# Formally Verified Arguments of Knowledge in Lean

July 1, 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 $\Sigma$, 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 [11, 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 $\mathsf{D}$ consists of the following:

- A type $\mathsf{Q}$ for queries to the oracle,
- A type $\mathsf{R}$ 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 $\mathsf{oSpec}$) 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 $\mathsf{oSpec}$ 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 $\mathsf{pSpec}$.

**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 $\mathsf{oSpec}$, 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)**.** A reduction satisfies **completeness** with error $\epsilon \geq 0$ and with respect to input relation $R_{\text{in}}$ and output relation $R_{\text{out}}$, if for all valid statement-witness pair $(x_{\text{in}}, w_{\text{in}})$ for $R_{\text{in}}$, the execution between the honest prover and the honest verifier will result in a tuple $((x^P_{\text{out}}, w_{\text{out}}), x^V_{\text{out}})$ such that:

- $R_{\text{out}}(x^V_{\text{out}}, w_{\text{out}}) = \text{True}$ (the output statement-witness pair is valid), and
- $x^P_{\text{out}} = x^V_{\text{out}}$ (the output statements are the same from both prover and verifier)

except with probability $\epsilon$.

**Definition 22** (Perfect Completeness)**.** A reduction satisfies **perfect completeness** if it satisfies completeness with error 0. This means that the probability of the reduction outputting a valid statement-witness pair is *exactly* 1 (instead of at least $1 - 0$).

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.

### Extractors

For knowledge soundness, we need to consider different types of extractors that can recover witnesses from malicious provers.

**Definition 23** (Straightline Extractor)**.** A **straightline, deterministic, non-oracle-querying extractor** takes in:

- the output witness $w_{\text{out}}$,
- the initial statement $x_{\text{in}}$,
- the IOR transcript $\tau$,
- the query logs from the prover and verifier

and returns a corresponding initial witness $w_{\text{in}}$.

Note that the extractor does not need to take in the output statement, since it can be derived via re-running the verifier on the initial statement, the transcript, and the verifier's query log.

This form of extractor suffices for proving knowledge soundness of most hash-based IOPs.

**Definition 24** (Round-by-Round Extractor)**.** A **round-by-round extractor** with index $m$ is given:

- the input statement $x_{\text{in}}$,
- a partial transcript of length $m$,
- the prover's query log

and returns a witness to the statement.

Note that the RBR extractor does not need to take in the output statement or witness.

**Definition 25** (Rewinding Extractor)**.** A **rewinding extractor** consists of:

- An extractor state type
- Simulation oracles for challenges and oracle queries for the prover
- A function that runs the extractor with the prover's oracle interface, allowing for calling the prover multiple times

This allows the extractor to rewind the prover to earlier states and try different challenges.

### Adversarial Provers

**Definition 26** (State-Restoration Prover)**.** A **state-restoration prover** is a modified prover that has query access to challenge oracles that can return the $i$-th challenge, for all $i$, given the input statement and the transcript up to that point.

It takes in the input statement and witness, and outputs a full transcript of interaction, along with the output statement and witness.

This models adversaries in the state-restoration setting where challenges can be queried programmably.

### Soundness Definitions

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 27** (Soundness)**.** A reduction satisfies **soundness** with error $\epsilon \geq 0$ and with respect to input language $L_{\text{in}} \subseteq \text{Statement}_{\text{in}}$ and output language $L_{\text{out}} \subseteq \text{Statement}_{\text{out}}$ if:

- for all (malicious) provers with arbitrary types for witness types,
- for all arbitrary input witness,
- for all input statement $x_{\text{in}} \notin L_{\text{in}}$,

the execution between the prover and the honest verifier will result in an output statement $x_{\text{out}} \in L_{\text{out}}$ with probability at most $\epsilon$.

**Definition 28** (Knowledge Soundness)**.** A reduction satisfies **(straightline) knowledge soundness** with error $\epsilon \geq 0$ and with respect to input relation $R_{\text{in}}$ and output relation $R_{\text{out}}$ if:

- there exists a straightline extractor $E$, such that
- for all input statement $x_{\text{in}}$, witness $w_{\text{in}}$, and (malicious) prover,
- if the execution with the honest verifier results in a pair $(x_{\text{out}}, w_{\text{out}})$,
- and the extractor produces some $w'_{\text{in}}$,

then the probability that $(x_{\text{in}}, w'_{\text{in}})$ is not valid for $R_{\text{in}}$ and yet $(x_{\text{out}}, w_{\text{out}})$ is valid for $R_{\text{out}}$ is at most $\epsilon$.

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.

### Round-by-Round Security

To define round-by-round (knowledge) soundness, we need to define the notion of a *state function*. This is a (possibly inefficient) function $\mathsf{StateF}$ 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 29** (State Function)**.** A **(deterministic) state function** for a verifier, with respect to input language $L_{\text{in}}$ and output language $L_{\text{out}}$, consists of a function that maps partial transcripts to boolean values, satisfying:

- For all input statements not in the language, the state function is false for the empty transcript
- If the state function is false for a partial transcript, and the next message is from the prover to the verifier, then the state function is also false for the new partial transcript regardless of the message
- If the state function is false for a full transcript, the verifier will not output a statement in the output language

**Definition 30** (Knowledge State Function)**.** A **knowledge state function** for a verifier, with respect to input relation $R_{\text{in}}$, output relation $R_{\text{out}}$, and intermediate witness types, extends the basic state function to track witness validity throughout the protocol execution. This is used to define round-by-round knowledge soundness.

**Definition 31** (Round-by-Round Soundness)**.** A protocol with verifier $\mathcal{V}$ satisfies **round-by-round soundness** with respect to input language $L_{\text{in}}$, output language $L_{\text{out}}$, and error function $\epsilon : \text{ChallengeIdx} \to \mathbb{R}_{\geq 0}$ if:

- there exists a state function for the verifier and the input/output languages, such that
- for all initial statements $x_{\text{in}} \notin L_{\text{in}}$,
- for all initial witnesses,
- for all provers,
- for all challenge rounds $i$,

the probability that:

- the state function is false for the partial transcript output by the prover
- the state function is true for the partial transcript appended by next challenge (chosen randomly)

is at most $\epsilon(i)$.

**Definition 32** (Round-by-Round Knowledge Soundness)**.** A protocol with verifier $\mathcal{V}$ satisfies **round-by-round knowledge soundness** with respect to input relation $R_{\text{in}}$, output relation $R_{\text{out}}$, and error function $\epsilon : \text{ChallengeIdx} \to \mathbb{R}_{\geq 0}$ if:

- there exists a knowledge state function for the verifier and the languages of the input/output relations,
- there exists a round-by-round extractor,
- for all initial statements,
- for all initial witnesses,
- for all provers,
- for all challenge rounds $i$,

the probability that:

- the extracted witness does not satisfy the input relation
- the state function is false for the partial transcript output by the prover
- the state function is true for the partial transcript appended by next challenge (chosen randomly)

is at most $\epsilon(i)$.

### Extractor Properties

These definitions are highly experimental and may change in the future. The goal is to put some conditions on the extractor in order for prove sequential composition preserves knowledge soundness.

**Definition 33** (Monotone Straightline Extractor)**.** An extractor is **monotone** if its success probability on a given query log is the same as the success probability on any extension of that query log. This property ensures that the extractor's performance does not degrade when given more information.

**Definition 34** (Monotone RBR Extractor)**.** A round-by-round extractor is **monotone** if its success probability on a given query log is the same as the success probability on any extension of that query log.

### Implications Between Security Notions

We have a lattice of security notions, with knowledge and round-by-round being two strengthenings of soundness.

**Theorem 35** (Knowledge Soundness Implies Soundness)**.** *Knowledge soundness with knowledge error $\epsilon < 1$ implies soundness with the same soundness error $\epsilon$, and for the corresponding input and output languages.*

**Theorem 36** (RBR Soundness Implies Soundness)**.** *Round-by-round soundness with error function $\epsilon$ implies soundness with error $\sum_i \epsilon(i)$, where the sum is over all challenge rounds $i$.*

**Theorem 37** (RBR Knowledge Soundness Implies RBR Soundness)**.** *Round-by-round knowledge soundness with error function $\epsilon$ implies round-by-round soundness with the same error function $\epsilon$.*

**Theorem 38** (RBR Knowledge Soundness Implies Knowledge Soundness)**.** *Round-by-round knowledge soundness with error function $\epsilon$ implies knowledge soundness with error $\sum_i \epsilon(i)$, where the sum is over all challenge rounds $i$.*

### Zero-Knowledge

**Definition 39** (Simulator)**.** A **simulator** consists of:

- Oracle simulation capabilities for the shared oracles
- A prover simulation function that takes an input statement and produces a transcript

The simulator should have programming access to the shared oracles and be able to generate transcripts that are indistinguishable from real protocol executions.

**Remark 40** (Zero-Knowledge Definition)**.** We define honest-verifier zero-knowledge as follows: There exists a simulator such that for all (malicious) verifiers, the distributions of transcripts generated by the simulator and the interaction between the verifier and the prover are (statistically) indistinguishable. A full definition will be provided in future versions.

### Oracle-Specific Security

For oracle reductions, the security definitions are analogous to those for standard reductions, but adapted to work with oracle interfaces:

**Definition 41** (Oracle Reduction Completeness)**.** Completeness of an oracle reduction is the same as for non-oracle reductions, but applied to the converted reduction where oracle statements are handled through their interfaces.

**Definition 42** (Oracle Verifier Soundness)**.** Soundness of an oracle verifier is defined by converting it to a standard verifier and applying the standard soundness definition.

**Definition 43** (Oracle Verifier Knowledge Soundness)**.** Knowledge soundness of an oracle verifier is defined by converting it to a standard verifier and applying the standard knowledge soundness definition.

**Definition 44** (Oracle Verifier RBR Soundness)**.** Round-by-round soundness of an oracle verifier is defined by converting it to a standard verifier and applying the standard round-by-round soundness definition.

**Definition 45** (Oracle Verifier RBR Knowledge Soundness)**.** Round-by-round knowledge soundness of an oracle verifier is defined by converting it to a standard verifier and applying the standard round-by-round knowledge soundness definition.

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

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

#### State-Restoration Security

**Definition 46** (State-Restoration Soundness)**. State-restoration soundness** is a security notion where the adversarial prover has access to challenge oracles that can return the $i$-th challenge for any round $i$, given the input statement and the transcript up to that point. This models stronger adversaries in the programmable random oracle model or when challenges can be computed deterministically.

A verifier satisfies state-restoration soundness if for all input statements not in the language, for all witnesses, and for all state-restoration provers, the probability that the verifier outputs a statement in the output language is bounded by the soundness error.

*Note: This definition is currently under development in the Lean formalization.*

**Definition 47** (State-Restoration Knowledge Soundness)**. State-restoration knowledge soundness** extends state-restoration soundness with the requirement that there exists a straightline extractor that can recover valid witnesses from any state-restoration prover that convinces the verifier.

