RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity

Jeremy Bruestle, Paul Gafni, and the RISC Zero Team

June 30, 2023

Abstract

RISC Zero offers an open source virtual machine coupled with a zero-knowledge proof system. Together this is referred to as a zero-knowledge virtual machine, or *zkVM*. When a RISC-V binary executes inside the zkVM, the output is paired with a computational receipt, which serves as a zero-knowledge argument of computational integrity. The RISC Zero proof system implements a zk-STARK instantiated using the FRI protocol, DEEP-ALI, and an HMAC-SHA-256 based PRF. In this paper, we formally articulate the RISC Zero Proof System with as few technical barriers to understanding as possible, including a detailed construction of a RISC Zero cryptographic seal.

Contents

Acknowledgements

Many thanks to Bolton Bailey, Lennart Beringer, Tim Carstens, Victor Graf, Tim Zerrell, and the rest of the RISC Zero team for their support and feedback.

1 Introduction

1.1 Verifiable Computing

The era of verifiable computing is upon us: we now live in a world where the actions of untrusted parties in distributed systems can be authenticated as trustworthy quickly and easily, using *arguments of computational integrity* [Ben+18]. This technology is the result of decades of incremental progress in the field of zero-knowledge cryptography and interactive proofs [GMR85; AS98; BCS16]. Over the past few years, it has become practical and impactful to implement in real-world applications [Ben+14; Sta21]. Initial applications showed zero-knowledge proofs to be a powerful tool for ensuring privacy in blockchains, and the technology has evolved to the point that arguments of computational integrity are staged to become key infrastructure to the digital world. We expect that verifiable computation is not only the answer to the question of blockchain scaling, but also to fixing problems like Twitter bots and telephone spam. In a world where trust is becoming more and more scarce, verifiable computing provides a path forward.

Thousands upon thousands of developers are eager to start writing verifiable software. But current systems for writing verifiable software require developers to learn brand new languages with various limitations and challenges. In order to make verifiable software development possible in languages like Rust and C++, **RISC Zero has built a mechanism for demonstrating the integrity of any RISC-V computation.**

1.2 Verifiable RISC-V

When a RISC-V binary executes in the RISC Zero zkVM, the output is paired with a computational receipt. The receipt contains a cryptographic seal, which serves as a zero-knowledge argument of computational integrity. By offering computational receipts for any code that will compile to RISC-V, our zkVM offers a platform to build truly trustable software, where skeptical third parties can verify authenticity in milliseconds. The idea of "running a verifier" to assure the integrity of massive computations is a novel addition to the world of digital security.

1.3 A Formally Verified Verifier

Formal verification is a process of translating code into a model that can be reasoned about with mathmatical tools to ensure properties relevant to security and correctness. Formal verification is used to ensure that a piece of code is actually doing what it's supposed to be doing. To ensure that the RISC Zero verifier (which checks the receipts) is functioning without security bugs, we're working on a formal verification of the verifier implementation. This formal verification work is available at `https://www.github.com/risc0/risc0-lean4`.

1.4 The RISC Zero Argument System

RISC Zero's argument system is based on [Ben+18]; the seal on a RISC Zero receipt is a zk-STARK. The arithmetization scheme is a randomized AIR with preprocessing [Azt20]. The randomized preprocessing constructs constraints for a PLONK-based memory permutation argument and a PLOOKUP-based range-check [GWC19; GW20]. After preprocessing, the STARK is instantiated using the DEEP-ALI protocol and the FRI protocol [Ben+19; Ben+17].

1.5 Paper Organization

In this paper, we specify the RISC Zero protocol and analyze its knowledge soundness. The paper is organized as follows:

In Section 2, we introduce some key ideas and background. In particular, we introduce the following idea: *An assertion of computational integrity can be viewed as an assertion that an execution trace satisfies a set of constraints.*

In Section 3, we present the interactive version of the protocol and analyze its soundness in the Interactive Oracle Proof model [BCS16].

2 Background

The argument system behind RISC Zero's receipts is built in terms of an *execution trace* and a number of *constraints* that enforce checks of computational integrity. We start by introducing those terms as they're used in the context of the RISC Zero protocol.

2.1 The Columns of the Execution Trace

When a piece of code runs on a machine, the *execution trace* (or simply, the *trace*) is a record of the full state of the machine at each clock cycle of the computation. It's typical to write an execution trace as a rectangular array, where each row shows the complete state of the machine at a given moment in time, and each column shows a temporal record of some particular aspect of the computation (say, the value stored in a particular RISC-V register) at each clock cycle.

The columns of the RISC Zero execution trace are categorized as follows:

Control Columns - Public The data in these columns describe the RISC-V architecture, and various control signals that define the stage of execution and therefore what constraints are applied.[1]

Data Columns - Private The data in these columns represents the running state of the processor and memory. In order to efficiently check the integrity of RISC-V memory operations, each register associated with RISC-V memory

[1] In the source code, the Control Columns are referred to as "Code" Columns; we intend to update the terminology to align with this document.

operations has two associated columns: one in the original execution order and the second sorted first by memory location and then by clock-cycle.

Accumulator Columns - Private We use accumulators to instantiate grand-product constraints for a PLONK-based permutation check and a PLOOKUP-based range-check. The Accumulator Columns contain the associated accumulator data. The entries in the Accumulator Columns and the associated constraints are constructed during the randomized preprocessing phase [GWC19; GW20; Azt20].

2.2 Enforcing Computational Integrity via Constraints

An assertion of computational integrity can be re-framed as an assertion that an execution trace satisfies a certain set of *constraints*. These constraints enforce that the zkVM execution is consistent with the RISC-V ISA. Each constraint is a low-degree polynomial relation over the constrained values; the execution trace is valid if and only if each constraint evaluates to 0.

Example 1. The constraint $(k)(k-1) = 0$ enforces that k is either 0 or 1.

Example 2. The constraint $j - 2k = 0$ enforces that $j = 2k$.

Enforcing computational integrity of our implementation of the RISC-V instruction set architecture involves thousands of constraints, each of which is expressed as a multi-variable polynomial.[2] At a high level, the constraints enforce that each of the following phases of the zkVM operation was executed as claimed:

- Init: Initialization of all registers to 0
- Setup: Prepare to load ELF file
- Load: Loading the RISC-V binary into memory
- Reset: Ends the loading phase and prepares for execution.
- Body: Main execution phase. This is where user-defined code is executed.
- RamFini: Generates the memory-based grand product accumulation values for our memory permutation[GWC19]
- BytesFini: Generates the bytes-based grand product accumulation values for PLOOKUP[GW20]

Together, these constraints enforce that the purported output of the computation agrees with the expected rv32im [ISA19] execution.

2.3 Arguments and Proofs of Knowledge

The seal on the RISC Zero receipt is a STARK – a scalable, transparent argument of knowledge [Ben+18]. An argument of knowledge allows a Prover to justify an "assertion of knowledge." More formally, the Prover asserts knowledge of a witness

[2] Inputs to these polynomials may include register-values at various clock-cycles, including previous cycles.

$\mathbf{w}$ that satisfies some constraints $\mathbb{x}$. In the case of an assertion of computational integrity, the witness is the execution trace and the constraints are the various rules that define computational integrity.

In Section 3.3, we present the protocol as an Interactive Oracle Proof (IOP) [BCS16]. IOPs involve a series of interactions between a Prover and a Verifier, where the Prover's messages depend on randomness supplied by the Verifier at various points throughout the protocol. The IOP protocol is a theoretical model; in practice, the zkVM uses a non-interactive version of this protocol where the Verifier participation is replaced by HMAC-SHA-256 through the Fiat-Shamir transform.

In the context of the IOP protocol, the seal constitutes a proof of knowledge. In code, the *proof* becomes an *argument*.[3] More specifically, the seal is a STARK. The Fiat-Shamir Heuristic allows us to derive security results for our STARK based on soundness analysis of the IOP protocol [FS87].

3 The RISC Zero IOP Protocol

In this section, we present the interactive version of the RISC Zero protocol and analyze the knowledge soundness of the protocol in the context of the IOP model [BCS16].

3.1 Overview of the Protocol

In an interactive oracle proof, a Prover convinces a skeptical Verifier of some assertion via a series of interactions. In each round, the Prover commits to evaluations of one or more functions over a domain known to both parties. The Verifier may *query* these Prover messages without reading them in full. The IOP model idealizes these queries using the concept of *oracle access*. This theoretical model allows us to prove soundness of our protocol without reference to any cryptographic primitives. The seal on a RISC Zero receipt is the transcript of the interactive protocol, with a random oracle as the verifier. When the zkVM finishes execution of a method, the resultant seal serves as a zero-knowledge proof of computational integrity, linking a cryptographic imageID (which identifies the RISC-V binary file that was executed) to the asserted code output in a way that third-parties can quickly verify.

The protocol consists of a transparent *set-up phase*, a *randomized preprocessing phase*, and a *main phase*. The set-up phase establishes certain protocol parameters known to both the Prover and the Verifier, including the number and length of the trace columns as well as a full enumeration of the constraints that are to be enforced.

3.1.1 Set-up Phase

The set-up phase only needs to be done once for each program, and the public parameters can be used to generate an arbitrary number of execution proofs. These

[3] The argument depends on the security of HMAC-SHA-256 as a random oracle whereas the proof has no cryptographic assumptions.

public parameters include specification of the circuit and the constraints for the zkVM, a cryptographic identifier for the program to be executed, as well as various parameters that specify the handling of hashing, DEEP-ALI and FRI. These parameters can either be distributed to provers and verifiers via a trusted channel, or calculated independently from the program definition (e.g. source code). This ability to recalculate the parameters independently, using only public information, is what defines the "transparency" property.

3.1.2 Randomized Preprocessing

With the set-up complete, the Prover runs the program, which generates the control columns and the data columns, and then begins the randomized preprocessing, which constructs the accumulators for memory and for bytes. The Prover sends two trace commitments; one for the control columns and one for the data columns (see Section 2.1). To generate the trace commitments, the trace columns are encoded into trace blocks using Reed-Solomon encoding, and the trace blocks are committed to Merkle trees. We write $\mathbb{w}_{\mathsf{control}}$ and $\mathbb{w}_{\mathsf{data}}$ to represent the witnesses, and we write $\mathsf{Com}(\mathbb{w}_{\mathsf{control}})$ and $\mathsf{Com}(\mathbb{w}_{\mathsf{data}})$ to represent the associated commitments. After these first two commitments, the Prover uses verifier-randomness to construct the accum blocks and sends a third commitment $\mathsf{Com}(\mathbb{w}_{\mathsf{accum}})$, which allows for a permutation argument and a lookup argument.[4]

3.1.3 Main Phase

The main phase consists of a standard AIR-FRI protocol, using DEEP-ALI[Ben+19] and the batched FRI protocol[Ben+20], as in [Sta21] and [Tea22].

