Formally Verified SNARKs in Lean

February 17, 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). We plan to build a general framework for IOP-based SNARKs that can state specifications of the protocols and prove their security properties in a clean and modular way.

Chapter 2

Oracle Reductions

2.1 Definitions

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

2.1.1 Format

An oracle reduction is an interactive protocol between two parties, a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$. Its format is as follows:

1. The protocol structure 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.

2. All messages from $\mathcal{V}$ are chosen uniformly at random (more generally, from some fixed probability distribution).

3. 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).

4. The prover and verifier has access to some inputs at the beginning of the protocol. These inputs are classified as follows:

   - *Public inputs*: available to both parties;
   - *Private inputs* (or *witness*): available only to the prover;
   - *Oracle inputs*: the underlying data is available to the prover, but it's only exposed as an oracle to the verifier.
   - *Shared oracle*: the oracle is available to both parties via an interface, and it may be randomized.

We collect all the public inputs, private inputs, and oracles into a *context*.

Definition 1 (Context)**.** In an oracle reduction, its *context* consists of a list of public inputs, a list of witness inputs, a list of oracle inputs, and a shared oracle (possibly represented as a list of lazily sampled query-response pairs). These inputs have the expected visibility.

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

Definition 2 (Type Signature of an Oracle Reduction)**.** An *n-message oracle reduction* between two parties $\mathcal{P}$ and $\mathcal{V}$ consists of a sequence of messages $m_0, \ldots, m_n$, where each message m_i (of a given type) is associated with a *direction* (to $\mathcal{P}$ or to $\mathcal{V}$), and a message visibility (public or oracle) if coming from $\mathcal{P}$.

Definition 3 (Type Signature of a Prover)**.** A prover $\mathcal{P}$ in an oracle reduction, given a context, is a stateful oracle computation that at each step of the protocol, either takes in a new message from the verifier, or sends a new message to the verifier.

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

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

Definition 4 (Type Signature of a Verifier)**.** A verifier $\mathcal{V}$ in an oracle reduction is an oracle computation that may perform a series of checks (i.e. 'Bool'-valued, or 'Option Unit') on the given context.

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

Definition 5 (Interactive Oracle Reduction)**.** An interactive oracle reduction is a combination of a type signature `ProtocolSpec`, a prover for `ProtocolSpec`, and an oracle verifier for `ProtocolSpec`.

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

Remark 7 (More efficient representation of oracle reductions)**.** The presentation of oracle reductions as protocols on an append-only context is useful for reasoning, but it does not lead to the most efficient implementation for the prover and verifier. In particular, the prover cannot keep intermediate state, and thus needs to recompute everything from scratch for each new message.

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

2.1.2 Security properties

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

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

Definition 8 (Completeness)**.**

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

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

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

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

Definition 9 (Soundness)**.**

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

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

Definition 10 (Knowledge Soundness)**.**

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

Definition 11 (State Function)**.**

Definition 12 (Round-by-Round Soundness)**.**

Definition 13 (Round-by-Round Knowledge Soundness)**.**

2.2 A Program Logic for Oracle Reductions

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

The first of these rules is *sequential composition*.

The second is *virtualization*, which allow for reductions on *virtual* or *ghost* values derivable from the actual values in the context.

The third is *substitution (?)*, which allows for substituting a value in the context with another value, followed by a reduction establishing the relationship between the new and old values.

We will also consider weakening / strengthening / framing of predicates on contexts.

2.2.1 Sequential Composition

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

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

2.2.2 Virtualization

See the section on sum-check 3.1 for an example.

Definition 14 (Mapping into Virtual Context)**.** In order to apply an oracle reduction on virtual data, we will need to provide a mapping from the current context to the virtual context. This includes:

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

Definition 15 (Virtual Oracle Reduction)**.** Given a suitable mapping into a virtual context, we may define an oracle reduction via the following construction:

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

We will show security properties for this virtualization process. One can see that completeness and soundness are inherited from the completeness and soundness of the virtual oracle reduction. However, knowledge soundness is more tricky; this is because we must go back

2.2.3 Substitution

This transformation happens when we substitute a value in the context with another value, followed by a reduction establishing the relationship between the new and old values.

Examples include:

1. Substituting an oracle input by a public input:
    - Often by just revealing the underlying data. This has no change on the prover, and for the verifier, this means that any query to the oracle input can be locally computed.
    - A variant of this is when the oracle input consists of a data along with a proof that the data satisfies some predicate. In this case, the verifier needs to additionally check that the predicate holds for the substituted data.
    - Another common substitution is to replace a vector with its Merkle commitment, or a polynomial with its polynomial commitment.

2. Substituting an oracle input by another oracle input, followed by a reduction for each oracle query the verifier makes to the old oracle:

    - This is also a variant of the previous case, where we do not fully substitue with a public input, but do a "half-substitution" by substituting with another oracle input. This happens e.g. when using a polynomial commitment scheme that is itself based on a vector commitment scheme. One can cast protocols like Ligero / Brakedown / FRI / STIR in this two-step process.

Chapter 3

Proof Systems

3.1 The Sum-Check Protocol

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

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

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

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

$$P(r) = v,$$

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

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

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

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

Theorem 16. *The sum-check protocol is complete.*

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

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

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

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

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

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

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

- The univariate polynomial p is instantiated as $\sum_{x \in \{0,1\}^{n-i-1}} P(r_1, \ldots, r_{i-1}, X, x)$.

- The scalar d is instantiated as c if $i = 0$, and as $s_{i-1}(r_{i-1})$ otherwise.

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

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

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

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

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

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

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

3.2 The Spartan Protocol

3.3 The Ligero Polynomial Commitment Scheme

Chapter 4

Commitment Schemes

4.1 Definitions

4.2 The Merkle Commitment Scheme

Chapter 5

Supporting Theories

5.1 Polynomials

Definition 17 (Multilinear Extension).

Theorem 18 (Multilinear Extension is Unique).

5.2 Coding Theory

Definition 19 (Code Distance).

Definition 20 (Distance from a Code).

Definition 21 (Generator Matrix).

Definition 22 (Parity Check Matrix).

Definition 23 (Interleaved Code).

Definition 24 (Reed-Solomon Code).

Definition 25 (Proximity Measure).

Definition 26 (Proximity Gap).

Chapter 6

References