*Note: This definition is currently under development in the Lean formalization.*

## 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 48** (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 49** (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 50** (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 51** (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 52** (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 53** (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 54** (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 55** (General Prover Composition)**.**

**Definition 56** (General Verifier Composition)**.**

**Definition 57** (General Reduction Composition)**.**

### Security Properties of Sequential Composition

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

**Theorem 58** (Completeness Preservation under Append)**.** *If reductions $R_1$ and $R_2$ satisfy completeness with compatible relations and respective errors $\epsilon_1$ and $\epsilon_2$, then their sequential composition $R_1.\mathsf{append}\ R_2$ satisfies completeness with error $\epsilon_1 + \epsilon_2$.*

**Theorem 59** (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 60** (Soundness Preservation under Append)**.** *If verifiers $V_1$ and $V_2$ satisfy soundness with respective errors $\epsilon_1$ and $\epsilon_2$, then their sequential composition satisfies soundness with error $\epsilon_1 + \epsilon_2$.*

**Theorem 61** (Knowledge Soundness Preservation under Append)**.** *If verifiers $V_1$ and $V_2$ satisfy knowledge soundness with respective errors $\epsilon_1$ and $\epsilon_2$, then their sequential composition satisfies knowledge soundness with error $\epsilon_1 + \epsilon_2$.*

**Theorem 62** (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 63** (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 64** (General Completeness Preservation)**.**

**Theorem 65** (General Soundness Preservation)**.**

**Theorem 66** (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 67** (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 68** (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 69** (Context Lens)**.** A context lens combines a statement lens and a witness lens for adapting complete reduction contexts.

**Definition 70** (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 71** (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 72** (Verifier Context Lifting)**.**

**Definition 73** (Reduction Context Lifting)**.**

### Conditions for Security Preservation

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

**Definition 74** (Completeness-Preserving Context Lens)**.** A context lens preserves completeness if it maintains relation satisfaction under projection and lifting.

**Definition 75** (Soundness-Preserving Statement Lens)**.** A statement lens preserves soundness if it maps invalid statements to invalid statements.

**Definition 76** (RBR Soundness-Preserving Statement Lens)**.** For round-by-round soundness, we need a slightly relaxed soundness condition.

**Definition 77** (Knowledge Soundness-Preserving Context Lens)**.** A context lens preserves knowledge soundness if it maintains witness extractability.

### Security Preservation Theorems for Context Lifting

**Theorem 78** (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 79** (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 80** (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 81** (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 82** (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 83** (Straightline Extractor Lifting)**.**

**Definition 84** (Round-by-Round Extractor Lifting)**.**

**Definition 85** (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.

## 2.3 The Fiat-Shamir Transformation

(NOTE: generated by Claude 4 Sonnet, will need to be cleaned up)

The Fiat-Shamir transformation is a fundamental cryptographic technique that converts a public-coin interactive reduction into a non-interactive reduction by replacing verifier challenges with queries to a random oracle. This transformation removes the need for interaction while preserving important security properties under certain assumptions.

In our formalization, the Fiat-Shamir transformation takes an interactive reduction $R$ and produces a non-interactive reduction where the prover computes all messages at once, and the verifier derives the challenges using queries to a hash function (modeled as a random oracle) applied to the statement and the messages up to each challenge round.

### 2.3.1 Oracle Interface for Fiat-Shamir Challenges

The key insight of the Fiat-Shamir transformation is to replace interactive challenges with deterministic computations based on the protocol messages so far.

**Definition 86** (Fiat-Shamir Challenge Oracle Interface)**.** For a protocol specification pSpec and input statement type StmtIn, the Fiat-Shamir challenge oracle interface for the $i$-th challenge is defined as follows:

- **Query type**: StmtIn $\times$ pSpec.MessagesUpTo $i$.val.castSucc
- **Response type**: pSpec.Challenge $i$
- **Oracle behavior**: Returns the challenge (which is determined by the random oracle)

The query consists of the input statement and all prover messages sent up to (but not including) round $i$.

**Definition 87** (Fiat-Shamir Oracle Specification)**.** The Fiat-Shamir oracle specification for a protocol pSpec with input statement type StmtIn is:

$$\text{fiatShamirSpec pSpec StmtIn} : \text{OracleSpec pSpec.ChallengeIdx}$$

where for each challenge index $i$, the oracle domain is StmtIn $\times$ pSpec.MessagesUpTo $i$.val.castSucc and the range is pSpec.Challenge $i$.

This specification defines a family of oracles, one for each challenge round, that deterministically computes challenges based on the statement and messages up to that round.

### 2.3.2 Fiat-Shamir Transformation for Provers

The Fiat-Shamir transformation modifies the prover's execution to compute all messages non-interactively while simulating the verifier's challenges using oracle queries.

**Definition 88** (Fiat-Shamir Round Processing)**.** The modified round processing function for Fiat-Shamir maintains the prover messages (but not challenges) and the input statement throughout execution:

$$\text{processRoundFS } j \text{ prover currentResult}$$

For each round $j$:

- If $j$ is a challenge round: Query the Fiat-Shamir oracle with the statement and messages so far, then update the prover state with the received challenge
- If $j$ is a message round: Generate the message using the prover's sendMessage function and append it to the message history

The key difference from standard execution is that challenges are derived via oracle queries rather than received from an interactive verifier.

**Definition 89** (Fiat-Shamir Prover Execution)**.** The Fiat-Shamir prover execution up to round $i$ is defined as:

$$\mathsf{runToRoundFS}\ i\ \mathsf{stmt}\ \mathsf{prover}\ \mathsf{state}$$

This executes the prover inductively using processRoundFS, starting from the initial state and accumulating messages and the statement. Returns the messages up to round $i$, the input statement, and the prover's final state.

**Definition 90** (Fiat-Shamir Prover Transformation)**.** Given an interactive prover $P$ for protocol pSpec, the Fiat-Shamir transformation produces a non-interactive prover:

$P.\mathsf{fiatShamir} : \mathsf{NonInteractiveProver}\ (\forall i, \mathsf{pSpec.Message}\ i)\ (\mathsf{oSpec} +\!\!+_{\mathsf{o}} \mathsf{srChallengeOracle}\ \mathsf{pSpec}\ \mathsf{StmtIn})\ \mathsf{StmtIn}\ \mathsf{WitIn}\ \mathsf{StmtO}$

The transformed prover:

- Has state type that combines the statement with the original prover's state at round 0, and uses the final state type for subsequent rounds
- On input, stores both the statement and initializes the original prover's state
- Sends a single message containing all of the original prover's messages, computed via runToRoundFS
- Never receives challenges (since it's non-interactive)
- Outputs using the original prover's output function

### 2.3.3 Transcript Derivation and Verifier Transformation

The Fiat-Shamir verifier must reconstruct the full interactive transcript from the prover's messages in order to run the original verification logic.

**Definition 91** (Fiat-Shamir Transcript Derivation)**.** Given a collection of prover messages and an input statement, the function deriveTranscriptFS reconstructs the full protocol transcript up to round $k$:

$\mathsf{messages.deriveTranscriptFS}\ \mathsf{stmt}\ k : \mathsf{OracleComp}\ (\mathsf{oSpec} +\!\!+_{\mathsf{o}} \mathsf{srChallengeOracle}\ \mathsf{pSpec}\ \mathsf{StmtIn})\ (\mathsf{pSpec.Transcript}\ k)$

This is computed inductively:

- For challenge rounds: Query the Fiat-Shamir oracle with the statement and messages up to that point
- For message rounds: Use the corresponding message from the prover

The result is a complete transcript that includes both prover messages and verifier challenges.

**Definition 92** (Fiat-Shamir Verifier Transformation)**.** Given an interactive verifier $V$ for protocol pSpec, the Fiat-Shamir transformation produces a non-interactive verifier:

$V$.fiatShamir : NonInteractiveVerifier $(\forall i,$ pSpec.Message $i)$ (oSpec $++_o$ srChallengeOracle pSpec StmtIn) StmtIn StmtOut

The transformed verifier:

- Takes the input statement and a proof consisting of all prover messages
- Derives the full transcript using deriveTranscriptFS
- Runs the original verifier's verification logic on the reconstructed transcript

### 2.3.4 Fiat-Shamir Transformation for Reductions

**Definition 93** (Fiat-Shamir Reduction Transformation)**.** Given an interactive reduction $R$ for protocol pSpec, the Fiat-Shamir transformation produces a non-interactive reduction:

$R$.fiatShamir : NonInteractiveReduction $(\forall i,$ pSpec.Message $i)$ (oSpec $++_o$ srChallengeOracle pSpec StmtIn) StmtIn WitIn St

This transformation simply applies the Fiat-Shamir transformation to both the prover and verifier components of the reduction.

### 2.3.5 Security Properties

The Fiat-Shamir transformation preserves important security properties of the original interactive reduction, under appropriate assumptions about the random oracle.

**Theorem 94** (Fiat-Shamir Preserves Completeness)**.** *Let $R$ be an interactive reduction with completeness error $\epsilon$ with respect to input relation $R_{in}$ and output relation $R_{out}$. Then the Fiat-Shamir transformed reduction $R$.*fiatShamir *also satisfies completeness with error $\epsilon$ with respect to the same relations.*

*Formally:* $R$.completeness $R_{in}$ $R_{out}$ $\epsilon \to (R$.fiatShamir$)$.completeness $R_{in}$ $R_{out}$ $\epsilon$

**Remark 95** (Additional Security Properties)**.** While completeness is straightforward to establish, soundness properties require more careful analysis. In particular:

- State-restoration knowledge soundness of the original reduction implies knowledge soundness of the Fiat-Shamir transformed reduction

- Honest-verifier zero-knowledge of the original reduction implies zero-knowledge of the transformed reduction

These results require the random oracle model and careful handling of the oracle programming needed for simulation and extraction. The formal statements and proofs of these results are currently under development.

**Remark 96** (Implementation Considerations)**.** Our formalization models the "theoretical" version of Fiat-Shamir where the entire statement and transcript prefix are hashed to derive each challenge. In practice, more efficient variants use cryptographic sponges or other techniques to incrementally absorb transcript elements and squeeze out challenges. Our theoretical model provides the foundation for analyzing these practical variants.

# Chapter 3

# Proof Systems

## 3.1 Simple Oracle Reductions

We start by introducing a number of simple oracle reductions that serve as fundamental building blocks for more complex proof systems. These components can be composed together to construct larger protocols.

### 3.1.1 Trivial Reduction

The simplest possible oracle reduction is one that performs no computation at all. Both the prover and verifier simply pass their inputs through unchanged. While seemingly trivial, this reduction serves as an important identity element for composition and provides a base case for lifting and lens operations.

**Definition 97** (DoNothing Reduction)**.** The DoNothing reduction is a zero-round protocol with the following components:

- **Protocol specification**: $\mathsf{pSpec} := []$ (empty protocol, no messages exchanged)
- **Prover**: Simply stores the input statement and witness, and outputs them unchanged
- **Verifier**: Takes the input statement and outputs it directly
- **Input relation**: Any relation $R_{\mathsf{in}} : \mathsf{StmtIn} \to \mathsf{WitIn} \to \mathsf{Prop}$
- **Output relation**: The same relation $R_{\mathsf{out}} := R_{\mathsf{in}}$

**Theorem 98** (DoNothing Perfect Completeness)**.** *The DoNothing reduction satisfies perfect completeness for any input relation.*

The oracle version of DoNothing handles oracle statements by passing them through unchanged as well. The prover receives both non-oracle and oracle statements as input, and outputs them in the same form to the verifier.

### 3.1.2 Sending the Witness

A fundamental building block in many proof systems is the ability for the prover to transmit witness information to the verifier. The SendWitness reduction provides this functionality in both direct and oracle settings.