The Prover uses the constraints and the execution trace to generate the validity polynomial f_{validity}, uses f_{validity} to construct the validity witness $\mathbb{w}_{\mathsf{validity}}$, and sends $\mathsf{Com}(\mathbb{w}_{\mathsf{validity}})$ to the Verifier. The Verifier responds with a random $z \in \mathbb{F}_q$, the Prover evaluates f_{validity} at z and uses the DEEP quotienting technique to construct the $\mathsf{DEEPAnswerSequence}$.

3.2 IOP Definitions

Recall the definition of interactive oracle proof from [BCS16]. In the IOP model, our protocol satisfies the definition of STIK from [Ben+18].[5]

Building on the definitions and notation of an AIR from Section 5 of [Sta21], as well as the notion of a randomized AIR with preprocessing from [Azt20], our IOP proves knowledge of a witness that satisfies a RAP[Azt20] (randomized AIR with preprocessing) instance defined as

$$\mathsf{A}_{\mathsf{RAP}} = (\mathbb{F}, \mathbb{K}, \mathsf{e}, \mathsf{w}_{\mathsf{RAP}}, \mathsf{n}_{\sigma_{\mathsf{mem}}}, \mathsf{n}_{\sigma_{\mathsf{bytes}}}, \mathsf{h}, \mathsf{d}, \mathsf{s}, \omega, \beta, \mathsf{l}, \mathsf{C}_{\mathsf{set}})$$

where

[4] Because we reveal many branches from each tree, we commit Merkle caps rather than Merkle roots. See Appendix B and [CY21] for more details.

[5] The difference in acronyms between STIK and STARK is a substitution of "IOP" for "ARgument."

- $\mathbb{F} = \mathbb{F}_{2^{31}-2^{27}+1}$, known commonly as the Baby Bear field.
- $\mathbb{K}$ is a degree $\mathsf{e} = 4$ extension of $\mathbb{F}$.
- $\mathsf{w}_{\mathsf{RAP}}$ is the number of columns in the RAP witness. We write $\mathsf{w}_{\mathsf{RAP}} = \mathsf{w}_{\mathsf{control}} + \mathsf{w}_{\mathsf{data}} + \mathsf{w}_{\mathsf{accum}}$ where
  - $\mathsf{w}_{\mathsf{control}}$ is the number of control columns,
  - $\mathsf{w}_{\mathsf{data}}$ is the number of data columns, and
  - $\mathsf{w}_{\mathsf{accum}} = 2\mathsf{e} \cdot \mathsf{n}_{\sigma_{\mathsf{mem}}} + 2\mathsf{e} \cdot \mathsf{n}_{\sigma_{\mathsf{bytes}}}$ is the number of accumulator columns generated in the randomized preprocessing.[6]
- $\mathsf{n}_{\sigma_{\mathsf{mem}}}$ is the number of duplicated data columns in the witness that appear due to the permutation from $\mathsf{trace}_{\mathsf{time}}$ to $\mathsf{trace}_{\mathsf{mem}}$.
- $\mathsf{n}_{\sigma_{\mathsf{bytes}}}$ is the number of columns in the witness that appear due to the bytes-lookup in our PLOOKUP implementation.
- 2^{h} is the size of the trace domain, H (i.e., the length of the columns).
- d is the maximum degree of the rule-checking polynomials.
- $\omega \in \mathbb{F}$ has multiplicative order $\mathsf{e} \cdot 2^{\mathsf{h}}$ and $\beta \in \mathbb{F}$ is non-zero. We define the commitment domain D as the coset $\beta\mathsf{D}_0$ where $\mathsf{D}_0 \subseteq \mathbb{F}_p$ is generated by ω, and we define the trace domain H as the set generated by $\omega^{\frac{1}{\rho}}$, where ρ is defined below.
- I is the set of indices for the tapset[7].
- $\mathsf{C}_{\mathsf{set}}$ is the set of constraints which enforce the computational integrity checks. Each constraint, $C_i \in \mathbb{F}^{\leq \mathsf{d}}[\mathsf{Y}]$ is a multi-variate polynomial over the tapset variables, of total degree at most d, called the i^{th} rule-checking polynomial. Each constraint is enforced over the entire trace domain.
- s is the number of RAP constraints.

In addition to the inputs to the RAP instance listed above, the IOP also uses the following auxiliary inputs, denoted $\mathsf{aux} = (\mathsf{D}, \mathsf{k}, \mathsf{aux}_{\mathsf{FRI}})$.

- D is the zk commitment domain[8], which is a non-trivial coset[9] of a multiplicative subgroup D_0 where $\mathsf{H} \subseteq \mathsf{D}_0 \subseteq \mathbb{F}$.
- 2^{k} is the size of the zk commitment domain, D. We define the rate of the IOP by $\rho := \frac{2^{\mathsf{h}}}{2^{\mathsf{k}}}$.

[6] These accumulator columns are used for a PLONK-based permutation check and a PLOOKUP-based range check. In our system, $\mathsf{n}_{\sigma_{\mathsf{mem}}} = 5$ and $\mathsf{n}_{\sigma_{\mathsf{bytes}}} = 15$

[7] A tap is a reference to an entry in the trace; the constraints are expressed as a function of various taps. For more details on taps, see Appendix A.5.2. Note that ethSTARK uses the term *mask* where we use *tapset.*

[8] ethSTARK calls this the evaluation domain.

[9] Using D_0 as a commitment domain would introduce divide-by-zero issues and would mean queries might reveal information about the execution trace. The coset D doesn't intersect D_0, which is important for zero-knowledge purposes.

- $\mathsf{aux}_{\mathsf{FRI}}$ is defined as $(\vec{\mathsf{t}}, \mathsf{n}_{\mathsf{queries}}, \mathsf{d}_{\mathsf{final}})$ where
  - $\vec{\mathsf{t}} = (t_1, ..., t_r)$ is a vector describing the the FRI-folding factor for each round of the COMMIT phase. In our system, $t_i = 16$ for all i.
  - $\mathsf{n}_{\mathsf{queries}} = 50$ is the number of FRI queries, and
  - $\mathsf{d}_{\mathsf{final}} = 256$ is the degree at which we terminate our FRI algorithm.

3.3 IOP Protocol Specification

After a transparent set-up, the IOP consists of three components: a randomized preprocessing step, the DEEP-ALI protocol, and the FRI protocol.

3.3.1 Randomized Preprocessing

1. **Prover** generates IOP witness $\mathbb{w}_{\mathsf{control}} \cup \mathbb{w}_{\mathsf{data}}$ and sends commitments $\mathsf{Com}(\mathbb{w}_{\mathsf{control}})$ and $\mathsf{Com}(\mathbb{w}_{\mathsf{data}})$.[10]

2. **Verifier** samples $\mathsf{n}_\sigma = \mathsf{n}_{\sigma_{\mathsf{mem}}} + \mathsf{n}_{\sigma_{\mathsf{bytes}}}$ elements of $\mathbb{K}$, the randomness used to generate the Accumulator Columns.

3. **Prover** generates witness $\mathbb{w}_{\mathsf{accum}}$ and sends $\mathsf{Com}(\mathbb{w}_{\mathsf{accum}})$.

$\mathbb{w}_{\mathsf{RAP}} = \mathbb{w}_{\mathsf{control}} \cup \mathbb{w}_{\mathsf{data}} \cup \mathbb{w}_{\mathsf{accum}}$ is a witness that satisfies the RAP instance:

$$\mathsf{A}_{\mathsf{RAP}} = (\mathbb{F}, \mathbb{K}, \mathsf{e}, \mathsf{w}_{\mathsf{RAP}}, \mathsf{n}_{\sigma_{\mathsf{mem}}}, \mathsf{n}_{\sigma_{\mathsf{bytes}}}, \mathsf{h}, \mathsf{d}, \mathsf{s}, \omega, \beta, \mathsf{l}, \mathsf{C}_{\mathsf{set}})$$

The rest of the protocol implements DEEP-ALI + Batched FRI as in [Sta21], [Ben+19], [Ben+20], using auxiliary IOP parameters $\mathsf{aux} = (\mathsf{D}, \mathsf{k}, \mathsf{aux}_{\mathsf{FRI}})$.[11]

3.3.2 Sub-Protocol: DEEP-ALI

4. **Verifier** samples $\alpha_{\mathsf{constraints}} \in \mathbb{K}$, the randomness used in DEEP-ALI constraint batching.[12]

5. **Prover** generates $\mathbb{w}_{\mathsf{validity}}$ and sends $\mathsf{Com}(\mathbb{w}_{\mathsf{validity}})$.
This serves as a commitment to f_{validity}, defined as follows:

$$f_{\mathsf{validity}}(x) = \frac{\sum_{i=0}^{|\mathsf{C}_{\mathsf{set}}|-1} \alpha^i_{\mathsf{constraints}} \cdot \mathsf{C}_i(P_0(x), \ldots, P_{\mathsf{w}_{\mathsf{RAP}}-1})}{Z(x)}$$

where P_j are the interpolations of the columns of $\mathbb{w}_{\mathsf{RAP}}$, C_i are the constraints, and Z is the minimal polynomial that vanishes on the trace domain H.[13]

[10] For details on this step, see Appendix C.1.

[11] Unlike those references, but as in [Hab22] and [Tea22], we use powers of a single verifier randomness for constraint batching and a single verifier randomness for FRI batching.

[12] ethSTARK samples 2 random parameters per constraint, whereas we use a single randomness parameter here. See affine batching vs. parametric batching in [Hab22].

[13] In the source code, f_{validity} is called the CheckPoly. The Prover splits f_{validity} into four low-degree validity polynomials in order to construct $\mathsf{Com}(\mathbb{w}_{\mathsf{validity}})$. For details on this step, see Section C.5.2.

6. **Verifier** samples $z \in \mathbb{K} \backslash (\mathsf{H} \cup \mathsf{D})$ the random evaluation point used as a query for DEEP-ALI.

7. **Prover** uses z, $\mathbb{w}_{\mathsf{RAP}}$, and $\mathbb{w}_{\mathsf{validity}}$ to construct $\mathbb{w}_{\mathsf{BatchedFRI}}$. Rather than a Merkle commitment for $\mathbb{w}_{\mathsf{BatchedFRI}}$, the Prover sends enough information for the Verifier to reproduce the asserted values of $\mathbb{w}_{\mathsf{BatchedFRI}}$. This information is called the $\mathsf{DEEPAnswerSequence}$, and it consists of:

   (a) the full tapset of $\mathbb{w}_{\mathsf{RAP}}$ at the DEEP query point z,

   (b) the evaluation of $\mathbb{w}_{\mathsf{validity}}$ at the DEEP query point z, and

   (c) the coefficients of column-by-column interpolations of the tapset.[14]

3.3.3 Subprotocol: Batched FRI

8. **Verifier** samples $\alpha_{\mathsf{FRI}} \in \mathbb{K}$, the randomness used for FRI batching.

