SP1 Technical Whitepaper

Table of contents

1 Overview

SP1 (GitHub, Documentation) is a zero-knowledge virtual machine (zkVM) developed and maintained by Succinct. SP1 proves the validity of RISC-V programs; any developer can write Rust code, specify the claimed outputs for given inputs, and then use SP1 to generate a zero-knowledge proof that the claimed outputs are correct; this proof is in the form of a Scalable, Transparent ARgument of Knowledge (STARK).

SP1 is built on top of plonky3, an open-source modular toolkit for proof systems developed and maintained by the Polygon Zero team. SP1 is made with relatively standard STARK machinery, using in particular: 1. Arithmetization via algebraic intermediate representation (AIR, [Ben+18b]), 2. Polynomial commitments via batched Fast Reed-Solomon 3. Cross-table lookups via log derivative (LogUp) lookup argument [Hab22b].

We give a brief overview of the above protocols in the Appendix. The protocol operates over the Baby Bear prime field $\mathbb{F}$ of prime order $15 \times 2^{27} + 1$, and an extension $\mathbb{F}(4)$ of degree 4.

Figure 1 illustrates the SP1 workflow from Rust program to proof.

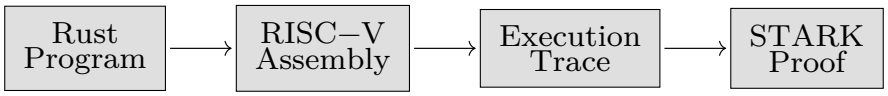

Figure 1: A Schematic of the SP1 workflow

In more detail, SP1 compiles the user's Rust code into RISC-V assembly (Section 2). RISC-V is a standard insruction set architecture (ISA), and is a regular supported target in Rust. The SP1 runtime then processes the RISC-V ELF (executable and linkable format) into an `ExecutionRecord`, which is essentially a list of the executed instructions and memory state. (Section 2). This `ExecutionRecord` is then used to generate an "execution trace", which is a single table of elements of $\mathbb{F}$, together with polynomial constraints whose enforcement guarantees proper execution; the prover's task is then to prove that the constraints are satisfied. The execution trace is a somewhat abstract version of the picture. In practice, the single table is divided into many tables (Section 3.1), each with its own constraints, and the tables coordinate with each other and with memory using LogUp (Section 3.2). Furthermore, when a single proof would be too large to store in memory, SP1 creates "shards"—pieces of full tables—and these shards are proved independently and then coordinated via LogUp (Section 3.2.3). To generate a constant-size proof, Succinct designed a recursion-specific virtual machine, together with a domain-specific language and a custom ISA (Section 4).

1.1 Conventions and Notation

1. Let $p = 15 \times 2^{27} + 1$ be the order of the BabyBear field, and let $\mathbb{F}$ denote the BabyBear field (which has order p). We let $\mathbb{F}(4)$ denote the degree 4 extension of $\mathbb{F}$. The symbol

$\mathbb{K}$ denotes a generic field.

2. We use the terms "table" and "trace" somewhat interchangeably. The term "execution trace" is reserved for the full list of instructions followed during the program's execution.
3. When referring to the LogUp protocol, we use the terms "bus", "accumulator", and "running sum" interchangeably. (We give a brief description of the LogUp protocol in the Appendix.)

2 From RISC-V Execution Trace to STARK

In this section, we describe the process by which SP1 converts a RISC-V program into a collection of tables $\mathrm{Tab}_1, \dots, \mathrm{Tab}_\ell$ whose elements take values in $\mathbb{F}$ and a set $f_1, \dots, f_k$ of polynomial constraints on the values in the table. This type of object is precisely the input into a STARK.