**Definition 99** (SendWitness Reduction)**.** The SendWitness reduction is a one-round protocol where the prover sends the complete witness to the verifier:

- **Protocol specification**: $\mathsf{pSpec} := [(\mathsf{P}\to\mathsf{V}, \mathsf{WitIn})]$ (single message from prover to verifier)
- **Prover**: Sends the witness $\mathtt{w}$ as its single message
- **Verifier**: Receives the witness and combines it with the input statement to form the output
- **Input relation**: $R_{\mathsf{in}} : \mathsf{StmtIn} \to \mathsf{WitIn} \to \mathsf{Prop}$
- **Output relation**: $R_{\mathsf{out}} : (\mathsf{StmtIn} \times \mathsf{WitIn}) \to \mathsf{Unit} \to \mathsf{Prop}$ defined by $((\mathsf{stmt}, \mathsf{wit}), ()) \mapsto R_{\mathsf{in}}(\mathsf{stmt}, \mathsf{wit})$

**Theorem 100** (SendWitness Perfect Completeness)**.** *The SendWitness reduction satisfies perfect completeness.*

In the oracle setting, we consider two variants:

**Definition 101** (SendWitness Oracle Reduction)**.** The oracle version handles witnesses that are indexed families of types with oracle interfaces:

- **Witness type**: $\mathsf{WitIn} : \iota_w \to \mathsf{Type}$ where each $\mathsf{WitIn}(i)$ has an oracle interface
- **Protocol specification**: $\mathsf{pSpec} := [(\mathsf{P}\to\mathsf{V}, \forall i, \mathsf{WitIn}(i))]$
- **Output oracle statements**: Combination of input oracle statements and the transmitted witness

**Definition 102** (SendSingleWitness Oracle Reduction)**.** A specialized variant for a single witness with oracle interface:

- **Witness type**: $\mathsf{WitIn} : \mathsf{Type}$ with oracle interface
- **Protocol specification**: $\mathsf{pSpec} := [(\mathsf{P}\to\mathsf{V}, \mathsf{WitIn})]$
- **Conversion**: Implicitly converts to indexed family $\mathsf{WitIn} : \mathsf{Fin}(1) \to \mathsf{Type}$

**Theorem 103** (SendSingleWitness Perfect Completeness)**.** *The SendSingleWitness oracle reduction satisfies perfect completeness.*

### 3.1.3 Oracle Equality Testing

One of the most fundamental oracle reductions is testing whether two oracles of the same type are equal. This is achieved through random sampling from the query space.

**Definition 104** (RandomQuery Oracle Reduction)**.** The RandomQuery reduction tests equality between two oracles by random querying:

- **Input**: Two oracles $\mathsf{a}, \mathsf{b}$ of the same type with oracle interface
- **Protocol specification**: $\mathsf{pSpec} := [(\mathsf{V}{\to}\mathsf{P}, \mathsf{Query})]$ (single challenge from verifier)
- **Input relation**: $R_{\mathsf{in}}((), (\mathsf{a}, \mathsf{b}), ()) := (\mathsf{a} = \mathsf{b})$
- **Verifier**: Samples random query $q$ and sends it to prover
- **Prover**: Receives query $q$, performs no computation
- **Output relation**: $R_{\mathsf{out}}((q, (\mathsf{a}, \mathsf{b})), ()) := (\mathsf{oracle}(\mathsf{a}, q) = \mathsf{oracle}(\mathsf{b}, q))$

**Theorem 105** (RandomQuery Perfect Completeness)**.** *The RandomQuery oracle reduction satisfies perfect completeness: if two oracles are equal, they will agree on any random query.*

The key security property of RandomQuery depends on the notion of oracle distance:

**Definition 106** (Oracle Distance)**.** For oracles $\mathsf{a}, \mathsf{b}$ of the same type, we define their distance as:

$$\mathsf{distance}(\mathsf{a}, \mathsf{b}) := |\{q : \mathsf{Query} \mid \mathsf{oracle}(\mathsf{a}, q) \neq \mathsf{oracle}(\mathsf{b}, q)\}|$$

We say an oracle type has distance bound $d$ if for any two distinct oracles $\mathsf{a} \neq \mathsf{b}$, we have $\mathsf{distance}(\mathsf{a}, \mathsf{b}) \geq |\mathsf{Query}| - d$.

**Theorem 107** (RandomQuery Knowledge Soundness)**.** *If the oracle type has distance bound $d$, then the RandomQuery oracle reduction satisfies round-by-round knowledge soundness with error probability $\frac{d}{|\mathsf{Query}|}$.*

**Definition 108** (RandomQueryWithResponse Variant)**.** A variant of RandomQuery where the second oracle is replaced with an explicit response:

- **Input**: Single oracle $\mathsf{a}$ and target response $r$
- **Output relation**: $R_{\mathsf{out}}(((q, r), \mathsf{a}), ()) := (\mathsf{oracle}(\mathsf{a}, q) = r)$

This variant is useful when one wants to verify a specific query-response pair rather than oracle equality.

We mention two special cases of RandomQuery that are useful for specific applications.

### Polynomial Equality Testing

A common application of oracle reductions is testing equality between polynomial oracles. This is a specific instance of the RandomQuery reduction applied to polynomial evaluation oracles.

**Definition 109** (Polynomial Equality Testing)**.** Consider two univariate polynomials $P, Q \in \mathbb{F}[X]$ of degree at most $d$, available as polynomial evaluation oracles. The polynomial equality testing reduction is defined as:

- **Input relation**: $P = Q$ as polynomials
- **Protocol specification**: Single challenge of type $\mathbb{F}$ from verifier to prover
- **Honest prover**: Receives the random field element $r$ but performs no computation
- **Honest verifier**: Checks that $P(r) = Q(r)$ by querying both polynomial oracles
- **Output relation**: $P(r) = Q(r)$ for the sampled point $r$

**Theorem 110** (Polynomial Equality Testing Completeness)**.** *The polynomial equality testing reduction satisfies perfect completeness: if $P = Q$ as polynomials, then $P(r) = Q(r)$ for any field element $r$.*

**Theorem 111** (Polynomial Equality Testing Soundness)**.** *The polynomial equality testing reduction satisfies round-by-round knowledge soundness with error probability $\frac{d}{|\mathbb{F}|}$, where $d$ is the maximum degree bound. This follows from the Schwartz-Zippel lemma: distinct polynomials of degree at most $d$ can agree on at most $d$ points.*

The state function for this reduction corresponds precisely to the input and output relations, transitioning from checking polynomial equality to checking evaluation equality at the sampled point.

### Batching Polynomial Evaluation Claims

Another important building block is the ability to batch multiple polynomial evaluation claims into a single check using random linear combinations.

TODO: express this as a lifted version of RandomQuery over a virtual polynomial whose variables are the random linear combination coefficients.

**Definition 112** (Batching Polynomial Evaluation Claims)**.** Consider an $n$-tuple of values $v = (v_1, \ldots, v_n) \in \mathbb{F}^n$ and a polynomial map $E : \mathbb{F}^k \to \mathbb{F}^n$. The batching reduction is defined as:

- **Protocol specification**: Two messages:
  1. Verifier sends random $r \in \mathbb{F}^k$ to prover
  2. Prover sends $\langle E(r), v\rangle := \sum_{i=1}^{n} E(r)_i \cdot v_i$ to verifier
- **Honest prover**: Computes the inner product $\langle E(r), v\rangle$ and sends it
- **Honest verifier**: Verifies that the received value equals the expected inner product
- **Extractor**: Trivial since there is no witness to extract

**Theorem 113** (Batching Completeness)**.** *The batching polynomial evaluation reduction satisfies perfect completeness.*

**Remark 114** (Batching Security)**.** The security of this reduction depends on the degree and non-degeneracy properties of the polynomial map $E$. The specific error bounds depend on the structure of $E$ and require careful analysis of the polynomial's properties.

### 3.1.4 Sending a Claim

The SendClaim reduction enables a prover to transmit a claim (oracle statement) to the verifier, who then verifies a relationship between the original and transmitted claims.

**Definition 115** (SendClaim Oracle Reduction)**.** The SendClaim reduction is a one-round protocol for claim transmission:

- **Protocol specification**: $\mathsf{pSpec} := [(\mathsf{P} \to \mathsf{V}, \mathsf{OStmtIn})]$ (single oracle message)
- **Input**: Statement and single oracle statement (via $\mathsf{Unique}$ index type)
- **Prover**: Sends the input oracle statement as protocol message
- **Verifier**: Executes oracle computation $\mathsf{relComp} : \mathsf{StmtIn} \to \mathsf{OracleComp}[\mathsf{OStmtIn}]_{\mathcal{O}}\mathsf{Unit}$
- **Output oracle statements**: Sum type $\mathsf{OStmtIn} \oplus \mathsf{OStmtIn}$ containing both original and transmitted claims
- **Output relation**: $R_{\mathsf{out}}((), \mathsf{oracles}) := \mathsf{oracles}(\mathsf{inl}) = \mathsf{oracles}(\mathsf{inr})$

**Theorem 116** (SendClaim Perfect Completeness)**.** *The SendClaim oracle reduction satisfies perfect completeness when the input relation matches the oracle computation requirement.*

**Remark 117** (SendClaim Development Status)**.** The SendClaim reduction is currently under active development in the Lean formalization. Several components including the verifier embedding and completeness proof require further implementation. The current version represents a specialized case that may be generalized in future iterations.

### 3.1.5 Claim Reduction

A fundamental building block for constructing complex proof systems is the ability to locally reduce one type of claim to another. The ReduceClaim reduction provides this functionality through mappings between statement and witness types.

**Definition 118** (ReduceClaim Reduction)**.** The ReduceClaim reduction is a zero-round protocol that transforms claims via explicit mappings:

- **Protocol specification**: $\mathsf{pSpec} := []$ (no messages exchanged)
- **Statement mapping**: $\mathsf{mapStmt} : \mathsf{StmtIn} \to \mathsf{StmtOut}$
- **Witness mapping**: $\mathsf{mapWit} : \mathsf{WitIn} \to \mathsf{WitOut}$

- **Prover**: Applies both mappings to input statement and witness
- **Verifier**: Applies statement mapping to input statement
- **Input relation**: $R_{\mathsf{in}} : \mathsf{StmtIn} \to \mathsf{WitIn} \to \mathsf{Prop}$
- **Output relation**: $R_{\mathsf{out}} : \mathsf{StmtOut} \to \mathsf{WitOut} \to \mathsf{Prop}$
- **Relation condition**: $R_{\mathsf{in}}(\mathsf{stmt}, \mathsf{wit}) \iff R_{\mathsf{out}}(\mathsf{mapStmt}(\mathsf{stmt}), \mathsf{mapWit}(\mathsf{wit}))$

**Theorem 119** (ReduceClaim Perfect Completeness)**.** *The ReduceClaim reduction satisfies perfect completeness when the relation condition holds.*

**Definition 120** (ReduceClaim Oracle Reduction)**.** The oracle version additionally handles oracle statements through an embedding:

- **Oracle statement mapping**: Embedding $\mathsf{embedIdx} : \iota_{\mathsf{out}} \hookrightarrow \iota_{\mathsf{in}}$
- **Type compatibility**: $\mathsf{OStmtIn}(\mathsf{embedIdx}(i)) = \mathsf{OStmtOut}(i)$ for all $i$
- **Oracle embedding**: Maps output oracle indices to corresponding input oracle indices

**Remark 121** (ReduceClaim Oracle Completeness)**.** The oracle version's completeness proof is currently under development in the Lean formalization.

### 3.1.6 Claim Verification

Another essential building block is the ability to verify that a given predicate holds for a statement without requiring additional witness information.