9. **Both parties** apply FRI with auxiliary information $\mathsf{aux}_{\mathsf{FRI}}$ to check proximity to the code $\mathsf{RS}[\mathbb{K}, \mathsf{D}, \rho]$ of the function

$$f_{\mathsf{FRI}}(x) = \sum_{i=0}^{\mathsf{w}_{\mathsf{RAP}}+3} \alpha_{\mathsf{FRI}}^{i} \cdot d_i(x)$$

   where d_i are the DEEP polynomials defined in Appendix C.5.5. This step involves a number of additional commitments from the **Prover**: one for the batching and one for each commit round of FRI.

3.3.4 Verification

Receipt verification consists of the following logical checks. If any of these checks fail, the verifier rejects the receipt.

1. Verifier uses the taps, $\mathsf{C}_{\mathsf{set}}$ and $\alpha_{\mathsf{constraints}}$ to compute $f_{\mathsf{validity}}(z)$ and checks for a mismatch against the purported value of $f_{\mathsf{validity}}(z)$ on the seal.[15]

2. Verifier checks the $\mathsf{DEEPAnswerSequence}$ for internal consistency, using the purported column-by-column interpolations to re-compute the purported tapset.

3. For each FRI query j, the Verifier uses the u coefficients and α_{FRI} to compute $f_{\mathsf{FRI}}(j)$ and checks for a mismatch against the purported value of $f_{\mathsf{FRI}}(j)$ on the seal.

4. Verifier checks the internals of each FRI query as described in Appendix C.6.

[14] These coefficients are called the u coefficients in the Rust code. There is one u polynomial per trace column, using the taps from that column. For details on this step, see Section C.5.5.

[15] f_{validity} is called the Check Poly in the Rust source.

3.4 Soundness Analysis in the IOP Model

We present soundness analysis of the IOP protocol using results from [Ben+20] and [Ben+19]. We use the Fiat-Shamir Heuristic to translate this IOP analysis into conclusions about our non-interactive protocol, instantiating the random oracle using HMAC-SHA-256.

Our soundness analysis follows that of [Sta21], aside from the following differences:

- **Randomized preprocessing**: Our protocol begins with a randomized preprocessing step which doesn't have an analog in [Sta21]. This randomized preprocessing instantiates constraints for a PLONK-based permutation check and a PLOOKUP-based range check. We bound the soundness error here using the Schwartz-Zippel Lemma.
- **Constraint Batching**: DEEP-ALI includes a step that compresses all the constraints into a single "Combined Constraint." As in [Hab22], our protocol uses powers of a single random field element whereas [Sta21] uses a vector of field elements. The technique we use is referred to as parametric batching in [Ben+20] whereas ethSTARK uses affine batching.
- **FRI Batching**: Similarly, the "batched FRI protocol" begins by using verifier randomness to compress a number of FRI instances into a single instance. As in [Hab22], our protocol uses parametric batching for the "FRI batching" step, whereas [Sta21] uses affine batching.
- **Degree reduction of $\mathbf{w}_{\mathsf{validity}}$**: The construction in [Ben+18] and [Sta21] results in a pair of FRI assertions, one for the trace ($\mathbf{w}_{\mathsf{control}} \cup \mathbf{w}_{\mathsf{data}} \cup \mathbf{w}_{\mathsf{accum}}$) and one for the "validity polynomial." As in [Hab22], we split[16] the validity polynomial into 4 low-degree validity polynomials so that we can compress this into a single FRI assertion. Each leaf of $\mathbf{w}_{\mathsf{validity}}$ contains one evaluation for each of the 4 low-degree validity polynomials. [Hab22] uses the term "segment polynomials" to refer to this splitting technique.

We bound the soundness of our protocol by

$$\epsilon_{\mathsf{IOP}} \leq \epsilon_{\mathsf{PLONK}} + \epsilon_{\mathsf{PLOOKUP}} + \epsilon_{\mathsf{DEEP-ALI}} + \epsilon_{\mathsf{FRI}}$$

where $\epsilon_{\mathsf{PLONK}}$, $\epsilon_{\mathsf{PLOOKUP}}$, $\epsilon_{\mathsf{DEEP-ALI}}$, and ϵ_{FRI} are the soundness error bounds for our PLOOKUP argument, our DEEP-ALI implementation, and our FRI implementation, respectively. In the following subsections, we bound $\epsilon_{\mathsf{PLONK}}$, $\epsilon_{\mathsf{PLOOKUP}}$, $\epsilon_{\mathsf{DEEP-ALI}}$, and ϵ_{FRI}, respectively.

3.4.1 Notation for Soundness Analysis

Using θ as the proximity parameter for the FRI low-degree test, the analysis in this section is proven to hold within the list-decoding radius ($\theta < 1 - \sqrt{\rho}$) and conjectured to hold up to code capacity ($\theta < 1 - \rho$). The proofs for these soundness results follow from Theorems 1.5 and 8.3 of [Ben+20] and Theorem 6.2 of [Ben+19]. The

[16] The split function is the same as the one used in the FRI commit rounds. For more details on this function, see Appendix A.2.3.

conjectured security follows from Conjecture 8.4 in [Ben+20] and Conjecture 2.3 of [Ben+19]. We include these theorems and conjectures in Appendix D.

In the following subsections, L and m come from the Guruswami-Sudan list-decoding algorithm, $\theta = 1 - \rho \cdot (1 + \frac{1}{2m})$ as in [Hab22], and all other variables are defined in Section 3.2. We point readers to [Hab22] and [Sta21] for more detailed descriptions of the DEEP-ALI and FRI error bounds, and to [Tea22] for more details on the error bounds for the permutation arguments.

3.4.2 Error Bound on PLONK and PLOOKUP

The randomized preprocessing phase instantiates accumulators and constraints for two grand-product arguments. One accumulator is used to prove memory integrity, and the other is used to prove tap integrity. For each, a simple application of the Schwartz-Zippel Lemma yields a bound on soundness error. We bound the error on the first by $\frac{\mathsf{e} \cdot \mathsf{n}_{\sigma_{\mathsf{mem}}} \cdot 2^{\mathsf{h}}}{|\mathbb{K}|}$ and the second by $\frac{\mathsf{e} \cdot \mathsf{n}_{\sigma_{\mathsf{bytes}}} \cdot 2^{\mathsf{h}}}{|\mathbb{K}|}$.

3.4.3 Error Bound on DEEP-ALI

In our implementation of DEEP-ALI, the verifier samples a single randomness $\alpha_{\mathsf{constraints}}$ for combining constraints and then $z \in \mathbb{K} \backslash (\mathsf{H} \cup \mathsf{D})$ as a DEEP query. Each of these random samplings has an associated error term. We write $\epsilon_{\mathsf{DEEP-ALI}} = \epsilon_{\mathsf{ALI}} + \epsilon_{\mathsf{DEEP}}$, where ϵ_{ALI} is the soundness error associated with combining constraints and ϵ_{DEEP} is the soundness error associated with the DEEP query point.

We differ from [Sta21] here in that we use powers of a single randomness for combining constraints. Our results here align with [Hab22], though since our constraints may access up to 5 rows of the trace, we find a different constant in the numerator in ϵ_{DEEP}. We find

$$\epsilon_{\mathsf{ALI}} = \frac{\mathsf{s} \cdot \mathsf{L}}{\mathbb{K}}$$

and

$$\epsilon_{\mathsf{DEEP}} = \frac{\mathsf{L} \cdot (4(\mathsf{k}+1) + (\mathsf{k}-1))}{|\mathbb{K}| - |\mathsf{H}| - |\mathsf{D}|}$$

3.4.4 Error Bound on FRI

The major results for the soundness of FRI in the list-decoding regime are Theorems 1.5 and 8.3 from [Ben+20]. As above, our soundness results for this part differ from the presentation in [Ben+20] and [Sta21] in that we use powers of a single random field element for FRI batching. Again, our results here align with [Hab22]; we find

$$\epsilon_{\mathsf{FRI}} = (\mathsf{w}_{\mathsf{RAP}} + 4 - \frac{1}{2}) \cdot \frac{(m + \frac{1}{2})^7}{3\sqrt{\rho}^3} \cdot \frac{|\mathsf{D}|^2}{|\mathbb{K}|} + \frac{(2m+1) \cdot (|\mathsf{D}|+1) \cdot \sum_{i=1}^{r} t_i}{\sqrt{\rho} \cdot |\mathbb{K}|} + (1-\theta)^{\mathsf{n}_{\mathsf{queries}}}$$

References

[RS60] Irving S. Reed and Gustave Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of The Society for Industrial and Applied Mathematics* 8 (1960), pp. 300–304.

[GMR85] S Goldwasser, S Micali, and C Rackoff. "The Knowledge Complexity of Interactive Proof-Systems". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing.* STOC '85. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 291–304. ISBN: 0897911512. DOI: `10.1145/22145.22178`. URL: `https://doi.org/10.1145/22145.22178`.

[FS87] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Proceedings on Advances in Cryptology—CRYPTO '86.* Santa Barbara, California, USA: Springer-Verlag, 1987, pp. 186–194. ISBN: 0387180478.

[AS98] Sanjeev Arora and Shmuel Safra. "Probabilistic Checking of Proofs: A New Characterization of NP". In: *J. ACM* 45.1 (Jan. 1998), pp. 70–122. ISSN: 0004-5411. DOI: `10.1145/273865.273901`. URL: `https://doi.org/10.1145/273865.273901`.

[Ben+14] Eli Ben Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy.* 2014, pp. 459–474. DOI: `10.1109/SP.2014.36`.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. *Interactive Oracle Proofs.* Cryptology ePrint Archive, Paper 2016/116. `https://eprint.iacr.org/2016/116`. 2016. URL: `https://eprint.iacr.org/2016/116`.

[Ben+17] Eli Ben-Sasson et al. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity". In: *Electron. Colloquium Comput. Complex.* TR17 (2017).

[Ben+18] Eli Ben-Sasson et al. "Scalable, transparent, and post-quantum secure computational integrity". In: *IACR Cryptol. ePrint Arch.* (2018), p. 46. URL: `http://eprint.iacr.org/2018/046`.

[Ben+19] Eli Ben-Sasson et al. *DEEP-FRI: Sampling outside the box improves soundness.* 2019. arXiv: `1903.12243 [cs.CC]`.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge.* Cryptology ePrint Archive, Report 2019/953. 2019.

[Azt20] Aztec. *From AIRs to RAPs - how PLONK-style arithmetization works.* 2020. URL: `https://hackmd.io/@aztec-network/plonk-arithmetiization-air`.

[Ben+20] Eli Ben-Sasson et al. "Proximity Gaps for Reed-Solomon Codes". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS).* 2020, pp. 900–909. DOI: `10.1109/FOCS46700.2020.00088`.

[GW20] Ariel Gabizon and Zachary J. Williamson. *plookup: A simplified polynomial protocol for lookup tables.* Cryptology ePrint Archive, Paper 2020/315. https://eprint.iacr.org/2020/315. 2020. URL: https://eprint.iacr.org/2020/315.