In a bit more formal detail: given a field $\mathbb{K}$, a number w of abstract "registers" (not necessarily corresponding to registers in a real system), a series of subsets $B_1, \dots, B_k \subset [w] \times [T]$, and a series of polynomial constraints $f_1 : \mathbb{K}^{B_1} \to \mathbb{K}, \dots, f_k : \mathbb{K}^{B_k} \to \mathbb{K}$, a STARK is a proof of the claim

Claim 1. *I know a table* $\mathrm{Tab} : [w] \times [T] \to \mathbb{K}$ *such that*

$$f_i \circ \mathrm{Tab}|_{B_i} = 0$$

for all $i \in \{1, \dots, k\}$.

By a process known as arithmetization, this claim is turned first into an equivalent claim concerning the vanishing of a **univariate** collection of polynomials (in one variable), and next into another equivalent claim about the low degree of a collection of related quotient polynomials. Once in this state, the claim can be proved by standard techniques, e.g. FRI + Merkle tree vector commitments. All of this is a standard part of the STARK workflow, and we review it briefly in the Appendix.

In the process of execution, SP1 converts a RISC-V program into the format of Claim 1. In the multi-table architecture of SP1, the result of execution is not a single claim of this form, but a number of such claims, one for each of a variety of tables. The process by which the claims are generated is essentially a two-stage process:

1. In the first step, the instructions of the RISC-V program are collected into an abstract representation of execution, concretely implemented in the struct `ExecutionRecord`. An `ExecutionRecord` contains in it all of the information about the instructions carried out in a RISC-V program, including a list of `Events`, which keep track of the facts that will need to be proved, e.g. that $14 + 2 = 16$. The `Events` are categorized into various types, e.g. `CpuEvent`, `AluEvent`, `KeccakPermuteEvent`, etc.; the `ExecutionRecord` has

separate fields to keep track of the various ALU operations (e.g. it has a field named `add_events` of type `Vec<AluEvent>`).

2. In the second step, the prover generates the tables $\{\text{Tab}_i\}$ and constraints $f_1, \ldots, f_k$. To generate the tables, referred to in the code as traces and implemented concretely as row-major matrices with values in $\mathbb{F}$, each instruction type has a corresponding `Chip` (e.g. `AddSubChip`) with a method

```
fn generate_trace(
    &self,
    input: &ExecutionRecord,
    output: &mut ExecutionRecord,
) -> RowMajorMatrix<F>
```

whose return value is a concrete instantiation of the object Tab in Claim 1. Since the registers and memory addresses in the RISC-V architecture store 32 bits, and SP1's base field has order of around 2^{31}, it is necessary to encode `u32`s as multiple elements of the BabyBear field $\mathbb{F}$. In practice, this is done by encoding a `u32` as four bytes, and embedding each byte as a separate BabyBear field element. Range checks are employed to ensure that the BabyBear elements really do come from bytes.

Finally, to impose the constraints that guarantee proper execution of the program, SP1 makes use of two types of traits: `AirBuilder` and `Air` (and similarly named objects; each `Chip` implements `Air`). The `AirBuilders` have methods for abstract constraint construction, while each implementation of `Air` (and therefore each `Chip`) has a function

```
fn eval(&self, builder: &mut AB);
```

which tells its parameter `builder` to impose the constraints on the values in the table.

Figure 2 summarizes the workflow of Execution in SP1.

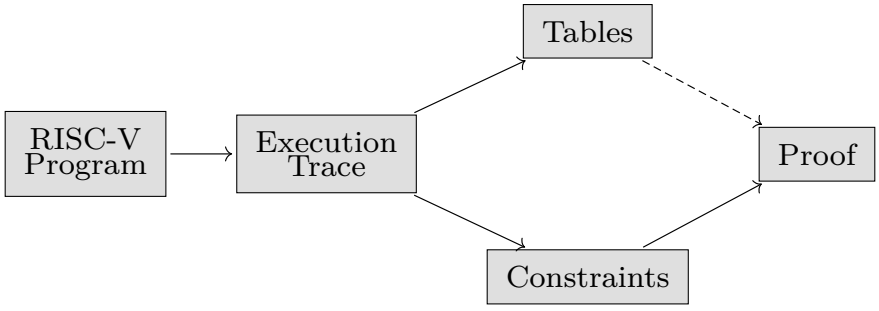

Figure 2: A Schematic of the execution process. The dashed arrow indicates that the verifier does not get full access to the tables in the course of the proof.