**Definition 122** (CheckClaim Reduction)**.** The CheckClaim reduction is a zero-round protocol that verifies predicates:

- **Protocol specification**: $\mathsf{pSpec} := []$ (no messages exchanged)
- **Predicate**: $\mathsf{pred} : \mathsf{StmtIn} \to \mathsf{Prop}$ (decidable)
- **Prover**: Simply stores and outputs the input statement with unit witness
- **Verifier**: Checks $\mathsf{pred}(\mathsf{stmt})$ and outputs statement if successful
- **Input relation**: $R_{\mathsf{in}}(\mathsf{stmt}, ()) := \mathsf{pred}(\mathsf{stmt})$
- **Output relation**: $R_{\mathsf{out}}(\mathsf{stmt}, ()) := \mathsf{True}$ (trivial after verification)

**Theorem 123** (CheckClaim Perfect Completeness)**.** *The CheckClaim reduction satisfies perfect completeness.*

**Definition 124** (CheckClaim Oracle Reduction)**.** The oracle version handles predicates that require oracle access:

- **Oracle predicate**: $\mathsf{pred} : \mathsf{StmtIn} \to \mathsf{OracleComp}[\mathsf{OStmtIn}]_{\mathcal{O}}\mathsf{Prop}$
- **Never-fails condition**: $\mathsf{pred}(\mathsf{stmt})$ never fails for any statement
- **Oracle computation**: Verifier executes oracle computation to check predicate
- **Input relation**: Defined via oracle simulation of the predicate

**Theorem 125** (CheckClaim Oracle Perfect Completeness)**.** *The CheckClaim oracle reduction satisfies perfect completeness.*

**Remark 126** (CheckClaim Security Analysis)**.** The round-by-round knowledge soundness proofs for both reduction and oracle versions are currently under development in the Lean formalization.

## 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
- $\mathcal{D} : \{0, 1, ..., m-1\} \hookrightarrow R$: 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 127** (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 128** (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 129** (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 130** (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 131** (Single Round Completeness)**.** *Each individual round of the sum-check protocol is perfectly complete.*

**Theorem 132** (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 133** (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 134.** 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 135.** *The virtual sum-check round protocol is sound.*

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

**Theorem 136.** *The virtual sum-check round protocol is knowledge sound.*

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

**Definition 137** (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 138.** *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 Binius

This section documents our formalization of the Binius commitment scheme protocols. These protocols are built upon the unique hierarchical structure of binary tower fields, which enables highly efficient arithmetic. We first describe the primitives used in the protocols, and then describe the protocols themselves.

### 3.3.1 Binary Tower Fields

We define the binary tower fields [9] as defined originally as iterated quadratic extensions by Wie88[13]. These fields, denoted $(\mathcal{T}_\iota)_{\iota\in\mathbb{N}}$, provide a chain of nested field extensions for efficient arithmetic, particularly for operations involving subfields, by leveraging a highly compatible basis structure across the tower.

**Definition 139** (Binary Tower Field)**.** A binary tower field $\mathcal{T}_\iota$ for $\iota \in \mathbb{N}$ is defined inductively as the $\iota$-th field in the sequence of quadratic extensions over the ground field $\mathbb{F}_2$.

- $\mathcal{T}_0 := \mathbb{F}_2$
- $\forall \iota > 0$, $\mathcal{T}_\iota := \mathcal{T}_{\iota-1}[X_{\iota-1}]/(X_{\iota-1}^2 + X_{\iota-2} \cdot X_{\iota-1} + 1)$, where we conventionally set $X_{-1} := 1$.

**Theorem 140** (Irreducible defining polynomial)**.** *The defining polynomial $X_{\iota-1}^2 + X_{\iota-2} \cdot X_{\iota-1} + 1$ of $\mathcal{T}_\iota$ is irreducible over $\mathcal{T}_{\iota-1}$ for all $\iota > 0$.*

**Theorem 141** (Binary Tower Fields are fields)**.** *We prove that the binary tower fields are finite fields:*

- *The ground field $\mathcal{T}_0 := \mathbb{F}_2$ is a field. For all $\iota > 0$, $\mathcal{T}_\iota$ is the quotient ring of $\mathcal{T}_{\iota-1}[X_{\iota-1}]$ by the irreducible polynomial $X_{\iota-1}^2 + X_{\iota-2} \cdot X_{\iota-1} + 1$, therefore $\mathcal{T}_\iota$ is a field extension of $\mathcal{T}_{\iota-1}$.*
- *For all $\iota \in \mathbb{N}$, the cardinality of $\mathcal{T}_\iota$ is $2^{2^\iota}$.*
- *For all $\iota \in \mathbb{N}$, the characteristic of $\mathcal{T}_\iota$ is 2.*

The structure of the tower provides a natural way to represent elements using a consistent set of variables. This leads to a family of multilinear bases that are compatible across different levels of the tower.

**Definition 142** (Multilinear Bases for Tower Fields)**.** For any tower field $\mathcal{T}_\iota$, we define its canonical bases as follows:

- **$\mathbb{F}_2$-Basis of $\mathcal{T}_\iota$**: The set of multilinear monomials in the variables $\{X_0, \ldots, X_{\iota-1}\}$ forms a basis for $\mathcal{T}_\iota$ as a $2^\iota$-dimensional vector space over $\mathbb{F}_2$. An element is typically stored as a $2^\iota$-bit string corresponding to this basis.

- $\mathcal{T}_\iota$**-Basis of** $\mathcal{T}_{\iota+\kappa}$: For any $\kappa \geq 0$, the set of multilinear monomials in the variables $\{X_\iota, \ldots, X_{\iota+\kappa-1}\}$ forms a basis for $\mathcal{T}_{\iota+\kappa}$ as a $2^\kappa$-dimensional vector space over the subfield $\mathcal{T}_\iota$.

**Definition 143** (Computable Binary Tower Fields)**.** Building upon the abstract definition of binary tower fields, we define a concrete, computable representation of binary tower fields. This construction, which underpins our formalization, represents each element of the field $\mathcal{T}_\iota$ as a bit vector of length $2^\iota$ corresponding to the coefficients of the multilinear $\mathbb{F}_2$-basis.

The arithmetic operations on these bit-vector representations are defined as follows:

- **Addition**: The sum of two elements is defined as the bitwise XOR of their corresponding bit-vector representations.
- **Multiplication (in** $\mathcal{T}_\iota$**)**: The product of two elements within the same field $\mathcal{T}_\iota$ is defined via a recursive Karatsuba-based algorithm [10]. The complexity of this operation is $\Theta(2^{\log_2(3)\cdot\iota})$.
- **Cross-Level Multiplication**: The product of an element $\alpha \in \mathcal{T}_{\iota+\kappa}$ by a scalar $b \in \mathcal{T}_\iota$ is defined by representing $\alpha$ via its $2^\kappa$ coefficients $(a_u)_{u\in\{0,1\}^\kappa}$ in the $\mathcal{T}_\iota$-basis and performing the multiplication component-wise on those coefficients in the subfield $\mathcal{T}_\iota$. The total complexity is $2^\kappa \cdot \Theta(2^{\log_2(3)\cdot\iota})$.

## 3.4 The Spartan Protocol

### 3.4.1 Preliminaries

The Spartan protocol is designed to prove the satisfiability of Rank-1 Constraint System (R1CS). An R1CS instance is defined by a set of matrices $(A, B, C)$ over a field $\mathbb{F}$ (more generally the definition makes sense over any semiring $\mathsf{R}$), and it is satisfied by a public input $x$ and a private witness $w$ if the combined vector $z = (x, w)$ satisfies the relation:

$$(A \cdot z) \circ (B \cdot z) = (C \cdot z)$$

where $\circ$ denotes the Hadamard (entry-wise) product. Our formalization follows the definition in `ArkLib/ProofSystem/ConstraintSystem/R1CS.lean`.

### 3.4.2 Description in Paper

Figure 3.1 is the description of the Spartan protocol from the original paper [12]. Note that in this section, we only formalize the *Polynomial IOP (PIOP)* aspect of Spartan. In the PIOP model, the prover does not commit to polynomials using a Polynomial Commitment Scheme (PCS). Instead, the verifier is given oracle access to the polynomials. Therefore, steps involving 'PCS.Setup', 'PCS.Commit', and 'PCS.Eval' are replaced by simple oracle interactions.

After stripping away the polynomial commitment scheme, the protocol has the following structure:

**Setup:** This step is part of the polynomial commitment scheme and is not part of the PIOP we formalize.

**Interaction:** The interaction is between a prover $\mathcal{P}$ with witness $w$ and a verifier $\mathcal{V}$ with public inputs $(\mathbb{F}, A, B, C, \text{io}, m, n)$.

1. $\mathcal{P}$: Sends oracle access to the multilinear extension of the witness, $\tilde{w}$, to $\mathcal{V}$. In the original paper, this is a commitment $(C, S) \leftarrow \text{PC.Commit}(\text{pp}, \tilde{w})$.
2. $\mathcal{V}$: Samples a random challenge $\tau \in \mathbb{F}^{\log m}$ and sends it to $\mathcal{P}$.
3. Let $T_1 = 0$, $\mu_1 = \log m$, $\ell_1 = 3$. These are parameters for the first sum-check protocol.
4. $\mathcal{V}$: Samples a random challenge $r_x \in \mathbb{F}^{\mu_1}$.
5. **Sum-check#1.** A sum-check protocol is executed. The verifier receives the claimed evaluation $e_x$. The prover of the sum-check has oracle access to a polynomial $G_{io,\tau}$, and the verifier has oracle access to $r_x$. The parameters for this sub-protocol are $(\mu_1, \ell_1, T_1)$.
6. $\mathcal{P}$: Computes evaluations $v_A = \tilde{A}(r_x)$, $v_B = \tilde{B}(r_x)$, $v_C = \tilde{C}(r_x)$ and sends them to $\mathcal{V}$. $\tilde{A}, \tilde{B}, \tilde{C}$ are multilinear extensions of the matrices $A, B, C$.
7. $\mathcal{V}$: Aborts if $e_x \neq (v_A \cdot v_B - v_C) \cdot \tilde{\text{eq}}(r_x, \tau)$. This is the verifier's check for the first sum-check.
8. $\mathcal{V}$: Samples random challenges $r_A, r_B, r_C \in \mathbb{F}$ and sends them to $\mathcal{P}$.
9. Let $T_2 = r_A \cdot v_A + r_B \cdot v_B + r_C \cdot v_C$, $\mu_2 = \log n$, $\ell_2 = 2$. These are parameters for the second sum-check protocol. Note: The image states $\mu_2 = \log m$, which is likely a typo and should be $\log n$.
10. $\mathcal{V}$: Samples a random challenge $r_y \in \mathbb{F}^{\mu_2}$.
11. **Sum-check#2.** Another sum-check protocol is executed. The verifier receives the claimed evaluation $e_y$.
12. $\mathcal{P}$: Computes $v \leftarrow \tilde{w}(r_y[1..])$ and sends $v$ to $\mathcal{V}$.
13. This step involves a polynomial commitment evaluation proof. In our PIOP formalization, this check is not needed as the verifier has direct oracle access to $\tilde{w}$.
14. $\mathcal{V}$: This step is part of the evaluation proof check, so it is omitted.
15. $\mathcal{V}$: Computes $v_Z \leftarrow (1 - r_y[0]) \cdot \tilde{w}(r_y[1..]) + r_y[0] \cdot \widetilde{(\text{io}, 1)}(r_y[1..])$. This reconstructs the evaluation of the combined input-witness vector polynomial $\tilde{z}$.
16. $\mathcal{V}$: Queries oracles for $\tilde{A}, \tilde{B}, \tilde{C}$ at $(r_x, r_y)$ to get $v_1, v_2, v_3$.
17. $\mathcal{V}$: Aborts if $e_y \neq (r_A \cdot v_1 + r_B \cdot v_2 + r_C \cdot v_3) \cdot v_Z$. This is the final check.
18. $\mathcal{V}$: Outputs 1.