[CY21] Alessandro Chiesa and Eylon Yogev. *Subquadratic SNARGs in the Random Oracle Model.* Cryptology ePrint Archive, Paper 2021/281. https://eprint.iacr.org/2021/281. 2021. URL: https://eprint.iacr.org/2021/281.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. *Cairo – a Turing-complete STARK-friendly CPU architecture.* Cryptology ePrint Archive, Paper 2021/1063. https://eprint.iacr.org/2021/1063. 2021. URL: https://eprint.iacr.org/2021/1063.

[Sta21] StarkWare. *ethSTARK Documentation.* Cryptology ePrint Archive, Paper 2021/582. https://eprint.iacr.org/2021/582. 2021. URL: https://eprint.iacr.org/2021/582.

[Hab22] Ulrich Haböck. *A summary on the FRI low degree test.* Cryptology ePrint Archive, Paper 2022/1216. https://eprint.iacr.org/2022/1216. 2022. URL: https://eprint.iacr.org/2022/1216.

[ISA19] The RISC-V Instruction Set Manual Volume I: User-Level ISA. "Document Version 20191213". In: Editors Andrew Waterman and Krste Asanović (RISC-V Foundation, December 2019). https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf.

[Tea22] Polygon Zero Team. "Plonky2: Fast Recursive Arguments with PLONK and FRI". In: Draft (Sept 7, 2022). https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf.

Appendices

A Preliminaries

A.1 Cyclic Domains (Indexing in RISC Zero)

In this section, we define the indexing for an execution trace. RISC Zero's architecture is finite-field based, and the indexing is based on powers of $\omega \in \mathbb{F}_p$.

In order to index a sequence of l elements of $\mathbb{F}_q$, we choose an element of the base field $\omega \in \mathbb{F}_p$ with multiplicative order l and use the sequence $\{\omega^0, \omega^1, \ldots, \omega^{l-1}\}$ as an indexing set. This indexing is cyclical, in the sense that $\omega^z = \omega^{z-l}$ for any integer z. We call the indexing set a cyclic domain, which we formalize as follows:

Let $\mathbb{F}_p$ be a field of prime order, and let $\mathbb{F}_q$ be an extension of $\mathbb{F}_p$.

Definition 1 (Domain)**.** We call any non-empty subset $\mathcal{D} \subseteq \mathbb{F}_p$ a **domain**.

Definition 2 (Cyclic domain)**.** Given $\omega \in \mathbb{F}_p$, the set $\mathcal{D}(\omega) = \{w^i : i \in \mathbb{N}\}$ is a domain. If $\mathcal{D} = \mathcal{D}(\omega)$ for some non-zero $\omega \in \mathbb{F}_p$, then $\mathcal{D}$ is said to be a **cyclic domain** with **counter** ω.

Definition 3 (Cycle)**.** Given a cyclic domain $\mathcal{D}(\omega)$, we call c a **cycle** if $c \in \mathcal{D}(\omega)$.

We use the term "cycle" here as a step in the computation, related to a "processor cycle". This usage is distinct from the concept of a "cyclic domain".

We note that $|\mathcal{D}(\omega)|$ is equal to the multiplicative order of ω and therefore divides $|\mathbb{F}_p^*| = p - 1$.

Definition 4 (Cyclic Indexing)**.** Let $\mathcal{D}(\omega) \subseteq \mathbb{F}_p$ be the cyclic domain with counter ω. We use the powers of ω to define an indexing for $\mathcal{D}(w)$:

$$\begin{aligned} \text{Index-Cycle} : \mathbb{N} &\longrightarrow \mathcal{D}(\omega) \\ \text{Index-Cycle} : n &\longmapsto \omega^n \end{aligned}$$

This cyclic indexing on $\mathcal{D}(\omega)$ serves as our notion of sequential time throughout the protocol. We write **the k^{th} cycle** or **the cycle at clocktime k** to mean ω^k. We'll also use sequential language like **next cycle** and **previous cycle** based on this indexing, which we define as follows:

$$\begin{aligned} \text{Next-Cycle} : \mathcal{D}(\omega) &\longrightarrow \mathcal{D}(\omega) \\ \text{Next-Cycle} : \quad c &\longmapsto \omega c \end{aligned}$$

$$\begin{aligned} \text{Prev-Cycle} : \mathcal{D}(\omega) &\longrightarrow \mathcal{D}(\omega) \\ \text{Prev-Cycle} : \quad c &\longmapsto \omega^{-1} c \end{aligned}$$

We emphasize the multiplicative nature of this indexing: we can express the relationship between the jth cycle, c_j and the $(j+1)^{\text{th}}$ cycle, c_{j+1} by writing $c_{j+1} = \omega c_j$.

A.2 Blocks: Indexing, Metrics, and Operations

Informally, a *block* is a sequence of field elements, indexed by powers of some $\omega \in \mathbb{F}_p$, where the length of the sequence is equal to the multiplicative order of ω. In this section, we articulate the mathematical structure of blocks, including a sequential indexing, two metrics, and a number of algebraic operations on blocks.

A.2.1 The Structure of a Block

A **block** $\mathbf{u}$ over a cyclic domain $\mathcal{D}(\omega)$ is a function[17] $\mathcal{D}(\omega) \to \mathbb{F}_q$. We define the i^{th} **entry** of $\mathbf{u}$, denoted $\mathbf{u}[i]$ as follows:

$$\mathbf{u}[i] = \mathbf{u}(\omega^i)$$

We also define a block over a coset of a cyclic domain $\beta\mathcal{D}(\omega)$, where $\mathbf{u}[i] = \mathbf{u}(\beta \cdot \omega^i)$. In what follows, we'll focus on blocks over cyclic domains for simplicity.

Given $\mathbf{u}[n]$, we'll use sequential language like **next entry**, $\mathbf{u}[n+1]$ and **previous entry**, $\mathbf{u}[n-1]$. The **length** of $\mathbf{u}$ is the number of entries of $\mathbf{u}$, which is equal to the multiplicative order of ω.

Example
The entry of $\mathbf{u}$ that appears n cycles before $\mathbf{u}(c)$ can be written as $\mathbf{u}(w^{-n}c)$.

A.2.2 Block Distance and Divergence

Let $\mathcal{D}(\omega) \subseteq \mathbb{F}_q$ be the cyclic domain with counter ω, and let $\mathbf{u}, \mathbf{v}$ be blocks indexed by ω. We define the **distance** between blocks (aka **Hamming Distance**) to be the number of entries that differ between $\mathbf{u}$ and $\mathbf{v}$. Formally,

$$\delta(\mathbf{u}, \mathbf{v}) = |\{\omega^n \in \mathcal{D}(\omega) : \mathbf{u}(\omega^n) \neq \mathbf{v}(\omega^n)\}|$$

We say that $\mathbf{u}$ is **δ-close** to $\mathbf{v}$ if the distance from $\mathbf{u}$ to $\mathbf{v}$ is less than or equal to δ.

We define the **divergence** between $\mathbf{u}$ and $\mathbf{v}$ to be the proportion of entries that differ.

$$\Delta(\mathbf{u}, \mathbf{v}) = \frac{\delta(\mathbf{u}, \mathbf{v})}{|D(\omega)|}$$

We note that for any **divergence** and **distance** are both metrics over the set of blocks.

Defining Divergence of a Block and a Set

[17] Intuitively, a "block" is just a sequence of elements of $\mathbb{F}_q$ indexed by powers of ω.

Let $\mathcal{D}(\omega) \subseteq \mathbb{F}_q$ be the cyclic domain with counter ω. Let $\mathbf{u}$ be a block indexed by ω, and let V be a set of blocks indexed by ω. We define the **distance** between $\mathbf{u}$ and V to be the distance between $\mathbf{u}$ and the closest block of V.

$$\delta(\mathbf{u}, V) = \min_{\mathbf{v} \in V} \delta(\mathbf{u}, \mathbf{v})$$

We define the **divergence** of $\mathbf{u}$ and V as follows:

$$\Delta(\mathbf{u}, V) = \frac{\delta(\mathbf{u}, V)}{|\mathcal{D}(\omega)|}$$

A.2.3 Operations on Blocks

Blocks can be represented as vectors over $\mathbb{F}_q$, and we define addition and scalar multiplication in the familiar way.

Addition of Blocks
Given blocks $\mathbf{u}$ and $\mathbf{v}$, we define the block $\mathbf{u} + \mathbf{v}$:

$$\begin{aligned} (\mathbf{u} + \mathbf{v}) : \ & \mathcal{D}(\omega) \longrightarrow \mathbb{F}_q \\ (\mathbf{u} + \mathbf{v}) : \ & \quad x \longmapsto (\mathbf{u}(x) + \mathbf{v}(x)) \end{aligned}$$

Scalar Multiplication of Blocks
Given a block $\mathbf{u}$ and $\alpha \in \mathbb{F}_q$, we define the block $\alpha \cdot u$:

$$\begin{aligned} \alpha \cdot \mathbf{u} : \ & \mathcal{D}(\omega) \longrightarrow \mathbb{F}_q \\ \alpha \cdot \mathbf{u} : \ & \quad x \longmapsto \alpha \cdot \mathbf{u}(x) \end{aligned}$$

With these definitions, we see that the set of blocks indexed by ω forms a vector space over $\mathbb{F}_q$.

Mixing Blocks We define a method of mixing n blocks into one, using a mixing parameter α. This method is used to mix the constraint polynomials, to mix the DEEP polynomials, and throughout the FRI protocol.

Let $D(\omega) \subseteq \mathbb{F}_q$ be a cyclic domain. Let $\mathbf{U} = \mathbf{u}_0, ..., \mathbf{u}_{m-1}$ be a sequence of blocks, each indexed by ω, and let $\alpha \in \mathbb{F}_q$ be a **mixing parameter**. We define the **mix** of $\mathbf{U}$ by α as

$$\text{mix}(\mathbf{U}, \alpha) = \alpha^0 \mathbf{u}_0 + \alpha^1 \mathbf{u}_1 + \ldots + \alpha^{m-1} \mathbf{u}_{m-1}$$

We note that a mix of $\mathbf{U}$ is a linear combination of $\mathbf{u}_i$.

NTTs and iNTTs An **iNTT** converts an array of evaluations over a domain $\mathcal{D}$ to an array of coefficients of the minimal-degree interpolating polynomial. Given a block $\mathbf{u}$ over cyclic domain $\mathcal{D}(\omega)$, we write $f_{\mathbf{u}} = \text{iNTT}(\mathbf{u})$.

An **NTT** converts an array of coefficients to an array of evaluations over a domain $\mathcal{D}$. Given a polynomial f, we write $\mathbf{u}_f = \mathrm{NTT}(f, \mathcal{D})$.

Splitting Blocks We define a method of splitting 1 block of length $b \cdot l$ into b blocks of length l. This method is used once per FRI commit round (see Appendix C.6 and as a means of reducing the degree of the validity polynomial (see Appendix C.5.2).