Let us give an example of the process by which a RISC-V instruction becomes a series of events. Suppose that at CPU clock time 7, the runtime encounters an `ADD` RISC-V instruction between registers 1 and 2 with output register 0. Let us suppose that the values in the input registers are 2 and 3, respectively. Then:

1. An `AluEvent` with opcode `ADD`, a `CpuEvent`, and some other `Events` are added to the `ExecutionRecord`.
    - The `AluEvent` has the following values for its main fields:
        - `a:5`
        - `b:2`
        - `c:3`
        - `clk:7`
    - The `CpuEvent` records the same values for `a,b,c,clk`, as well as:
        - `instruction:ADD`
        - `a_record`: a memory access record for the `a` operand (see the discussion of memory management in Section 3.2.2),
        - `b_record`
        - `c_record`
        - `pc`: the program counter value
2. When `generate_trace` is called from the `CpuChip` and `AddSubChip` `struct`s, it will add a row to each of the tables with the above information. The entries in the row are all elements of $\mathbb{F}$.
3. The `AddSubChip` and `CpuChip` call the `eval` method and use an `AirBuilder` to generate:
    - The constraints for the `AddSubChip` which enforce $5 = 2 + 3$.
    - The memory access constraints for the `CpuChip` (e.g. constraints guaranteeing that the value of register 0 at `clk` 7 is 5, and so on).
    - Additional constraints to coordinate that the values in the `Add` table match those in the `CPU` table (following the LogUp protocol).

We describe the process by which these latter two constraints are generated in Section 3.2.

3 SP1 Architecture

SP1's multi-table architecture consists of various tables that coordinate with each other to generate the proof of execution. At the center of this architecture is the CPU table, which contains a complete list of the instructions executed in the execution record. In this section, we delve into more detail about the specifications of these tables (Section 3.1) and the nature of their coordination (Section 3.2). Though the main discussion of table-to-table interactions can be found in Section 3.2, it is helpful to keep the following in mind when reading Section 3.1: table-to-table lookups and memory lookups are both managed by the same protocol (LogUp;

see [Hab22b]). Each of these lookups generates a "send" and "receive" fingerprint, and the full proof of execution consists of the table-by-table STARKs, together with a final check that the sums of the "send" and "receive" fingerprints match.

3.1 The "Beams" (Multi-Table Trace)

We describe the tables involved in generating an SP1 proof below. Two terms requiring explanation appear: "preprocessed" and "precompiled". "Preprocessed" tables are tables to which the prover commits as part of the setup. Other constraints can access/open these commitments, but there are no constraints on the tables themselves. "Precompiled" tables are tables for common cryptographic operations, e.g. Keccak; instead of unpacking these operations in the main (CPU) table, we encode these operations as `syscalls` in the CPU table and outsource them to specialized tables. The presence of these precompiled tables is a major factor behind SP1's high performance.

3.1.1 CPU Table

The CPU Table is responsible for the main CPU logic of processing RISC-V instructions. Each clock cycle in the program corresponds to a row in this table and is looked up via the `pc` column in the preprocessed `Program` table. We constrain the transition of the `pc, clk` and operands in this table according to the cycle's instruction. (For example, for a jump instruction, we verify that the `pc` value for the next instruction is as specified in the current jump instruction.) Each RISC-V instruction has three operands: `a`, `b`, and `c`, and the CPU table has a separate column for the value of each of these three operands. The `CPU` table has no constraints for the proper execution of the RISC-V instruction, nor does the table itself check that the values in the `a`, `b`, and `c` columns come from the memory addresses specified in the instruction; these are handled by lookups to the other tables and the memory argument, respectively.

3.1.2 ALU Tables

The ALU tables manage common field operations and bitwise operations: `Add/Sub`, `Bitwise`, `Divrem`, Less than (`Lt`), `Mul`, `Sll` (shift left), and `Sr` (shift right). These tables are responsible *only* for verifying correctness of an operation (with inputs `b, c` and output `a`). They "receive" a lookup from the main CPU table.