### 3.4.3 Formalization using IOR Composition

## 3.5 Stir

### 3.5.1 Tools for Reed Solomon codes

**Random linear combination as a proximity generator**

**Theorem 144.** *Let $\mathcal{C} := \text{RS}[\mathbb{F}, \mathcal{L}, d]$ be a Reed Solomon code with rate $\rho := \frac{d}{|\mathcal{L}|}$ and let $B^*(\rho) := \sqrt{\rho}$. For every $\delta \in (0, 1 - B^*(\rho))$ and functions $f_0, \ldots, f_{m-1} : \mathcal{L} \to \mathbb{F}$, if*

$$\Pr_{r \leftarrow \mathbb{F}}\Big[\Delta\Big(\sum_{j=0}^{m-1} r^j \cdot f_j, \text{RS}[\mathbb{F}, \mathcal{L}, d]\Big) \leq \delta\Big] > \mathsf{err}^*(d, \rho, \delta, m),$$

*then there exists a subset* $S \subseteq \mathcal{L}$ *with* $|S| \geq (1-\delta) \cdot |L|$*, and for every* $i \in [m]$*, there exists* $u \in \mathrm{RS}[\mathbb{F}, \mathcal{L}, d]$ *such that* $f_i(S) = u(S)$*.*

*Above,* $\mathsf{err}^*(d, \rho, \delta, m)$ *is defined as follows:*

- *if* $\delta \in \left(0, \frac{1-\rho}{2}\right]$ *then*

$$\mathsf{err}^*(d, \rho, \delta, m) = \frac{(m-1) \cdot d}{\rho \cdot |\mathbb{F}|}$$

- *if* $\delta \in \left(\frac{1-\rho}{2}, 1 - \sqrt{\rho}\right)$ *then*

$$\mathsf{err}^*(d, \rho, \delta, m) = \frac{(m-1) \cdot d^2}{|\mathbb{F}| \cdot \left(2 \cdot \min\{1 - \sqrt{\rho} - \delta, \frac{\sqrt{\rho}}{20}\}\right)^7}$$

**Univariate Function Quotienting**

In the following, we start by defining the *quotient* of a univariate function.

**Definition 145.** Let $f : \mathcal{L} \to \mathbb{F}$ be a function, $S \subseteq \mathbb{F}$ be a set, and $\mathsf{Ans}, \mathsf{Fill} : S \to \mathbb{F}$ be functions. Let $\widehat{\mathsf{Ans}} \in \mathbb{F}^{<|S|}[X]$ be the (unique) polynomial with $\widehat{\mathsf{Ans}}(x) = \mathsf{Ans}(x)$ for every $x \in S$, and let $\widehat{V}_S \in \mathbb{F}^{<|S|+1}[X]$ be the unique non-zero polynomial with $\widehat{V}_S(x) = 0$ for every $x \in S$. The *quotient function* $\mathsf{Quotient}(f, S, \mathsf{Ans}, \mathsf{Fill}) : \mathcal{L} \to \mathbb{F}$ is defined as follows:

$$\forall x \in \mathcal{L}, \quad \mathsf{Quotient}(f, S, \mathsf{Ans}, \mathsf{Fill})(x) := \begin{cases} \mathsf{Fill}(x) & \text{if } x \in S \\ \dfrac{f(x) - \widehat{\mathsf{Ans}}(x)}{\widehat{V}_S(x)} & \text{otherwise} \end{cases}$$

Next we define the polynomial quotient operator, which quotients a polynomial relative to its output on evaluation points. The polynomial quotient is a polynomial of lower degree.

**Definition 146.** Let $\widehat{f} \in \mathbb{F}^{<d}[X]$ be a polynomial and $S \subseteq \mathbb{F}$ be a set, let $\widehat{V}_S \in \mathbb{F}^{<|S|+1}[X]$ be the unique non-zero polynomial with $\widehat{V}_S(x) = 0$ for every $x \in S$. The *polynomial quotient* $\mathsf{PolyQuotient}(\widehat{f}, S) \in \mathbb{F}^{<d-|S|}[X]$ is defined as follows:

$$\mathsf{PolyQuotient}(\widehat{f}, S)(X) := \frac{\widehat{f}(X) - \widehat{\mathsf{Ans}}(X)}{\widehat{V}_S(X)}$$

where $\widehat{Ans} \in \mathbb{F}^{<|S|}[X]$ is the unique non-zero polynomial with $\widehat{Ans}(x) = \widehat{f}(x)$ for every $x \in S$.

The following lemma, implicit in prior works, shows that if the function is "quotiented by the wrong value", then its quotient is far from low-degree.

**Lemma 147.** *Let* $f : \mathcal{L} \to \mathbb{F}$ *be a function,* $d \in \mathbb{N}$ *be the degree parameter,* $\delta \in (0, 1)$ *be a distance parameter,* $S \subseteq \mathbb{F}$ *be a set with* $|S| < d$*, and* $\mathsf{Ans}, \mathsf{Fill} : S \to \mathbb{F}$ *are functions. Suppose that for every* $u \in \mathsf{List}(f, d, \delta)$ *there exists* $x \in S$ *with* $\widehat{u}(x) \neq \mathsf{Ans}(x)$*. Then*

$$\Delta(\mathsf{Quotient}(f, S, \mathsf{Ans}, \mathsf{Fill}), \mathrm{RS}[\mathbb{F}, \mathcal{L}, d - |S|]) + \frac{|T|}{|\mathcal{L}|} > \delta,$$

*where* $T := \{x \in \mathcal{L} \cap S : \widehat{\mathsf{Ans}}(x) \neq f(x)\}$*.*

**Out of domain sampling**

**Lemma 148.** *Let $f : \mathcal{L} \to \mathbb{F}$ be a function, $d \in \mathbb{N}$ be a degree parameter, $s \in \mathbb{N}$ be a repetition parameter, and $\delta \in [0,1]$ be a distance parameter. If $\mathrm{RS}[\mathbb{F}, \mathcal{L}, d]$ be $(d,l)$-list decodable then*

$$\begin{aligned} \Pr_{r_0,\ldots,r_{s-1} \leftarrow \mathbb{F} \setminus \mathcal{L}} \left[ \begin{array}{c} \exists \text{ distinct } u, u' \in \mathsf{List}(f,d,\delta) : \\ \forall i \in [s],\ \hat{u}(r_i) = \hat{u}'(r_i) \end{array} \right] &\leq \binom{l}{2} \cdot \left(\tfrac{d-1}{|\mathbb{F}|-|\mathcal{L}|}\right)^s \\ &\leq \left(\tfrac{l^2}{2}\right) \cdot \left(\tfrac{d}{|\mathbb{F}|-|\mathcal{L}|}\right)^s \end{aligned}$$

**Folding univariate functions**

STIR relies on $k$-wise folding of functions and polynomials - this is similar to prior works, although presented in a slightly different form. As shown below, folding a function preserves proximity from the Reed-Solomon code with high probability. The folding operator is based on the following fact, decomposing univariate polynomials into bivariate ones.

**Lemma 149.** *Given a polynomial $\hat{q} \in \mathbb{F}[X]$:*

- *For every univariate polynomial $\hat{f} \in \mathbb{F}[X]$, there exists a unique bivariate polynomial $\hat{Q} \in \mathbb{F}[X,Y]$ with:*

$$\mathsf{deg}_X(\hat{Q}) := \left\lfloor \frac{\mathsf{deg}(\hat{f})}{\mathsf{deg}(\hat{q})} \right\rfloor, \quad \mathsf{deg}_Y(\hat{Q}) < \mathsf{deg}(\hat{q})$$

  *such that $\hat{f}(Z) = \hat{Q}(\hat{q}(Z), Z)$. Moreover, $\hat{Q}$ can be computed efficiently given $\hat{f}$ and $\hat{q}$. Observe that if $\mathsf{deg}(\hat{f}) < t \cdot \mathsf{deg}(\hat{q})$ then $\mathsf{deg}(\hat{Q}) < t$.*

- *For every $\hat{Q}[X,Y]$ with $\mathsf{deg}_X(\hat{Q}) < t$ and $\mathsf{deg}_Y(\hat{Q}) < \mathsf{deg}(\hat{q})$, the polynomial $\hat{f}(Z) = \hat{Q}(\hat{q}(Z), Z)$ has degree $\mathsf{deg}(\hat{f}) < t \cdot \mathsf{deg}(\hat{q})$.*

Below, we define folding of a polynomial followed by folding of a function.

**Definition 150.** Given a polynomial $\hat{f} \in \mathbb{F}^{<d}[X]$, a folding parameter $k \in \mathbb{N}$ and $r \in \mathbb{F}$, we define a polynomial $\mathsf{PolyFold}(\hat{f}, k, r) \in \mathbb{F}^{d/k}[X]$ as follows. Let $\hat{Q}[X,Y]$ be the bivariate polynomial derived from $\hat{f}$ using Fact 149 with $\hat{q}(X) := X^k$. Then $\mathsf{PolyFold}(\hat{f}, k, r)(X) := \hat{Q}(X, r)$.

**Definition 151.** Let $f : \mathcal{L} \to \mathbb{F}$ be a function, $k \in \mathbb{N}$ a folding parameter and $\alpha \in \mathbb{F}$. For every $x \in \mathcal{L}^k$, let $\hat{p}_x \in \mathbb{F}^{<k}[X]$ be the polynomial where $\hat{p}_x(y) = f(y)$ for every $y \in \mathcal{L}$ such that $y^k = x$. We define $\mathsf{Fold}(f, k, \alpha) : \mathcal{L} \to \mathbb{F}$ as follows.

$$\mathsf{Fold}(f, k, \alpha) := \hat{p}_x(\alpha).$$

In order to compute $\mathsf{Fold}(f, k, \alpha)(x)$ it suffices to interpolate the $k$ values $\{f(y) : y \in \mathcal{L} \text{ s.t. } y^k = x\}$ into the polynomial $\hat{p}_x$ and evaluate this polynomial at $\alpha$.

The following lemma shows that the distance of a function is preserved under folding. If a functions $f$ has distance $\delta$ to a Reed-Solomon code then, with high probability over the choice of folding randomness, its folding also has a distance of $\delta$ to the "$k$-wise folded" Reed-Solomon code.

**Lemma 152.** *For every function $f : \mathcal{L} \to \mathbb{F}$, degree parameter $d \in \mathbb{N}$, folding parameter $k \in \mathbb{N}$, distance parameter $\delta \in (0, \min\{\Delta(\mathsf{Fold}[f, k, r^{\mathsf{fold}}], \mathrm{RS}[\mathbb{F}, \mathcal{L}^k, d/k]), 1 - \mathsf{B}^*(\rho)\})$, letting $\rho := \frac{d}{|\mathcal{L}|}$,*

$$\Pr_{r^{\mathsf{fold}} \leftarrow \mathbb{F}}[\Delta(\mathsf{Fold}[f, k, r^{\mathsf{fold}}], \mathrm{RS}[\mathbb{F}, \mathcal{L}^k, d/k]) < \delta] > \mathsf{err}^*(d/k, \rho, \delta, k).$$

*Above, $\mathsf{B}^*$ and $\mathsf{err}^*$ are the proximity bound and error (respectively) described in Section 3.5.1.*

### Combine functions of varying degrees

We show a new method for combining functions of varying degrees with minimal proximity requirements using geometric sums. We begin by recalling a fact about geometric sums.