Given:

1. a block $\mathbf{u}$ over $\mathcal{D}^{(0)} = \beta\mathcal{D}(\omega)$, where ω has order $b \cdot l$, and
2. a subset $\mathcal{D}^{(1)} \subset \mathcal{D}^{(0)}$ where $\frac{|\mathcal{D}^{(0)}|}{|\mathcal{D}^{(1)}|} = b$,

we construct b-$\mathbf{split}(\mathbf{u}) = \mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{b-1}$, as follows:

1. Compute $P_{\mathbf{u}} = iNTT(\mathbf{u})$
2. Sort the coefficients of $P_{\mathbf{u}}$ to construct $P_{\mathbf{v}_0}, \ldots, P_{\mathbf{v}_{b-1}}$, such that

$$P_{\mathbf{u}}(x) = \sum_{i=0}^{b-1} x^i P_{\mathbf{v}_i}(x^b)$$

   Succinctly, the coefficient of the degree j term in $P_{\mathbf{v}_i}$ is the $(bj+i)^{\text{th}}$ coefficient of $P_{\mathbf{u}}$.

3. Compute each $\mathbf{v_i} = \mathrm{NTT}(P_{\mathbf{v}_i}, \mathcal{D}^{(1)})$

A.3 Reed Solomon Encoding

In this section, we define Reed Solomon codes and outline their place in the RISC Zero proof system.

A.3.1 Intuition

A Reed-Solomon encoding is a method of translating a *message* into a *codeword*. Intuitively, you can think of the messages as columns of the execution trace; these messages are converted to codewords, which are then committed to Merkle trees.

Put succinctly, a Reed Solomon encoding $\mathcal{R}$ encodes a message $\mathbf{m}$ into a *codeword* $\mathbf{u}$ by associating $\mathbf{m}$ with a polynomial $f_{\mathbf{m}} = \mathrm{iNTT}(\mathbf{m})$, and evaluating $f_{\mathbf{m}}$ over a larger cyclic domain.[18] Formally, messages and codewords are both blocks.

Reed-Solomon encoding is used in STARK protocols as a means of error amplification, and the *rate* of the encoding refers to the amount of error amplification. For an RS encoding of *rate* $\frac{1}{4}$, distinct codewords will agree at no more than $\frac{1}{4}$ of

[18] or a coset of a cyclic domain

their entries.[19] This means that the Verifier's probabilistic queries are very likely to check a discrepancy in the trace.

For zero-knowledge purposes, RISC Zero adds random padding to the trace columns before implementing Reed-Solomon encoding and evaluates the NTT on a domain that does not intersect the domain of the message.

A.3.2 Formal Definition of Reed-Solomon Encoding

Let $\omega \in \mathbb{F}_q$ have multiplicative order bl and let $\mathcal{D}(\omega^b) \subseteq \mathcal{D}(\omega) \subseteq \mathbb{F}_q$. Let $\mathcal{B}(\omega^b)$ be the set of all blocks over $\mathcal{D}(\omega^b)$ and $\mathcal{B}(\omega)$ be the set of all blocks over $\mathcal{D}(\omega)$.

We define[20] a Reed-Solomon Encoding, $\mathcal{R}$, where

$$\mathcal{R} : \mathcal{B}(\omega^b) \to \mathcal{B}(\omega)$$

$$\mathcal{R} : u \mapsto \overline{u}$$

where the i^{th} entry of $\overline{u}$ is $P_{\mathbf{u}}(\omega^i)$ and $P_{\mathbf{u}}(x)$ is the unique polynomial of degree $l-1$ that agrees with $\mathbf{u}$ on $\mathcal{D}(\omega^b)$. We say $\mathcal{R}$ has **rate** $\frac{1}{b}$.

Given $\mathcal{R} : \mathcal{B}(\omega^b) \to \mathcal{B}(\omega)$, we say a block $\mathbf{v} \in \mathcal{B}(\omega)$ is a **codeword** if there exists $u \in \mathcal{B}(w^b)$ such that $\mathcal{R}(u) = v$. We write $\mathcal{R}(\omega)$ to refer to the set of blocks over $\mathcal{D}(\omega)$ that are codewords with respect to $\mathcal{R}$. Given $\mathcal{R}$, the **code** is the collection of all such codewords.

A.4 Block Operations on RS Codes

In this section, we present various results about block operations in relation to block validity and block proximity[21].

A.4.1 Motivation

Fundamentally, most of the protocols for the proof system described here work by operating on Reed-Solomon codes, combining and transforming them in various ways.

This transformations in turn modify the polynomials they RS codes represent. But because of the redundancy introduced by the error correcting code, once the Prover cryptographically commits to the codes before and after an operations in some way, we can verify the correctness of the operation by randomly spot checking.

Generally, we will say that if the result of the operations is a (mostly) correct

[19] The core insight of Reed-Solomon codes is that two degree n polynomials will agree at no more than n locations.

[20] For simplicity, we state the definitions here in terms of cyclic domains. We note that these definitions generalize easily to cosets of cyclic domains.

[21] Proximity in this context refers to the distance between a given block and the nearest codeword.

codeword, and the spot checks succeed with high likelihood, that input was (very likely) a (mostly) correct codeword as well, and that the operation was done correctly across the whole code, in the sense that if we corrected both codes, we would find the entire process was error free. Of course the details here matter a great deal, as do all of the precise probabilities and divergences.

We often care that the divergence of a given block from the set of RS codewords is below some critical distance δ. We say such a block is δ-close to the code.

In the next sub-sections we will introduce the the core building blocks we will use for the protocols, and provide proofs of their correctness (directly or via reference).

In all cases below, we presume δ is within the unique decode range. The key concepts are (informally):

Mixing If we randomly mix codewords, and the output is δ close, it is very likely that all of inputs were also δ close.

Splitting We can split a codeword of length bl into b smaller codewords of length l, and can evaluate the split is correct by spot checking, so long as the outputs are all δ close.

Evaluating Given a codeword, we can verify the evaluation of the underlying message polynomial at an arbitrary field element by using a quotient to compute a new codeword, and then spot checking the two codewords for the implied algebraic relationship (so long as both are δ close).

One can immediately see that by applying splitting and mixing recursively, one could check an arbitrarily large RS-code was very likely δ close in $O(\log(N))$ steps, N being the length of the code. This is in fact how the FRI protocol works.

A.4.2 Operations on RS Codewords

For proving completeness of the protocol, we need to show that if we start with RS codewords and perform algebraic block operations, the results will also be codewords. In this section, we show that the set of codewords, $\mathcal{R}(\omega)$ is closed under block addition and scalar multiplication, mixing, and splitting.

Let $\mathbf{u}, \mathbf{v}$ be codewords and let $\alpha \in \mathbb{F}_q$. Since $\mathbf{u}$ and $\mathbf{v}$ are codewords, we can associate each of them with a low-degree polynomial: $P_{\mathbf{u}}$ and $P_{\mathbf{v}}$.

Addition
Block addition is defined as pointwise addition, so $\mathbf{u}+\mathbf{v}$ can be expressed as $P_{\mathbf{u+v}}$ where $P_{\mathbf{u+v}}$ is the polynomial formed via point-wise addition of $P_{\mathbf{u}}$ and $P_{\mathbf{v}}$.

$$P_{\mathbf{u+v}}(x) = P_{\mathbf{u}}(x) + P_{\mathbf{v}}(x)$$

Given that $\mathbf{u}$ and $\mathbf{v}$ are codewords, we conclude that $\mathbf{u}+\mathbf{v}$ is also a codeword since $\deg(P_{\mathbf{u+v}}) \leq \max(\deg(P_{\mathbf{u}}), \deg(P_{\mathbf{v}}))$.

Scalar Multiplication
Similarly, $\alpha\mathbf{u}$ can be expressed as $P_{\alpha\mathbf{u}}$, where

$$P_{\alpha\mathbf{u}}(x) = \alpha P_{\mathbf{u}}(x)$$

Given that $\mathbf{u}$ is a codeword, we conclude that $\alpha\mathbf{u}$ is a codeword since $\deg(P_{\alpha\mathbf{u}}) = \deg(P_{\mathbf{u}})$.

Mix
It follows immediately that the **mix** of codewords is also a codeword, since **mix** is just a linear combination of blocks.

Split
It is similarly immediate that if $\mathbf{u}$ is a codeword on $\mathcal{D}(\omega)$, then each $\mathbf{v}_i$ in $\textbf{b-split}(u)$ is a codeword on $\mathcal{D}(\omega^b)$. This follows from the definition of **split**, since each $\mathbf{v}_i$ is constructed as the evaluation of a low-degree polynomial $P_{\mathbf{v}_i}$.

Eval
The **eval** function is used to verify the evaluation of an RS-code at an arbitrary field element (the DEEP query point, in particular). **eval** is defined as follows:

Definition 5. Let

- $\mathcal{D}$ be a domain over field $\mathbb{F}_q$
- $\mathbf{u}$ be a codeword over $\mathcal{D}$
- $e = ((x_1, y_1), ..., (x_m, y_m))$ be a sequence of m pairs $(\mathbb{F}_q \times \mathbb{F}_q)$, with all x_i unique.
- $b = \text{interpolate}(e)$; i.e. b is the minimum-degree polynomial over $\mathbb{F}_q$ such that $b(x_i) = y_i$ for $i = 1, \ldots, m$.

We define a function $\textbf{eval} : (\mathbb{F}_q^{\mathcal{D}}, (\mathbb{F}_q \times \mathbb{F}_q)^n) \to \mathbb{F}_q^{\mathcal{D}}$

$$\mathbf{u}' = \textbf{eval}(u, e)$$

where

$$\mathbf{u}'(z) = \frac{u(z) - b(z)}{\prod_{i=1}^{m}(z - x_i)}$$

The Prover's construction of the DEEPAnswerSequence involves evaluations of witness polynomials that aren't part of the associated Merkle commitments. The **eval** function allows the Verifier to validate the DEEPAnswerSequence without relying on Merkle branches. The intuition here is that if $b(z)$ is not equal to $u(z)$, then $\mathbf{u}'$ will not be a codeword.

A.4.3 Operations and Block Proximity

Throughout the protocol, the Prover uses the **mix** function in order to consolidate multiple arguments about **block proximity** to a single argument about block proximity. For proving soundness of the protocol, we need to show that if the block associated with the final FRI polynomial f_r is δ-close to a codeword, then the original trace blocks and validity blocks are also δ-close to a codeword.

The major theorem we rely on for this argument is presented in Theorem 1.5 of Eli Ben-Sasson et al, 2020. Put succinctly, the premise is that if $\mathbf{U}$ is a sequence of blocks and some **mix** of $\mathbf{U}$ gives a result that is δ-close to $\mathcal{R}(\omega)$, then it's extremely likely that each of the blocks of $\mathbf{U}$ are also δ-close to $\mathcal{R}(\omega)$. Moreover, the δ-closeness of the $\mathbf{mix}(\mathbf{U})$ allows us to conclude that the locations-of-agreement in the pre-mixed blocks are **correlated**.