3.1.3 Memory Tables

In the main operation of its memory argument, SP1 records a "diff" for each memory access; it is not necessary to keep track of the entire state of memory in each cycle. However, it is still necessary to record the memory state at the beginning and end of the program. The Memory

Tables `MemoryInit`, `MemoryFinal`, `ProgramMemory` are responsible for exactly these operations; the preprocessed `ProgramMemory` table loads the program constants into memory.

Note that the "memory" we keep track of here includes the RISC-V registers in addition to the RISC-V memory.

3.1.4 Precompiled Tables

One of SP1's major innovations is heavy usage of separate tables for common cryptographic operations like hash functions and signature schemes that decrease proving time dramatically. Instead of paying main CPU cycles (and trace area) for these operations, SP1 instead dispatches the computations to their own tables and the `CPU` table looks up the appropriate values in this tables (precompiles are called via `ecall`s). Precompiles can directly read/write memory via our memory argument and are usually passed a `clk` value and pointers to memory addresses that dictate where the operation should read/write memory. All precompile tables can be found here and include:

- Sha256 compress
- Sha256 extend
- Keccak Permute
- Short weierstrass curve decompress (secp256k1, bn254add, bn254double, bls12-381add, bls12-381double)
- Short weierstrass curve add + double (secp256k1add, secp256k1double, bn254, bls12-381)
- Edwards add curve (ed25519 add)–note that double is the same as add for edwards curves
- Edwards decompress (ed25519 decompress)
- k256 decompress
- uint256 mul.

3.1.5 Other Tables

- The `Bytes` table is a preprocessed table to handle `u8` arithmetic operations and range checks.
- Program Table: The program table is preprocessed and constrains the program counter, instructions and selectors for the program. The `CPU` table looks up its instructions in the `Program` table.

3.2 SP1's "Joints" (Cross-Table Communication)

The multi-table architecture of SP1 is responsible for a reduction in total trace area. The cost of this division of labor, however, is that these tables now need to communicate with each other. The `ADD` table verifies that its output column is the sum of its input columns, i.e. it verifies that additions are done correctly. But it doesn't verify that these additions are the ones called for in the program. On the other hand, the `CPU` table has the opposite problem: it verifies that the additions are the ones called for in the program, but it doesn't verify that the additions were executed correctly. It is therefore necessary to guarantee that whenever there is an `ADD` instruction in the `CPU` table, the same operands appear in both tables. This is the lookup problem.

Tables communicate with each other and with memory via the a lookup protocol that uses logarithmic derivatives (LogUp, [Hab22b]), which we describe in more detail in the appendix. At a high level, whenever a table T needs to look something up in another table or in memory, T "sends" a request and the latter entity "receives" the request. This lookup happens column-wise: an entire column of one table (or, more accurately, a single affine linear combination of columns) is checked to belong to another table in some permutation. Each `send` and `receive` pair results in a matching set of fingerprints, and these fingerprints are accumulated in a cross-table bus. At the end of the proof, in addition to checking that the main constraints are satisfied, SP1 checks that the sum of all send fingerprints matches the sum of the receive fingerpints. To guarantee accurate computation of the fingerprints, it is necessary to add one permutation column for each two lookups, and a single final cumulative sum column to each table. In certain cases, it is also necessary to add a column keeping track of the multiplicity of interactions. Altogether, these columns are added using the `generate_permutation_trace` method in the `Chip` structs.

One must also add constraints to guarantee that the permutation columns were generated correctly: this is done via the `eval_permutation_constraints` method, which notably receives a `Builder` as one of its parameters. The `Builder` will impose these constraints in the AIR. The sum of the fingerprints is the last entry in the cumulative sum column. For security reasons, these columns all have values in $\mathbb{F}(4)$, so each permutation column should be counted as equivalent to four main trace columns. Refer to Figure 3 for a depiction of this structure.