**Lemma 153.** *Let $\mathbb{F}$ be a field, $r \in \mathbb{F}$ be a field element, $a \in \mathbb{N}$ be a natural number. Then*

$$\sum_{i=0}^{a} r^i := \begin{cases} \left(\dfrac{1 - r^{a+1}}{1 - r}\right) & \text{if } r \neq 1 \\ a + 1 & \text{if } r = 1 \end{cases}$$

**Definition 154.** Given target degree $d^* \in \mathbb{N}$, shifting parameter $r \in \mathbb{F}$, functions $f_0, \ldots, f_{m-1} : \mathcal{L} \to \mathbb{F}$, and degrees $0 \leq d_0, \ldots, d_{m-1} \leq d^*$, we define $\mathsf{Combine}(d^*, r, (f_0, d_0), \ldots, (f_{m-1}, d_{m-1})) : \mathcal{L} \to \mathbb{F}$ as follows:

$$\begin{aligned} \mathsf{Combine}(d^*, r, (f_0, d_0), \ldots, (f_{m-1}, d_{m-1}))(x) &:= \sum_{i=0}^{m-1} r_i \cdot f_i(x) \cdot \left(\sum_{l=0}^{d^* - d_i} (r \cdot x)^l\right) \\ &= \begin{cases} \displaystyle\sum_{i=0}^{m-1} r_i \cdot f_i(x) \cdot \left(\frac{1 - (xr)^{d^* - d_i + 1}}{1 - xr}\right) & \text{if } x \cdot r \neq 1 \\ \displaystyle\sum_{i=0}^{m-1} r_i \cdot f_i(x) \cdot (d^* - d_i + 1) & \text{if } x \cdot r = 1 \end{cases} \end{aligned}$$

Above, $r_i := r^{i - 1 + \sum_{j<i}(d^* - d_j)}$.

**Definition 155.** Given target degree $d^* \in \mathbb{N}$, shifting parameter $r \in \mathbb{F}$, function $f : \mathcal{L} \to \mathbb{F}$, and degree $0 \leq d \leq d^*$, we define $\mathsf{DegCor}(d^*, r, f, d)$ as follows.

$$\begin{aligned} \mathsf{DegCor}(d^*, r, f, d)(x) &:= f(x) \cdot \left(\sum_{l=0}^{d^* - d} (r \cdot x)^l\right) \\ &= \begin{cases} f(x) \cdot \left(\dfrac{1 - (xr)^{d^* - d_i + 1}}{1 - xr}\right) & \text{if } x \cdot r \neq 1 \\ f(x) \cdot (d^* - d_i + 1) & \text{if } x \cdot r = 1 \end{cases} \end{aligned}$$

(Observe that $\mathsf{DegCor}(d^*, r, f, d) = \mathsf{Combine}(d^*, r, (f, d))$.)

Below it is shown that combining multiple polynomials of varying degrees can be done as long as the proximity error is bounded by $(\min\{1 - \mathsf{B}^*(\rho), 1 - \rho - 1/|\mathcal{L}|\})$.

**Lemma 156.** *Let $d^*$ be a target degree, $f_0, \ldots, f_{m-1} : \mathcal{L} \to \mathbb{F}$ be functions, $0 \leq d_0, \ldots, d_{m-1} \leq d^*$ be degrees, $\delta \in \min\{1 - \mathsf{B}^*(\rho), 1 - \rho - 1/|\mathcal{L}|\}$ be a distance parameter, where $\rho = d^*/|\mathcal{L}|$. If*

$$\Pr_{r \leftarrow \mathbb{F}}[\Delta(\mathsf{Combine}(d^*, r, (f_0, d_0), \ldots, (f_{m-1}, d_{m-1})), \mathrm{RS}[\mathbb{F}, \mathcal{L}, d^*])] > \mathsf{err}^*(d^*, \rho, \delta, m \cdot (d^* + 1) - \sum_{i=0}^{m-1} d_i),$$

*then there exists $S \subseteq \mathcal{L}$ with $|S| \geq (1 - \delta) \cdot |\mathcal{L}|$, and*

$$\forall i \in [m-1], \exists u \in \mathrm{RS}[\mathbb{F}, \mathcal{L}, d_i], f_i(S) = u(S).$$

*Note that this implies $\Delta(f_i, \mathrm{RS}[\mathbb{F}, \mathcal{L}, d_i]) < \delta$ for every $i$. Above, $\mathsf{B}^*$ and $\mathsf{err}^*$ are the proximity bound and error (respectively) described in the proximity gap theorem.*

### 3.5.2 Stir Main theorems

**Theorem 157** (STIR Main Theorem)**.** *Consider the following ingrediants:*

- *A security parameter $\lambda \in \mathbb{N}$.*
- *A Reed-Solomon code $\mathrm{RS}[\mathbb{F}, \mathcal{L}, d]$ with $\rho := \frac{d}{|\mathcal{L}|}$ where $d$ is a power of 2, and $\mathcal{L}$ is a smooth domain.*
- *A proximity parameter $\delta \in (0, 1 - 1.05 \cdot \sqrt{\rho})$.*
- *A folding parameter $k \in \mathbb{N}$ that is power of 2 with $k \geq 4$.*

*If $|\mathbb{F}| = \Omega(\frac{\lambda \cdot 2^\lambda \cdot d^2 \cdot |\mathcal{L}|^2}{\log(1/\rho)})$, there is a public-coin IOPP for $\mathrm{RS}[\mathbb{F}, \mathcal{L}, d]$ with the following parameters:*

- *Round-by-round soundness error $2^{-\lambda}$.*
- *Round complexity: $M := O(\log_k d)$.*
- *Proof length: $|\mathcal{L}| + O_k(\log d)$.*
- *Query complexity to the input: $\frac{\lambda}{-\log(1-\delta)}$.*
- *Query complexity to the proof strings: $O_k(\log d + \lambda \cdot \log\left(\frac{\log d}{\log 1/\rho}\right))$.*

**Lemma 158.** *Consider $(\mathbb{F}, M, d, k_0, \ldots, k_M, \mathcal{L}_0, \ldots, \mathcal{L}_M, t_0, \ldots, t_M)$ and for every $i \in \{0, \ldots, M\}$, let $d_i := \frac{d}{\prod_{j<i} k^j}$ and $\rho_i := d_i/|\mathcal{L}_i|$. For every $f \notin \mathrm{RS}[\mathbb{F}, \mathcal{L}_0, d_0]$ and every $\delta_0, \ldots, \delta_M$ where*

- *$\delta_0 \in (0, \Delta(f, \mathrm{RS}[\mathbb{F}, \mathcal{L}_0, d_0])] \cap (0, 1 - \mathsf{B}^*(\rho_0))$*
- *for every $0 < i \leq M$: $\delta_i \in (0, \min\{1 - \rho_i - \frac{1}{|\mathcal{L}_i|}, 1 - \mathsf{B}^*(\rho_i)\})$, and*
- *for every $0 < i \leq M$: $\mathrm{RS}[\mathbb{F}, \mathcal{L}_i, d_i]$ is $(\delta_i, l_i)$-list decodable,*

*There exists an IOPP with above parameters, that has round-by-round soundness error $(\epsilon^{\mathsf{fold}}, \epsilon_1^{\mathsf{out}}, \epsilon_1^{\mathsf{shift}}, \ldots, \epsilon_M^{\mathsf{out}}, \epsilon_M^{\mathsf{shift}}, \epsilon^{\mathsf{fin}})$ where:*

- *$\epsilon^{\mathsf{fold}} \leq \mathsf{err}^*(d_0/k_0, \rho_0, \delta_0, k_0)$.*

- $\epsilon_i^{\mathsf{out}} \leq \frac{l_i^2}{2} \cdot \left(\frac{d_i}{|\mathbb{F}|-|\mathcal{L}_i|}\right)^s$
- $\epsilon_i^{\mathsf{shift}} \leq (1-\delta_{i-1})^{t_{i-1}} + \mathsf{err}^*(d_i, \rho_i, \delta_i, t_{i-1}+s) + \mathsf{err}^*(d_i/k_i, \rho_i, \delta_i, k_i)$.
- $\epsilon^{\mathsf{fin}} \leq (1-\delta_M)^{t_M}$.

*Above,* $\mathsf{B}^*$ *and* $\mathsf{err}^*$ *are the proximity bound and error (respectively) described in Section 3.5.1.*

## 3.6 Whir

### 3.6.1 Tools for Reed Solomon codes

#### Mutual Correlated Agreement as a Proximity Generator

**Definition 159.** Let $\mathcal{C} \subseteq \mathbb{F}^{\mathcal{L}}$ be a linear code. We say that $\mathsf{Gen}$ is a proximity generator for $\mathcal{C}$ with proximity bounds $\mathsf{B}$ and $\mathsf{err}$ if the following implication holds for $f_0, \ldots, f_{\mathsf{par}\ell-1} : \mathcal{L} \to \mathbb{F}$ and $\delta \in (0, 1-\mathsf{B}(\rho, \mathsf{par}\ell))$. If

$$\Pr_{r_0,\ldots,r_{\mathsf{par}\ell-1}\leftarrow\mathsf{Gen}}[\Delta(\sum_{i\in[0,(\mathsf{par}\ell-1)]} r_i \cdot f_i, \mathcal{C}) \leq \delta] > err(\mathcal{C}, \mathsf{par}\ell, \delta),$$

then there exists $S \subseteq \mathcal{L}$, $|S| > (1-\delta)\cdot|\mathcal{L}|$, and $\forall i \in [0, (\mathsf{par}\ell-1)]$, $\exists u \in \mathcal{C}, \forall x \in S$, $f_i(x) = u(x)$.

**Theorem 160.** *Let* $\mathcal{C} = \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ *be a Reed Solomon code with rate* $\rho = 2^m/|\mathcal{L}|$. $\mathsf{Gen}(\alpha, \mathsf{par}\ell) = \{1, \alpha, \ldots, \alpha^{\mathsf{par}\ell-1}\}$ *is a proximity generator for* $\mathcal{C}$ *with proximity bounds* $\mathsf{B}(\rho, \mathsf{par}\ell) = \sqrt{\rho}$ *and* $\mathsf{err}(C, \mathsf{par}\ell, \delta)$ *defined below.*

- *if* $\delta \in \left(0, \frac{1-\rho}{2}\right]$ *then*
$$\mathsf{err}(\mathcal{C}, \mathsf{par}\ell, \delta) = \frac{(m-1)\cdot d}{\rho \cdot |\mathbb{F}|}$$
- *if* $\delta \in \left(\frac{1-\rho}{2}, 1-\sqrt{\rho}\right)$ *then*
$$\mathsf{err}(\mathcal{C}, \mathsf{par}\ell, \delta) = \frac{(m-1)\cdot d^2}{|\mathbb{F}| \cdot \left(2 \cdot \min\{1-\sqrt{\rho}-\delta, \frac{\sqrt{\rho}}{20}\}\right)^7}$$

**Definition 161.** Let $\mathcal{C}$ be a linear code. We say that $\mathsf{Gen}$ be a proximity generator with mutual correlated agreement with proximity bounds $\mathsf{B}^\star$ and $\mathsf{err}^\star$, if for $f_0, \ldots, f_{\mathsf{par}\ell-1} : \mathcal{L} \to \mathbb{F}$ and $\delta \in (0, 1-\mathsf{B}^\star(\mathcal{C}, \mathsf{par}\ell))$ the following holds.

$$\Pr_{(r_0,\ldots,r_{\mathsf{par}\ell-1})\leftarrow\mathsf{Gen}(\mathsf{par}\ell)}\left[\exists S \subseteq \mathcal{L} \;\; s.t. \quad \begin{array}{l} |S| \geq (1-\delta)\cdot|\mathcal{L}| \\ \wedge\, \exists u \in \mathcal{C}, u(S) = \sum_{j\in[0,(\mathsf{par}\ell-1)]} r_j \cdot f_j(S) \\ \wedge\, \exists i \in [0, (\mathsf{par}\ell-1)], \forall u' \in \mathcal{C}, u'(S) \neq f_i(S) \end{array}\right] \leq \mathsf{err}^\star(\mathcal{C}, \mathsf{par}\ell, \delta).$$

**Lemma 162.** *Let* $\mathcal{C}$ *be a linear code with minimum distance* $\delta_{\mathcal{C}}$ *and let* $\mathsf{Gen}$ *be a proximity generator for* $\mathcal{C}$ *with proximity bound* $\mathsf{B}$ *and error* $\mathsf{err}$. *Then* $\mathsf{Gen}$ *has mutual correlated agreement with proximity bound* $\mathsf{B}^\star(\mathcal{C}, \mathsf{par}\ell) = \min\{1-\delta_{\mathcal{C}}/2, \mathsf{B}(\mathcal{C}, \mathsf{par}\ell)\}$ *and error* $\mathsf{err}^\star(\mathcal{C}, \mathsf{par}\ell, \delta) := \mathsf{err}(\mathcal{C}, \mathsf{par}\ell, \delta)$.