This correlation is non-trivial and quite useful: typically if $\mathbf{a}$ and $\mathbf{b}$ are blocks that are each δ-close to $\mathcal{R}(\omega)$, $\mathbf{a}+\mathbf{b}$ will not be δ-close. But if $\mathbf{a}$ and $\mathbf{b}$ have correlated agreement, we can conclude that:

$$\delta(\mathbf{a}+\mathbf{b}, \mathcal{R}(\omega)) \leq \max[\delta(\mathbf{a}, \mathcal{R}(\omega)), \delta(\mathbf{b}, \mathcal{R}(\omega))]$$

Without correlated agreement, we'd have the weaker

$$\delta(\mathbf{a}+\mathbf{b}, \mathcal{R}(\omega)) \leq \delta(\mathbf{a}, \mathcal{R}(\omega)) + \delta(\mathbf{b}, \mathcal{R}(\omega))$$

Correlated Agreement of Mixed Codes
Theorem 3 allows the Verifier to conclude that the proximity of $\mathbf{u}$ to $\mathcal{V}$ implies the proximity of each $\mathbf{u}_i$ as well.

Restated informally, Theorem 3 basically says: If we randomly mix together some set of blocks, and the result is δ-close to a codeword more than a small amount ϵ of the time, then there is a large subset of $\mathcal{D}$ in which the elements are in fact all parts of a codeword. Of course, this means that in fact the original blocks we mixed together are in fact also δ close. We capture this simplified understanding below

Corollary 1 (Mixing)**.** *Let $\mathcal{V}$, $\mathbf{u}$, δ and ϵ be as defined in Theorem 3.*
If $\Pr_{\alpha \in \mathbb{F}_q}[mix(\mathbf{u}, \alpha)$ is δ-close$] > \epsilon$, then all $u \in \mathbf{u}$ are also δ-close.

Proof. By Theorem 3, there exists a set $\mathcal{D}'$ such that for each $\mathbf{u}_i$, $\mathbf{u}_i(x) = \mathbf{v}_i(x)$ for all x in $\mathcal{D}'$, and $\mathbf{v}_i \in V$. Since $\mathbf{u}_i$ and $\mathbf{v}_i$ agree on all points of $\mathcal{D}'$, and $|\mathcal{D}'|/|\mathcal{D}| \geq 1 - \delta$, we have $\Delta(\mathbf{u}_i, \mathbf{v}_i) \leq \delta$, and thus $\Delta(\mathbf{u}_i, \mathcal{V}) \leq \delta$. □

Split and δ-closeness
Haven't thought about this yet.

Evaluate and δ-closeness
The following theorem shows that spot-checking the output of **eval** is sufficient to ensure a unique decoding of the input.

Theorem 1. *Let*

- *$V = \mathsf{RS}[\mathbb{F}_q, \mathcal{D}, k]$, be an RS code with rate ρ and block size n.*

- $\mathbf{u}$ *and* $\mathbf{u}'$ *be blocks in* $\mathbb{F}_q^{\mathcal{D}}$ *which are* δ*-close to* V, $\delta < \frac{1-\rho}{2}$
- $e = ((x_1, y_1), ..., (x_m, y_m))$ *be a sequence of* m *pairs* $(\mathbb{F}_q \times \mathbb{F}_q)$*, with all* x_i *unique.*
- $b = interpolate(e)$

If

$$\Pr_{z \in \mathcal{D}}[u(z) = \mathbf{u}'(z) \prod_{i=1}^{m}(z - x_i) + b(z)] \geq \frac{k+m}{n} \tag{1}$$

Then

1. *The polynomial* u *can be corrected uniquely to some* $v \in V$
2. *For all* $0 < i < n$*,* $y_i = v(x_i)$

Proof. Claim 1 follows directly from the requirements on $\mathbf{u}$ and Theorem 2. Given the requirement of $\mathbf{u}'$, it also has a unique decoding via Theorem 2, which we call $\mathbf{v}'$.

If $\mathbf{v} = \mathbf{v}' \prod(z - x_i) + b$, then it is clear that $\mathbf{v}$ meets the requirements of claim 2, since $\mathbf{v}' \prod(z - x_i)$ will be 0 at all x_i, and thus adding $\mathbf{b}$ will thus satisfy claim 2.

On the other hand, if $\mathbf{v} \neq \mathbf{v}' \prod(z - x_i) + b$, then the difference $d = \mathbf{v} - (\mathbf{v}' \prod(z - x_i) + b)$ is a non-zero polynomial of degree at most $k + n - 1$. Thus it can have at most $k + m - 1$ zeros. But this would contradict equation 1, since it implies at least $k + m$ zeros over the n element of $\mathcal{D}$. □

A.5 Constraints and Taps

A.5.1 Constraints

Checking that an execution trace satisfies a particular RISC-V rule involves checking some arithmetic relationship involving multiple blocks across multiple clock cycles.

Given a particular rule of RISC-V that we want to check at clock cycle c, we can plug the associated taps into some equation that will return zero if the trace is valid.[22].

In other words, we can write a rule-checking function, $\mathbf{r}$, whose inputs are taps of the trace, such that when $\mathbf{T}$ is a valid trace over cyclic domain $\mathcal{D}(\omega)$, $\mathbf{r}(t_1(c), ...t_k(c)) = 0$ for all $c \in \mathcal{D}(\omega)$.

This gives a construction of a single-input constraint function C:

$$C : \mathcal{D}(\omega) \to \mathbb{F}_q$$
$$C : c \mapsto \mathbf{r}(t_1(c), ..., t_k(c))$$

By writing the taps in terms of trace polynomials, the Prover can express C as a polynomial. Putting this all together, we construct **constraint polynomials**[23], C,

[22] The control blocks allow for a construction of rule-checking functions that are time-symmetric: by using control blocks to turn on and off the enforcement of the rules, we construct rule-checking functions that evaluate to 0 at each cycle associated with a valid trace.

[23] The key feature of the Constraint Polynomials is that they have roots at each $c \in \mathcal{D}(\omega^4)$.

as follows:

$$C(c) = \mathbf{r}(\overline{t_1}, ..., \overline{t_n})$$

and each $\overline{t_i}$ is a polynomial tap of the trace. In practice, we have one constraint polynomial, $\mathbf{c}_i$, per rule-checking function, $\mathbf{r}_i$.

A.5.2 Trace Taps and Polynomial Taps

Trace taps offer a way to express references across multiple blocks and across multiple cycles. Concretely, the **tap** (i, j) at c is the entry in $\mathbf{u}_i$ that appears j clock cycles after c. Concretely, the tap (1,2) at ω^3 is $\mathbf{u}_1(\omega^{(3+2)})$.

Polynomial taps $\overline{t_i} = (a_i, b_i)$ are a natural extension taps. Whereas a tap points to an entry in a trace, a **polynomial tap** points to an evaluation of a trace polynomial. Formally, the polynomial tap $\overline{t_i} = (a_i, b_i)$ is defined as follows:

$$\overline{t_i}(c) = P_{a_i}(w^{b_i}c)$$

where P_{a_i} is the trace polynomial specified by the i^{th} tap.

Formal Definitions

Given a trace $\mathbf{T} = \mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_n$ over cyclic domain $\mathcal{D}(\omega)$, we define a **tap** $t = (i, j)$ as a function

$$t : \mathcal{D}(\omega) \to \mathbb{F}_q$$
$$t : c \mapsto \mathbf{u}_i(\omega^j c)$$

Given a collection of Trace Polynomials $P_1, P_2, \ldots, P_n$, we define a **polynomial tap** $\bar{t} = (i, j)$ as a function

$$\bar{t} : \mathbb{F}_q \to \mathbb{F}_q$$
$$\bar{t} : c \mapsto P_i(\omega^j c)$$

B Merkle Tree Structure

In this section, we describe the structure of the Merkle commitments and Merkle proofs.

Merkle Leaves and Trees

Throughout the protocol, each Merkle leaf is a vector in $\mathbb{K}$, constructed by evaluating a sequence of polynomials at a single point in D.

These polynomials are organized into **polyGroups**, where one Merkle tree is constructed per polyGroup.

The Prover generates a Merkle root associated with each of the following polyGroups:

1. Control polyGroup
2. Data polyGroup
3. `accum` polyGroup
4. Validity polyGroup
5. FRI polyGroup (1 per round of FRI)

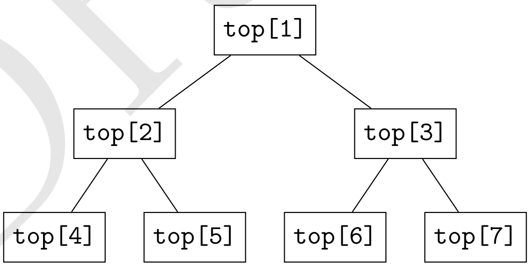
Merkle Branch Structure

Given that we're doing lots of queries from the same Merkle Tree, these queries are expected to have quite a bit of overlap close to the root. To minimize computation, rather than checking all the way to the root, we choose a cut-off point called `top_layer`.

We store that entire layer, and for each query, we check our Merkle branches from `leaf` to `top_layer`. The paths from `top_layer` to the Merkle root only need to be checked once.

More concretely, we initialize a vector `top` that will hold the top of the Merkle Tree. We initialize a vector `top` of size `2 * top_size`.[24] We populate the second half of this vector with digests from `iop.read-digests`. Then, for `i` in `(1..top_size)`, in reverse order, we compute
`top[i] = hash(top[2*i], top[2*i +1])`.

We end with the top of the Merkle tree stored in `top`, with the `top[0]` untouched and the Merkle Root in `top[1]`.



Now, to check the validity of a branch that passes through `top[4]`, this optimization allows us to check the root matches `top[1]` and that the branch from `leaf` to `top[4]` is valid.

[24] You can think of `top-size` as the size of `top-layer`.

C Protocol Details

A concise description of the IOP protocol is presented in Section 3.3. This section presents a more detailed explanation of each step. In this section, we use the notation M_{label} as shorthand for $\mathsf{Com}(\mathbb{w}_{\mathsf{label}})$.

C.1 Constructing $\mathbb{w}_{\mathsf{control}}$ and $\mathbb{w}_{\mathsf{data}}$

As in [Ben+18], the trace columns are encoded into trace blocks using Reed-Solomon encoding [RS60]. $\mathbb{w}_{\mathsf{control}}$ and $\mathbb{w}_{\mathsf{data}}$ correspond to the control blocks and data blocks, respectively, as explained in Section 2.1.

The trace columns ($\mathbf{m}_i$) are encoded as **trace blocks**[25] ($\mathbf{u}_i$) as follows:

1. Each (padded) column, $\mathbf{m}_i$ is treated as a Reed-Solomon[RS60] "message." Each message is used to construct a low-degree polynomial, $P_{\mathbf{u_i}}(x)$ using an iNTT (over $\mathcal{D}(\omega^4)$, as defined in Appendix A.1).
2. The resulting polynomial $P_{\mathbf{u_i}}(x)$ is then evaluated over $\beta\mathcal{D}(\omega)$ using an NTT, where $\beta\mathcal{D}(\omega)$ is a multiplicative coset[26] of $\mathcal{D}(\omega)$:

   $$\beta\mathcal{D}(\omega) = \{\beta x \in \mathbb{F}_q : x \in \mathcal{D}(\omega)\}$$

   The resulting array of evaluations is called a **block**. Succinctly, we define the i^{th} trace block as follows: $\mathbf{u}_i = P_{\mathbf{u}_i}\big|_{\beta\mathcal{D}(\omega)}$.