3.2.1 Interactions

All table-to-table and table-to-memory communication is handled via the [Interaction](https://github.com/succinctlabs/sp1/blob/94d992d69431736c01d9896efcda4f9d2c892680/core/
`struct. AnInteractionencodes an affine-linear combination of columns in the receiver to be looked up in the sending table. It also carries a multiplicity and an enum value indicating the interaction type (memory, program, ALU, etc.). This simplest kind of affine linear combination just selects a single column of the receiving table, but more general lookups are possible. For`

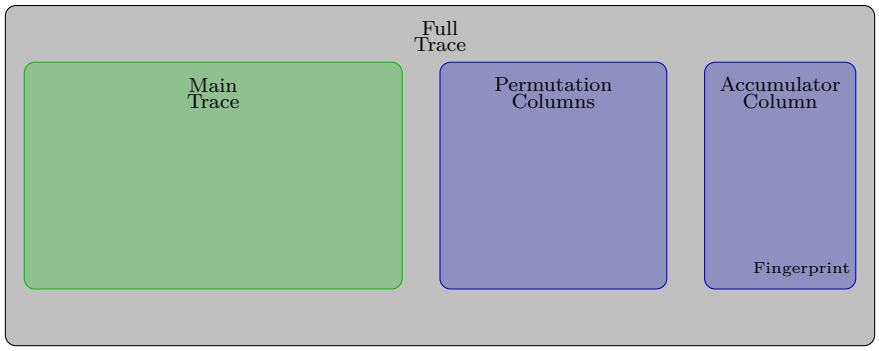

Figure 3: The extra columns needed for LogUp.

`example, the following lookup is carried out using a general affine-linear combination: the CPU table receives from the`ADD`table the values of the`a,b,c`operands only for the rows whose corresponding instruction in the CPU table is`ADD‘.

3.2.2 Memory in the Head

To ensure the consistency of memory throughout the execution of the program, SP1 slightly adapts an approach first described in [Blu+91], Section 4. Figure 4 gives a schematic depiction of SP1's memory argument.

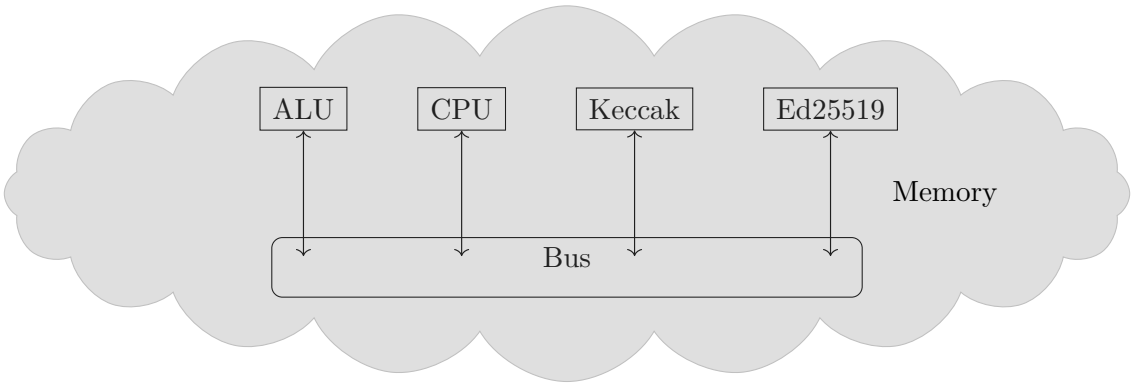

Figure 4: SP1's multi-table architecture.

This approach is based on the following idea: if we record each memory access (whether read or write) as a "read" operation followed by a "write" operation (possibly writing the same value as was read), then the memory is consistent if and only if each read operation has a matching write operation. More precisely, suppose that we maintain two sets W, R of triples (v, a, t) where v records the value of the operation, a the address, and t the time of the operation. Every time a read operation is performed at address a, add (v, a, t) to R, and (v, a, t') to W where:

- v is the value read at address a,
- t is the most recent read operation at address a
- t' is the current time.