**Lemma 163.** *Let $\mathcal{C} := \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ be a Reed Solomon code with rate $\rho$. The function $\mathsf{Gen}(\mathsf{par}\ell; \alpha) = (1, \alpha, \ldots, \alpha^{\mathsf{par}\ell-1})$ is a proximity generator for $\mathcal{C}$ with mutual correlated agreement with proximity bound $\mathsf{B}^\star(\mathcal{C}, \mathsf{par}\ell) := \frac{1+\rho}{2}$ and error $\mathsf{err}^\star(\mathcal{C}, \mathsf{par}\ell, \delta) = \frac{(\mathsf{par}\ell-1)\cdot 2^m}{\rho\cdot|\mathbb{F}|}$.*

**Theorem 164.** *The function $\mathsf{Gen}(\mathsf{par}\ell; \alpha) := (1, \alpha, \ldots, \alpha^{\mathsf{par}\ell-1})$ is a proximity generator with mutual correlated agreement for every smooth Reed Solomon code $\mathcal{C} := \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ (with rate $\rho := 2^m/|\mathcal{L}|$). We give two conjectures, for the parameters of the proximity bound $\mathsf{B}^\star$ and the error $\mathsf{err}^\star$:*

1. *Up to the Johnson bound: $\mathsf{B}^\star(\mathcal{C}, \mathsf{par}\ell) := \sqrt{\rho}$, and*

$$\mathsf{err}(\mathcal{C}, \mathsf{par}\ell, \delta) := \frac{(\mathsf{par}\ell - 1)\cdot 2^m}{|\mathbb{F}|\cdot\left(2\cdot\min\left\{1-\sqrt{\rho}-\delta, \frac{\sqrt{\rho}}{20}\right\}\right)^7}.$$

2. *Up to capacity: $\mathsf{B}^\star(\mathcal{C}, \mathsf{par}\ell) := \rho$, and there exist constants $c_1, c_2, c_3 \in \mathbb{N}$ such that for every $\eta > 0$ and $0 < \delta < 1 - \rho - \eta$:*

$$\mathsf{err}^\star(\mathcal{C}, \mathsf{par}\ell, \delta) := \frac{(\mathsf{par}\ell - 1)^{c_2}\cdot\delta^{c_2}}{\eta^{c_1}\cdot\rho^{c_1+c_2}\cdot|\mathbb{F}|}.$$

### Mutual correlated agreement preserves list decoding

**Lemma 165.** *Let $\mathcal{C} \subseteq \mathbb{F}^{\mathcal{L}}$ be a linear code with minimum distance $\delta_{\mathcal{C}}$, and let $\mathsf{Gen}$ be a proximity generator for $\mathcal{C}$ with mutual correlated agreement with proximity bound $\mathsf{B}^\star$ and error $\mathsf{err}^\star$. Then, for every $f_0, \ldots, f_{\mathsf{par}\ell-1} : \mathcal{L} \to \mathbb{F}$ and $\delta \in (0, \min\{\delta_{\mathcal{C}}, 1 - \mathsf{B}^\star(\mathcal{C}, \mathsf{par}\ell)\})$:*

$$\Pr_{\substack{\alpha\leftarrow\{0,1\}^{w^\star}\\ r:=\mathsf{Gen}(\mathsf{par}\ell;\alpha)}}\left[\Lambda\left(\mathcal{C}, \sum_{j\in[0,(\mathsf{par}\ell-1)]} r_j\cdot f_j, \delta\right) \neq \left\{\sum_{j\in[0,(\mathsf{par}\ell-1)]} r_j\cdot u_j : u \in \Lambda\left(\mathcal{C}^\ell, (f_0, \ldots, f_{\mathsf{par}\ell-1}), \delta\right)\right\}\right] \leq \mathsf{err}^\star(\mathcal{C}, \mathsf{par}\ell, \delta).$$

### Folding univariate functions

**Definition 166.** Let $\mathsf{extract} : \mathcal{L}^{2^{k+1}} \to \mathcal{L}^{2^k}$ be a function. There exists $x \in \mathcal{L}$, such that $y = x^{2^{k+1}} \in \mathcal{L}^{2^{k+1}}$. Then $\mathsf{extract}$ returns $z = \sqrt{y} = x^{2^k} \in \mathcal{L}^{2^k}$ such that $y = z^2$.

**Definition 167.** Let $f : \mathcal{L}^{2^k} \to \mathbb{F}$ be a function, and $\alpha \in \mathbb{F}$. We define $\mathrm{Fold}_{\mathrm{f}}(f, \alpha) : \mathcal{L}^{(2^{k+1})} \to \mathbb{F}$ as follows:

$$\forall x \in \mathcal{L}^{2^k}, y \in \mathcal{L}^{2^{k+1}}, \quad \mathrm{Fold}_{\mathrm{f}}(f, \alpha)(y) := \frac{f(x) + f(-x)}{2} + \alpha\cdot\frac{f(x) - f(-x)}{2\cdot x}.$$

In order to compute $\mathrm{Fold}_{\mathrm{f}}(f, \alpha)(y)$ it suffices to query $f$ at $x$ and $-x$, by retrieving $x = \mathsf{extract}(y)$.

**Definition 168.** For $k \leq m$ and $\vec{\alpha} = (\alpha_0, \ldots, \alpha_{k-1}) \in \mathbb{F}^k$ we define $\mathrm{Fold}(f, \vec{\alpha}) : \mathcal{L}^{2^k} \to \mathbb{F}$ to equal $\mathrm{Fold}(f, \vec{\alpha}) := f_k$ where $f_k$ is defined recursively as follows: $f_0 := f$, and $f_i := \mathrm{Fold}_{\mathrm{f}}(f_{i-1}, \alpha_i)$.

**Definition 169.** For a set $S \subseteq \mathbb{F}^{\mathcal{L}}$ we denote $\mathrm{Fold}_{\mathrm{S}}(S, \vec{\alpha}) := \{\mathrm{Fold}_{\mathrm{S}}(f, \vec{\alpha}) \mid f \in S\}$.

**Lemma 170.** *Let $f : \mathcal{L} \to \mathbb{F}$ be a function, $\vec{\alpha} \in \mathbb{F}^k$ folding randomness and let $g := \mathrm{Fold}(f, \vec{\alpha})$. If $f \in \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ and $k \leq m$, then $g \in \mathrm{RS}[\mathbb{F}, \mathcal{L}^{2^k}, m-k]$, and further the multilinear extension of $g$ is given by $\hat{g}(X_k, \ldots, X_{m-1}) := \hat{f}(\vec{\alpha}, X_k, \ldots, X_{m-1})$ where $\hat{f}$ is the multilinear extension of $f$.*

**Block relative distance**

**Definition 171.** Let $\mathcal{L} \subseteq \mathbb{F}$ be a smooth evaluation domain and $k \in \mathbb{N}$ be a folding parameter. For $z \in \mathcal{L}^{2^k}$, define $\mathrm{Block}(\mathcal{L}, i, k, z) := \{x \in \mathcal{L}, y \in \mathcal{L}^{2^i} : y^{2^{k-i}} = z\}$.

**Definition 172.** Let $\mathcal{C} := \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ be a smooth Reed Solomon code and let $f, g : \mathcal{L}^{2^i} \to \mathbb{F}$. We define the $(i,k)$-wise block relative distance as

$$\Delta_r(\mathcal{C}, i, k, f, g) = \frac{\left|\left\{z \in \mathcal{L}^{2^k} : \exists y \in \mathrm{Block}(\mathcal{L}, i, k, z), f(y) \neq g(y)\right\}\right|}{|\mathcal{L}^{2^k}|}$$

**Definition 173.** For $S \subseteq \mathbb{F}^{\mathcal{L}}$, we let $\Delta_r(\mathcal{C}, i, k, f, S) := \min_{g \in S} \Delta_r(\mathcal{C}, i, k, f, g)$.

Note that $\Delta_r(\mathcal{C}, 0, 0, f, g) = \Delta(f, g)$ for any $\mathcal{C}$. We define the block list decoding of a codeword.

**Definition 174.** For a smooth Reed Solomon code $\mathrm{RS} := \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$, proximity parameter $\delta \in [0,1]$, and $f : \mathcal{L}^{2^i} \to \mathbb{F}$, we let

$$\Lambda_r(\mathcal{C}, i, k, f, \delta) := \{u \in \mathcal{C} \mid \Delta_r(\mathcal{C}, i, k, f, u) \leq \delta\},$$

denote the list of codewords in $\mathcal{C}$ within relative block distance at most $\delta$ from $f$.