We'll sometimes write this succinctly as $\mathbf{u}_i = \text{NTT}(\text{iNTT}(\mathbf{m}_i))$. Note that $\mathbf{u}_i \neq \mathbf{m}_i$ since the domain of the NTT and the iNTT differ.

C.2 Constructing $\mathsf{Com}(\mathbb{w}_{\mathsf{control}})$ and $\mathsf{Com}(\mathbb{w}_{\mathsf{data}})$

The Prover constructs two Merkle Trees, a **Control Tree** and a **Data Tree**, using the trace blocks (i.e., the evaluations of the Trace Polynomials on $\beta D(\omega)$) as the leaves. Each leaf's address corresponds to a point of the coset $\beta\mathcal{D}(\omega)$, and the contents of the leaf at address z consist of the trace polynomials evaluated at z. As articulated in Appendix B, the Prover computes the Merkle commitment for each tree, $\mathsf{Com}(\mathbb{w}_{\mathsf{control}}) = M_{\mathsf{control}}$ and $\mathsf{Com}(\mathbb{w}_{\mathsf{data}}) = M_{\mathsf{data}}$, and writes them to the seal.

C.3 Randomness for Accumulators

In the interactive version of the protocol, the Verifier sends some randomly chosen Accumulation Parameters, after receiving the commitments M_{control} and M_D. In the non-interactive version, we generate the Accumulation Parameters by running a SHA-2 CRNG on $M_{\mathsf{control}}||M_{\mathsf{data}}$.[27]

[25] We define blocks in Appendix A.2.1 and Reed-Solomon encoding in Appendix A.3.

[26] Note that while most discussions of NTTs and iNTTs occur over a multiplicative subgroup of a finite field, RISC Zero's NTT and iNTT implementation works over a coset of a multiplicative subgroup as well. We write NTTs and iNTTs as two-argument functions that receive (1) an array over an (implied) finite field and (2) a cyclic domain or a coset of a cyclic domain. See Appendix A.2.3 for more details.

[27] We use || to denote concatenation.

C.4 Constructing $\mathsf{Com}(\mathbf{w}_{\mathsf{accum}})$

As in [GPR21] and [Azt20], we instantiate our system as a randomized AIR with preprocessing (RAP). The random accumulation parameter defined in Appendix C.3 is used (along with $\mathbf{w}_{\mathsf{data}}$) to generate accumulator polynomials. The evaluations of these accumulator polynomials are called $\mathbf{w}_{\mathsf{accum}}$; the resulting Merkle cap, M_{accum}, is commited to the seal. The seal now reads

$$M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}$$

C.5 DEEP-ALI: Constructing $\mathsf{Com}(\mathbf{w}_{\mathsf{validity}})$ and the $\mathsf{DEEPAnswerSequence}$

The DEEP-ALI portion of the protocol includes two elements of randomness and two Prover commitments.

C.5.1 Randomness for Algebraic Linking

In the interactive version of the protocol, the Verifier sends a randomly chosen Mixing Parameter, $\alpha_{\mathsf{constraints}}$, after receiving the Merkle root M_P. In the non-interactive version, we generate the mixing parameter by running a SHA-2 CRNG on $M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}$.

C.5.2 Constructing $\mathsf{Com}(\mathbf{w}_{\mathsf{validity}})$

The Prover first generates a number of **Constraint Blocks**, $\mathbf{c}_i$ using time-symmetric rule-checking functions and the Trace Blocks[28]. Then, these $\mathbf{c}_i$ are mixed together using the Constraint Mixing Parameter $\alpha_{\mathsf{constraints}}$ to form the **Mixed Constraint Block**, $\mathbf{C}$:

$$\mathbf{C} = \mathbf{mix}_{\alpha_{\mathsf{constraints}}}(C_1, ...C_v) = \alpha_{\mathsf{constraints}} C_1 + ... + \alpha^n_{\mathsf{constraints}} C_v$$

We note that $\mathbf{C}$ is a uni-variate polynomial of degree at most $5|\mathcal{D}(\omega)|$, and that $\mathbf{C}(c) = 0$ for each $c \in \mathcal{D}(\omega)$. The Prover divides $\mathbf{C}$ by a publicly known **Zeros Block**[29] to form the **High Degree Validity Block**. To wrap up the construction, the Prover splits the High Degree Validity Block into 4 **Low Degree Validity Blocks**.[30], then commits those blocks to a new Merkle Tree (the Validity Tree) and appends the root of the tree, M_{validity}, to the seal. At this stage, the seal reads as follows:

$$M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}||M_{\mathsf{validity}}$$

[28] We use "trace block" to refer to NTT($P_{\mathbf{u}_i}, \mathcal{D}(\omega)$). Note that the i^{th} trace block has 4x the length of trace column $\mathbf{u}_i$

[29] The Zeros Block consists of evaluations of the Zeros Polynomial over $\beta\mathcal{D}(\omega)$.

[30] We abuse terminology here in referring to the "degree" of a block. We write "low degree block" to mean that the polynomial associated with the block has degree less than or equal to the degree of the trace polynomials, n (which is equal to the length of a column of the trace). Enforcing RISC-V rules uses relations up to degree 5, which means the degree of the mixed constraint block is $5n$. After dividing by the zeros block, the High Degree Validity Block has degree $4n$, and the Low Degree Validity Polynomials have degree n.

C.5.3 Randomness for DEEP

In the interactive version of the protocol, the Verifier sends a randomly chosen test point, $z \in \mathbb{F}_q$, after receiving the Merkle root M_v. In the non-interactive version, we generate z by running a SHA-2 CRNG on $M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}||M_{\mathsf{validity}}$.

C.5.4 Intuition for DEEP technique

The Prover will send the DEEPAnswerSequence, which allows the Verifier to enforce to check the $\mathbf{C}(z) = Z(z)V(z)$ at the DEEP Test point, z.

Since z is not in the Merkle Tree commitments for the Trace Polynomials or the Validity Polynomials, the Prover sends additional information in order enforce that the values provided agree with the earlier Merkle commitments. This information is called the DEEPAnswerSequence.

The core of the DEEP technique is the insight that if $f(z) = a$, then $x - z$ divides $f(x) - a$ as polynomials. Moreover, if $f(z_1) = a_1$ and $f(z_2) = a_2$, then the polynomial $(x - z_1)(x - z_2)$ divides $f(x) - \overline{f}(x)$, where $\overline{f}$ is the interpolation of (z_1, a_1) and (z_2, a_2).

C.5.5 Constructing the DEEPAnswerSequence

The DEEPAnswerSequence consists of the tapset, the evaluation of V at z, and the column-by-column interpolations of the tapset.

We define a DEEP polynomial for each Trace Polynomial P_i and each low-degree validity polynomial $\mathbf{v}_j$. Given trace polynomial P_i, we define the DEEP polynomial d_i as:

$$d_i(x) = \frac{P_i(x) - \overline{P_i}(x)}{(x - x_1)\cdots(x - x_n)}$$

where $(x_1, P_i(x_1)), \ldots, (x_n, P_i(x_n))$ are the taps[31] of P_i at z and $\overline{P_i}(x)$ is the polynomial interpolation of those taps.[32]

We can define d_i more succinctly using the **eval** function defined in Appendix A.2.3:

$$d_i = \mathbf{eval}(P_i, \mathsf{taps})$$

DEEP polynomials for v'_j are defined analogously and denoted by $d_{\mathsf{w_{RAP}}+j}$ where $\mathsf{w_{RAP}}$ is the total number of trace polynomials. Since there are always 4 validity polynomials in our use cases, we end up with 4 DEEP validity polynomials $d_{\mathsf{w_{RAP}}}, \ldots, d_{\mathsf{w_{RAP}}+3}$.

The Prover interpolates each $\overline{P_i}$ and sends the coefficients to the Verifier, unencrypted. The seal now reads:

$$M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}||M_{\mathsf{validity}}||\mathsf{DEEPAnswerSequence}$$

[31] For details on taps, see Appendix A.5.2.

[32] In the Rust source, the coefficients of $\overline{P_i}$ are referred to as "u coefficients."

The Verifier can now check[33] that $V(z)C(z) = Z(z)$.

C.6 The Batched FRI Protocol

RISC Zero uses the batched FRI protocol as described in Section 8.2 of the Proximity Gaps for Reed-Solomon Codes paper.

First, the DEEP blocks are batched and then re-indexed to form a single block $f^{(0)}$. At this stage, the Prover has reduced the assertion of computational integrity to an assertion that $\mathsf{deg}(f^{(0))}$ is less than some n.

Then, over r recursive steps, the Prover reduces the assertion that $\deg(f^{(0)}) < n$ to an assertion that $\deg(f^{(r)}) < 256$.

Batching

The Verifier uses an HMAC to choose a FRI batching parameter, α_{FRI}. The Prover uses α_{FRI} to mix $d_0, \dots, d_{\mathsf{w}_{\mathsf{RAP}}+3}$. The result of this mixing is denoted $d(x)$:

$$d(x) = \mathbf{mix}_{\alpha_{\mathsf{FRI}}}(d_0, \dots, d_{\mathsf{w}_{\mathsf{RAP}}+3}) = \sum_{i=0}^{\mathsf{w}_{\mathsf{RAP}}+3} \alpha_{\mathsf{FRI}}^i d_i(x)$$

Re-indexing

Rather than evaluating d over the coset $\beta\mathcal{D}(\omega)$, the Prover first defines $f^{(0)}$ as a re-indexing[34] of d:

$$f^{(0)}(x) = d(\beta^{-1}x)$$

Now, the Prover evaluates $f^{(0)}$ over $\mathcal{D}(\omega)$ (which yields the same array as evaluating d over $\beta\mathcal{D}(\omega)$). The Prover commits the evaluations to a Merkle tree and adds the associated Merkle root $M_{f^{(0)}}$ to the seal.

The seal now reads:

$$M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}||M_{\mathsf{validity}}||\mathsf{DEEPAnswerSequence}||M_{f^{(0)}}$$

FRI Commit Rounds

Over the course of r rounds of FRI, the Prover constructs $f^{(1)}, \dots, f^{(r)}$, committing a Merkle root $M_{f^{(i)}}$ for each.

RISC Zero uses a *split factor* of 16 throughout FRI, but we write the split factor as l for generality.[35]. Each Merkle tree $M_{f^{(i)}}$ is constructed by evaluating $f^{(i)}$

[33] The Verifier computes the LHS by evaluating the DEEP validity polynomials at z^4 and using those values to compute $V(z)$. The Verifier computes the RHS by evaluating $d_i(z)$ for each trace polynomial and uses those values to compute $C(z)$ and then $V(z)$.