Similarly, every time a write operation is performed at address a, add (v, a, t) to R and (v', a, t') to W, where everything is as in the previous case except that v' is the new, written value. The informal description of the memory argument at the beginning of this paragraph can now be formalized as the following slight modification of Lemma 1 of [Blu+91]:

Lemma 1. *If R and W are as described above, then the memory is consistent if and only if $R = W$ and for every pair of triples added to the sets as above, $t < t'$.*

In practice, R and W are implemented as ordered tuples $\hat{R}$ and $\hat{W}$, and to check that $R = W$, we must check that $\hat{R}$ and $\hat{W}$ are permutations of each other. This memory consistency check is simply another application of the LogUp protocol. The constraint $t < t'$ is checked via additional polynomial constraints. Because this argument only uses the LogUp bus, it does not require an explicit memory table, except to initialize and finalize the memory (in `MemoryInit` and `MemoryFinalize` tables); this is the sense in which memory in SP1 is virtual or "in the head".

3.2.3 SP1 sharding argument for large programs

For large programs (a Tendermint light client verification is around 30 million cycles), proving the full program at once will overflow the memory. Thus for large programs, we "shard" our execution trace into many smaller tables, which we call shards. We generate a proof for each smaller table and then combine the proofs together across tables to prove execution of an entire program. For example, for a program like Tendermint, assuming a "shard size" (i.e. maximum table height) of 2^{22} (roughly 2 million), there are around 20 tables. As for the cross-table lookups, cross-shard lookups are proved using the LogUp protocol. Figure 5 shows a schematic of the sharding process.

3.2.4 Combining Shards: Two-Phase Commitment

In the appendix, we describe the LogUp protocol as an interactive protocol. However, in order to obtain a non-interactive proof, we need to apply the Fiat-Shamir heuristic to generate a deterministic replacement for the verifier's random challenges. To maintain the security of the protocol, this deterministic replacement must be generated in a way that "absorbs the entropy" of *all* of the prover's previous commitments, for example by hashing those tables. Since all of the shards can interact with each other through the bus, we need to commit to the main traces first in order to generate the verifier's challenge. Only then can we compute the additional permutation columns and commit to them. These two commitments are referred

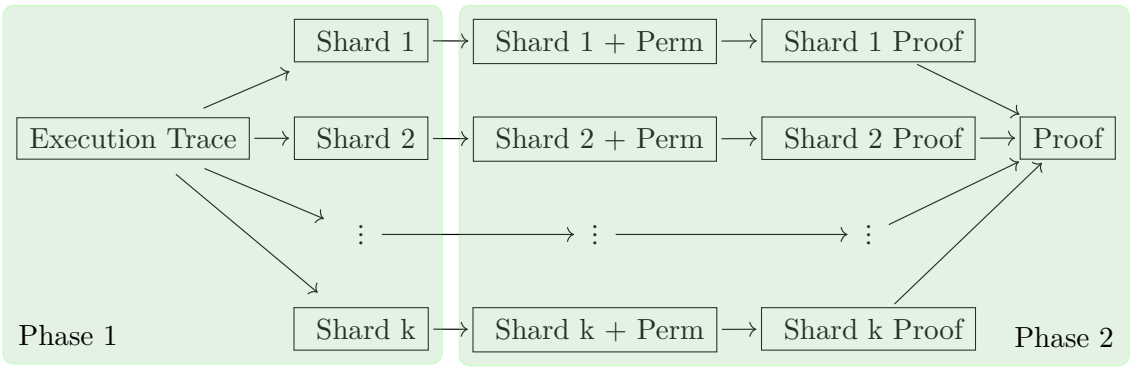

Figure 5: SP1 Sharding Process. The execution trace is divided into shards. The communication between shards is encoded by adding permutation columns to the shards. The shards are proved individually, and a final consistency check gives the full proof.

to, respectively, as the "Phase 1" and "Phase 2" commitments. In practice, we regenerate the trace and commitment from Phase 1 (this is "wasted" work) because saving the trace to disk and loading from disk is more expensive than redoing the work.

