Unique)**.**

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

Theorem 37 (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 38 (Computable Univariate Polynomials)**.**

Definition 39 (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 40 (Code Distance)**.**

Definition 41 (Distance from a Code)**.**

Definition 42 (Generator Matrix)**.**

Definition 43 (Parity Check Matrix)**.**

Definition 44 (Interleaved Code)**.**

Definition 45 (Reed-Solomon Code)**.**

Definition 46 (Proximity Measure)**.**

Definition 47 (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 48 (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 $\{0, \dots, n\}$.

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 49 (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 50 (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 51 (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 52 (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 53 (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 54 (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.