**Lemma 175.** *For any $\mathcal{C} := \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$, $k \in \mathbb{N}$, and $f, g : \mathcal{L}^{2^i} \to \mathbb{F}$, we have that $\Delta(f, g) \leq \Delta_r(\mathcal{C}, i, k, f, g)$. Consequently, $\Lambda_r(\mathcal{C}, i, k, f, \delta) \subseteq \Lambda(\mathcal{C}, f, \delta)$ for $\delta \in [0,1]$.*

**Folding preserves list decoding**

**Theorem 176.** *Let $\mathcal{C} = \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ be a smooth Reed Solomon code and $k \leq m$. For $0 \leq i \leq k$ let $\mathcal{C}^{(i)} := \mathrm{RS}[\mathbb{F}, \mathcal{L}^{2^i}, m-i]$. Let $\mathsf{Gen}(\mathsf{par}\ell; \alpha) = (1, \alpha, \ldots, \alpha^{\mathsf{par}\ell - 1})$ be a proximity generator with mutual correlated agreement for the codes $\mathcal{C}^{(0)}, \ldots, \mathcal{C}^{(k-1)}$ with proximity bound $\mathsf{B}^\star$ and error $\mathsf{err}^\star$. Then for every $f : \mathcal{L} \to \mathbb{F}$ and $\delta \in \left(0, 1 - \max_{i \in [0,(k-1)]}\{\mathsf{B}^\star(\mathcal{C}^{(i)}, 2)\}\right)$,*

$$\Pr_{\alpha \leftarrow \mathbb{F}^k}\left[\mathrm{Fold}_{\mathrm{S}}(\Lambda_r(\mathcal{C}, 0, k, f, \delta), \alpha) \neq \Lambda(\mathcal{C}^{(k)}, \mathrm{Fold}(f, \alpha), \delta)\right] < \mathsf{err}^{(k)}(\mathcal{C}, \delta).$$

**Lemma 177.** *Let $\mathcal{C} := \mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ be a Reed Solomon code, and $k \leq m$ be a parameter. Denote $\mathcal{C}' := \mathrm{RS}[\mathbb{F}, \mathcal{L}^2, m-1]$. Then for every $f : \mathcal{L} \to \mathbb{F}$ and $\delta \in (0, 1 - \mathsf{B}^\star(\mathcal{C}', 2))$,*

$$\Pr_{\alpha \leftarrow \mathbb{F}}\left[\mathrm{Fold}_{\mathrm{S}}(\Lambda_r(\mathcal{C}, 0, k, f, \delta), \alpha) \neq \Lambda_r(\mathcal{C}', 1, k, \mathrm{Fold}(f, \alpha), \delta)\right] < \mathsf{err}^\star(\mathcal{C}', 2, \delta).$$

**Lemma 178.** *For every $\alpha \in \mathbb{F}$, $\mathrm{Fold}_{\mathrm{S}}(\Lambda_r(\mathcal{C}, 0, k, f, \delta), \alpha) \subseteq \Lambda_r(\mathcal{C}', 1, k, \mathrm{Fold}(f, \alpha), \delta)$.*

**Lemma 179.**

$$\Pr_{\alpha \leftarrow \mathbb{F}}\left[\Lambda_r(\mathcal{C}', 1, k, \mathrm{Fold}(f, \alpha), \delta) \not\subseteq \mathrm{Fold}_{\mathrm{S}}(\Lambda_r(\mathcal{C}, 0, k, f, \delta), \alpha)\right] < \mathsf{err}^\star(\mathcal{C}', 2, \delta).$$

**Lemma 180.** *Let $f : \mathcal{L} \to \mathbb{F}$ be a function, $m \in \mathbb{N}$ be a number of variables, $s \in \mathbb{N}$ be a repetition parameter, and let $\delta \in [0,1]$ be a distance parameter. For every $\vec{r_0}, \ldots, \vec{r_{s-1}} \in \mathbb{F}^m$, the following are equivalent statements.*

- *There exist distinct $u, u' \in \Lambda(\mathrm{RS}[\mathbb{F}, \mathcal{L}, m], f, \delta)$ such that, for every $i \in [0, s-1]$, $\widehat{u}(\vec{r_i}) = \widehat{u}'(\vec{r_i})$.*

- *There exists* $\sigma_0, \ldots, \sigma_{s-1} \in \mathbb{F}$ *such that*

$$|\Lambda(\mathrm{CRS}[\mathbb{F}, \mathcal{L}, m, ((Z \cdot \mathrm{eq}(\vec{r_0}, \cdot), \sigma_0), \ldots, (Z \cdot \mathrm{eq}(\vec{r_{s-1}}, \cdot), \sigma_{s-1}))], f, \delta)| > 1.$$

**Lemma 181.** *Let* $f : \mathcal{L} \to \mathbb{F}$ *be a function,* $m \in \mathbb{N}$ *be a number of variables,* $s \in \mathbb{N}$ *be a repetition parameter, and* $\delta \in [0, 1]$ *be a distance parameter. If* $\mathrm{RS}[\mathbb{F}, \mathcal{L}, m]$ *is* $(\delta, \ell)$*-list decodable then*

$$\Pr_{r_0, \ldots, r_{s-1} \leftarrow \mathbb{F}} \left[ \begin{array}{c} \exists \sigma_0, \ldots, \sigma_{s-1} \in \mathbb{F} \text{ s.t.} \\ \left|\Lambda(\mathrm{CRS}[\mathbb{F}, \mathcal{L}, m, ((Z \cdot \mathrm{eq}(\mathrm{pow}(r_i, m), \cdot), \sigma_i))_{i \in [s]}], f, \delta)\right| > 1 \end{array} \right]$$

$$= \Pr_{r_0, \ldots, r_{s-1} \leftarrow \mathbb{F}} \left[ \begin{array}{c} \exists \text{ distinct } u, u' \in \Lambda(\mathrm{RS}[\mathbb{F}, \mathcal{L}, m], f, \delta) \\ s.t.\ \forall i \in [s],\ \hat{u}(\mathrm{pow}(r_i, m)) = \hat{u}'(\mathrm{pow}(r_i, m)) \end{array} \right]$$

$$\leq \frac{\ell^2}{2} \cdot \left(\frac{2^m}{|\mathbb{F}|}\right)^s.$$

**Theorem 182.** *Consider* $(\mathbb{F}, M, (k_i, m_i, \mathcal{L}_i, t_i)_{0 \leq i \leq M}, \widehat{w}_0, \sigma_0, m, d^\star, d)$ *with the following ingrediants and conditions,*

- *a constrained Reed Solomon code* $\mathrm{CRS}[\mathbb{F}, \mathcal{L}_0, m_0, \widehat{w}_0, \sigma_0]$*;*
- *an iteration count* $M \in \mathbb{N}$*;*
- *folding parameters* $k_0, \ldots, k_M$ *such that* $\sum_{i=0}^{M} k_i \leq m$*;*
- *evaluation domains* $\mathcal{L}_0, \ldots, \mathcal{L}_M \subseteq \mathbb{F}$ *where* $\mathcal{L}_i$ *is a smooth coset of* $\mathbb{F}^*$ *with order* $|\mathcal{L}_i| \geq 2^{m_i}$*;*
- *repetition parameters* $t_0, \ldots, t_M$ *with* $t_i \leq |\mathcal{L}_i|$*;*
- *define* $m_0 := m$ *and* $m_i := m - \sum_{j<i} k_j$*;*
- *define* $d^\star := 1 + \deg_{\mathbb{Z}}(\widehat{w}_0) + \max_{i \in [m_0]} \deg_{X_i}(\widehat{w}_0)$ *and* $d := \max\{d^\star, 3\}$.

*For every* $f \notin \mathrm{CRS}[\mathbb{F}, \mathcal{L}_0, m_0, \widehat{w}_0, \sigma_0]$ *and every* $\delta_0, \ldots, \delta_M$ *and* $(\mathsf{par}\ell_{i,s})_{0 \leq i \leq M}^{0 \leq s \leq k_i}$ *where*

- $\delta_0 \in (0, \Delta(f, \mathrm{CRS}[\mathbb{F}, \mathcal{L}_0, m_0, \widehat{w}_0, \sigma_0]))$*;*
- *the function* $\mathsf{Gen}(\mathsf{par}\ell; \alpha) = (1, \alpha, \ldots, \alpha^{\mathsf{par}\ell - 1})$ *is a proximity generator with mutual correlated agreement for the codes* $(\mathcal{C}_{\mathrm{RS}}^{(i,s)})_{0 \leq i \leq M}^{0 \leq s \leq k_i}$ *where* $\mathcal{C}_{\mathrm{RS}}^{(i,s)} := \mathrm{RS}[\mathbb{F}, \mathcal{L}_i^{(2^s)}, m_i - s]$ *with bound* $\mathsf{B}^\star$ *and error* $\mathsf{err}^\star$*;*
- *for every* $0 \leq i \leq M$*,* $\delta_i \in (0, 1 - \mathsf{B}^\star(\mathcal{C}_{\mathrm{RS}}^{(i,s)}, 2))$*;*
- *for every* $0 \leq i \leq M$*,* $\mathcal{C}_{\mathrm{RS}}^{(i,s)}$ *is* $(\ell_{i,s}, \delta_i)$*-list decodable.*

*Then there exists an IOPP for* $\mathrm{CRS}[\mathbb{F}, \mathcal{L}_0, m_0, \widehat{w}_0, \sigma_0]$ *with above parameters, with round-by-round soundness error*

$$((\varepsilon_{0,s}^{\mathrm{fold}})_{s \leq k_0},\ (\varepsilon_i^{\mathrm{out}},\ \varepsilon_i^{\mathrm{shift}})_{i \leq M},\ (\varepsilon_{i,s}^{\mathrm{fold}})_{i \in [M], s \leq k_i},\ \varepsilon^{\mathrm{fin}}),$$

*where:*

- $\varepsilon_{0,s}^{\text{fold}} \leq \dfrac{d^* \cdot \ell_{0,s-1}}{|\mathbb{F}|} + \text{err}^*(\mathcal{C}_{\text{RS}}^{(0,s)}, 2, \delta_0)$;
- $\varepsilon_i^{\text{out}} \leq \dfrac{2^{m_i} \cdot \ell_{i,0}^2}{2 \cdot |\mathbb{F}|}$;
- $\varepsilon_i^{\text{shift}} \leq (1-\delta_{i-1})^{t_i - 1} + \dfrac{\ell_{i,0} \cdot (t_i - 1 + 1)}{|\mathbb{F}|}$;
- $\varepsilon_{i,s}^{\text{fold}} \leq \dfrac{d \cdot \ell_{i,s-1}}{|\mathbb{F}|} + \text{err}^*(\mathcal{C}_{\text{RS}}^{(i,s)}, 2, \delta_i)$;
- $\varepsilon^{\text{fin}} \leq (1-\delta_{M-1})^{t_M - 1}$.

## 3.7 The Spartan Protocol

## 3.8 The Ligero Polynomial Commitment Scheme

- $pp \leftarrow \mathsf{Setup}(1^\lambda)$: I
- $b \leftarrow \langle \mathcal{P}(w), \mathcal{V}(r) \rangle ($
  1. $\mathcal{P} : (\mathcal{C}, \mathcal{S}) \leftarrow$ F
  2. $\mathcal{V} : \tau \in_R \mathbb{F}^{\log m}$
  3. Let $T_1 = 0, \mu_1$
  4. $\mathcal{V}$ : Sample $r_x \in$
  5. **Sum-check#1.**
  6. $\mathcal{P}$: Compute $v_A$

# 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 183** (Multilinear Extension)**.**

**Theorem 184** (Multilinear Extension is Unique)**.**

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

**Theorem 185** (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 186** (Computable Univariate Polynomials)**.**

**Definition 187** (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 188** (Code Distance)**.**

**Definition 189** (Distance from a Code)**.**

**Definition 190** (Generator Matrix)**.**

**Definition 191** (Parity Check Matrix)**.**

**Definition 192** (Code)**.**

**Definition 193** (Linear Code)**.**

**Definition 194** (Interleaved Code)**.**

**Definition 195** (Reed-Solomon Code)**.**

**Definition 196** (Smooth Reed-Solomon Code)**.**

**Definition 197** (Constrained Code)**.**

**Definition 198** (Multi-constrained Code)**.**

**Definition 199** (Proximity Measure)**.**

**Definition 200** (Proximity Gap)**.**

**Definition 201** (List Decodability)**.**

**Definition 202** (List of Close Codewords)**.**

## 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 203** (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 204** (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 205** (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 206** (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 207** (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 208** (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 209** (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] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. In *Advances in Cryptology – EUROCRYPT 2025: 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4–8, 2025, Proceedings, Part IV*, page 93–122, Berlin, Heidelberg, 2025. Springer-Verlag.

[10] J.L. Fan and C. Paar. On efficient inversion in tower fields of characteristic two. In *Proceedings of IEEE International Symposium on Information Theory*, pages 20–, 1997.

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

[12] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

[13] Doug Wiedemann. An iterated quadratic extension of gf(2). *The Fibonacci Quarterly*, 26(4):290–295, 1988.