[34] This re-indexing serves to simplify our application of the FRI protocol from "coset FRI" to "subgroup FRI."

[35] Ben-Sasson, et. al., write the split factor as $l^{(i)}$ to allow for the possibility of varying the split factor throughout the protocol.

over $\mathcal{D}^{(i)} = \mathcal{D}(\omega^{l^i})$.[36]

For $i = 1, \ldots, r$, each $f^{(i)}$ is constructed by computing $\mathbf{mix}_{\alpha^{(i)}}(\mathbf{split}(f^{(i-1)}))$. The functions **mix** and **split** are defined in Appendix A.2, and the mixing parameters $\alpha^{(i)}$ are verifier-supplied[37] random parameters.

The Prover constructs a Merkle tree and commits a root for each $f^{(1)}, \ldots, f^{(r-1)}$. For $f^{(0)}$, the Prover interpolates the result and sends the coefficients. The seal now reads:

$M_{\mathsf{control}}||M_{\mathsf{data}}||M_{\mathsf{accum}}||M_{\mathsf{validity}}||\text{Coefficients-for-DEEP}||M_{f^{(0)}}||M_{f^{(1)}}||\ldots||\text{Coefficients-for-}f^{(r)}$

FRI Queries

After the FRI commit rounds, the Verifier makes a number of **queries** to check the integrity of the FRI commitments.

For a single query, the Verifier specifies an element $g^{(0)} \in \mathcal{D}^{(0)}$, which then induces:

1. a choice of $g^{(i)} \in \mathcal{D}^{(i)}$ for each $i = 1, \ldots, r$. Specifically, $g^{(i)} = (g^{(i-1)})^l$.
2. a coset $C^{(i)}$ for each $i = 0, \ldots, r-1$. Specifically, $C^{(i)} \subset \mathcal{D}^{(i)}$ contains the l^{th} roots of $g^{i+1} \in \mathcal{D}^{(i+1)}$.

The Prover returns $f^{(i)}|_{C^{(i)}}$ for each $i = 0, \ldots, r-1$ and $f^{(r)}(g^{(r)})$.

The verify checks:

1. The FRI batching commitment matches against the DEEP polynomials at $g^{(0)}$.
2. The **split** and **mix** operations match from round-to-round. This operation can be checked locally, using only the values revealed for the query. In particular, the value of $f^{(i+1)}(g^{(i+1)})$ can be computed using the values $f^{(i)}|_{C^{(i)}}$.

D Key Theorems for Soundness Analysis

We state the key theorems for soundness analysis here.

Theorem 2 (Reed-Solomon unique decoding)**.** *Given an RS code,* $\mathsf{RS}[\mathbb{F}_q, \mathcal{D}, k]$ *of block length n, rate ρ, and a block* $\mathbf{u} : \mathcal{D} \to \mathbb{F}_q$ *if*

$$\Delta(\mathbf{u}, V) < (1-\rho)/2$$

then there exists a unique closest $\mathbf{u}' \in V$, $\Delta(\mathbf{u}, \mathbf{u}') = \Delta(\mathbf{u}, V)$, *and such* $\mathbf{u}'$ *can be found in polynomial time.*

[36] Note that $\mathcal{D}^{(r)} \subset \ldots \subset \mathcal{D}^{(0)}$ and that $\frac{|\mathcal{D}^{(i)}|}{|\mathcal{D}^{(i+1)}|} = l$.

[37] Generated using the Fiat-Shamir Heuristic

Correlated Agreement

Theorem 3 (Correlated agreement over parameterized curves)**.**

> Let $\omega \in \mathbb{F}_q$ have multiplicative order bl, and let $\mathcal{R} : \mathcal{D}(\omega^b) \to \mathcal{D}(\omega)$ be a Reed-Solomon encoding. Let $\mathcal{V}$ be the collection of codewords of $\mathcal{R}$.
> Let $\mathbf{U} = \mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{m-1}$ be a sequence of blocks over $\mathcal{D}(\omega^b)$.
> Let δ be a distance less than unique decoding radius: $\delta \in [0, \frac{1-\rho}{2})$
> Let ϵ be a probability defined as $\epsilon = \frac{mbl}{q}$
>
> If $\Pr_{\alpha \in \mathbb{F}_q}[\Delta(\text{mix}(\mathbf{U}, \alpha), \mathcal{V}) \leq \delta] > \epsilon$, then there exists a $\mathcal{D}' \subset \mathcal{D}(\omega)$ and $\mathbf{v}_0, \mathbf{v}_1, ..., \mathbf{v}_n \in V$ satisfying
>
> **Density:** $|\mathcal{D}'|/|\mathcal{D}(\omega)| \geq 1 - \delta$
>
> **Agreement:** For all $i \in \{0, 1, ..., l\}$, and $x \in \mathcal{D}'$
>
> $$u_i(x) = v_i(x)$$

An alternative statement of this theorem is given in [Hab22]:

Theorem 4. *(Correlated Agreement Theorem) Let $\mathsf{RS}_k = \mathsf{RS}_k[F, D]$ be the Reed-Solomon code over a finite field F with defining set $D \subseteq F$ and rate $\rho = \frac{k}{|D|}$. Given a proximity parameter $\theta \in (0, 1 - \sqrt{\rho})$ and words $f_0, f_1, \ldots, f_{N-1} \in F^D$ for which*

$$\frac{|\{\lambda \in F : \delta(f_0 + \lambda \cdot f_1 + \ldots + \lambda^{N-1} \cdot f_{N-1}, \mathsf{RS}_k) \leq \theta\}|}{|F|} > \epsilon$$

where ϵ is as in (1) and (2) below. Then there exist polynomials $p_0(X), p_1(X), \ldots, p_{N-1}(X)$ belonging to RS_k, and a set $A \subseteq D$ of density $\frac{|A|}{|D|} \geq 1 - \theta$ on which $f_0, \ldots, f_{N-1}$ jointly coincide with $p_0, \ldots, p_{N-1}$, respectively. In particular,

$$\delta(f_0 + \lambda \cdot f_1 + \ldots + \lambda^{N-1} \cdot f_{N-1}, \mathsf{RS}_k) \leq \theta$$

for every $\lambda \in F$.

Quoting [Hab22]: Depending on the decoding regime, the following values for ϵ are obtained by [Ben+20]:

1. Unique decoding regime.
   For $\theta \in \left(0, \frac{1-\rho}{2}\right)$, the theorem above holds with
   $$\epsilon = (N-1) \cdot \frac{|D|}{|F|}$$

2. List decoding regime.
   For $\theta \in \left(\frac{1-\rho}{2}, 1 - \sqrt{\rho}\right)$, the theorem above holds with
   $$\epsilon = (N-1) \cdot \frac{k^2}{|F| \cdot \min\left(\frac{1}{m}, \frac{1}{10}\right)^7} \approx (N-1) \cdot m^7 \cdot \rho^{-\frac{3}{2}} \cdot \frac{|D|^2}{|F|}$$

Theorem 5. *(Batched FRI soundness error) Suppose that* $q_i \in F^D, i = 0, \ldots, L-1$, *is a batch of functions given by their domain evaluation oracles. If an adversary passes batched FRI for* $\mathsf{RS}_k[F, D]$ *and proximity parameter* $\theta = 1 - \sqrt{\rho} \cdot (1 + \frac{1}{2m}), m \geq 3$, *with a probability larger than*

$$\epsilon = (L - \frac{1}{2}) \cdot \frac{(m + \frac{1}{2})^7}{3\sqrt{\rho}^3} \cdot \frac{|\mathsf{D}|^2}{|F|} + \frac{(2m+1) \cdot (|\mathsf{D}| + 1) \cdot \sum_{i=1}^r a_i}{\sqrt{\rho} \cdot |F|} + (1 - \theta)^s$$

then the functions $q_i \in F^D, i = 0, \ldots, L-1$, *have correlated agreement with* $\mathsf{RS}_k[D, F]$ *on a set of density of at least* $\alpha > (1 + \frac{1}{2m}) \cdot \sqrt{\rho}$.

RS Codes over Subfields

One important performance enhancement used in RISC Zero's protocol involves the use of a subfield $\mathbb{F}_p$ of a larger field $\mathbb{F}_q = \mathbb{F}_{p^n}$ for some of the protocol. The idea is that many of the RS codes will use the smaller subfield to reduce the computational burden, but will interpreted later as RS codes in the full field. Of course it is immediately clear that for $\omega \in \mathbb{F}_p$, $\mathsf{RS}[\mathbb{F}_p, \mathcal{D}(w), k] \subseteq \mathsf{RS}[\mathbb{F}_{p^n}, \mathcal{D}(w), k]$. That is, all elements of the RS code of the small field are elements of the RS code of the larger field. However, we also want to prove that any block from the larger field's code (or even a block close to a codeword), for which *most* of the elements are in $\mathbb{F}_p$, is in fact close to the smaller fields code.

Formally, we have:

Theorem 6 (RS Subfield Theorem)**.** *Let*

- $V_p = \mathsf{RS}[\mathbb{F}_p, \mathcal{D}, k]$, *be an RS code with rate* ρ *and block size* n.
- $q = p^m$
- $V_q = \mathsf{RS}[\mathbb{F}_q, \mathcal{D}, k]$, *be an RS code with rate* ρ *and block size* n.
- $u \in \mathbb{F}_q^{\mathcal{D}}$ *be a block over which is* δ*-close to* V_q, $\delta < \frac{1-\rho}{2}$
- $\mathbf{u}' \in \mathbb{F}_p^{\mathcal{D}}$ *be the block which a projection of* $\mathbf{u}$ *into* $\mathbb{F}_p$, *via the rule that:* $\mathbf{u}'(x) = \mathbf{u}(x)$ *if* $\mathbf{u}(x) \in \mathbb{F}_p$, *and* $\mathbf{u}'(x) = 0$ *otherwise.*

If

$$\Pr_{x \in \mathcal{D}}[\mathbf{u}(x) \notin \mathbb{F}_p] < \delta \tag{2}$$

Then $\mathbf{u}'$ *is* δ *close to* V_p.

Proof. Given that fact the $\mathbf{u}$ is within the unique decoding radius, there is some unique decoding $v \in V_q$, such that $\Delta(v, u) < \delta$. Therefore at most δ faction of the elements are not in $\mathbf{v}$, and we also have via 2 that at most δ factions of elements are not in $\mathbb{F}_p$. Thus at least $b > 1 - 2\delta$ elements must be both in $\mathbf{v}$ and in $\mathbb{F}_p$.

Since $\delta < \frac{1-\rho}{2}$, we have $b > \rho$, or counting the number of elements rather than the fraction $B = bn > \rho n > k$. Thus there are k element in $\mathbf{v}$ which are in $\mathbb{F}_p$. But via interpolation over the field $\mathbb{F}_p$, we can find coefficients in $\mathbb{F}_p$, and therefore all all values in $\mathbf{v}$ are in $\mathbb{F}_p$, and thus $v \in V_p$, and thus $\mathbf{u}'$ is δ close to V_p. □