4 Recursion

The process described above generates proofs whose length is approximately proportional to the size of the execution trace being verified: the longer the program, the larger the number of shards (and therefore shard proofs) needed. When a constant-size proof is needed, SP1 can generate a recursive argument. The shards are proved individually, and their "fingerprints" are passed up into the higher nodes of the recursion tree. The proofs in the higher nodes are written in a custom language that gets compiled (using a custom compiler) into a custom assembly language. This custom architecture is optimized for recursive proof verification over the BabyBear field. The proof at the root of the tree is "wrapped" into a final, GROTH16-compatible proof that can be efficiently verified on-chain. Figure 6 shows a schematic of the process.

5 Performance

Figure 7, Figure 8, and Figure 9 give some benchmarking statistics for running the Tendermint example on SP1. The benchmarks were run on an Apple M3 Max processor, using this commit (from April 11, 2024). Depending on the block being verified, the number of shards varies in the range $[20, 25]$. The time to generate a proof is about 195 seconds (3 minutes and 15 seconds).

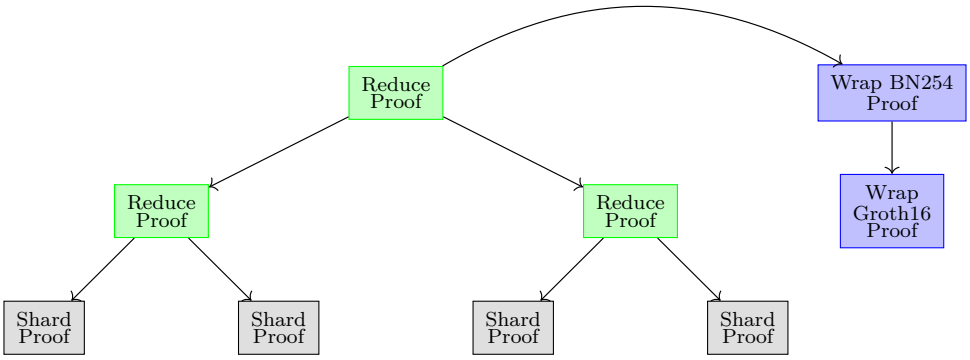

Figure 6: A schematic of recursion in SP1. The colored nodes are all written in the custom language and assembly. The different colors indicate different stages of the recursion process.

Figure 7 shows the number of columns (main and permutation) in each table. Across the chips, one finds that there is about one permutation column for every five main columns. Notice also that the precompiled tables EdAddAssign and EdDecompress account for the large majority of the total columns. This is the cost of using the precompiled tables: though they reduce the length of the CPU chip trace by "outsourcing" the computation to a separate table, the many steps of computing the function captured in the precompiled table are turned into columns therein.

Figure 8 shows the time it takes for the prover to complete various steps of the protocol for each of the chips on an Apple M3 Max processor. The times are measured in CPU time: as implemented in the code, the prover takes advantage of parallelization over the chips, so the wall clock time is less than the sum of the CPU times. Hence, the absolute times are not as important as the relative times between the phases and the tables.

We note that:

1. "Trace" refers to the process of computing the main execution trace given the RISC-V instructions.
2. "LDE" refers to the process of computing low-degree evaluations of the execution traces, i.e. to the process of encoding a polynomial as a Reed-Solomon codeword.
3. In Phase 1, the LDE is computed only for the main trace, while in Phase 2, the LDE is computed for the main trace, the permutation trace, and the quotient polynomial.
4. "Permutation Trace" refers to the process of computing the extra columns necessary for the LogUp argument.
5. "Quotient" refers to the process of computing the values of the quotient polynomial on the LDE domain.

The two steps that are repeated in Phase 1 and Phase 2—trace generation and low-degree evaluation—take essentially the same amount of time in Phase 1 as in Phase 2. In most tables,

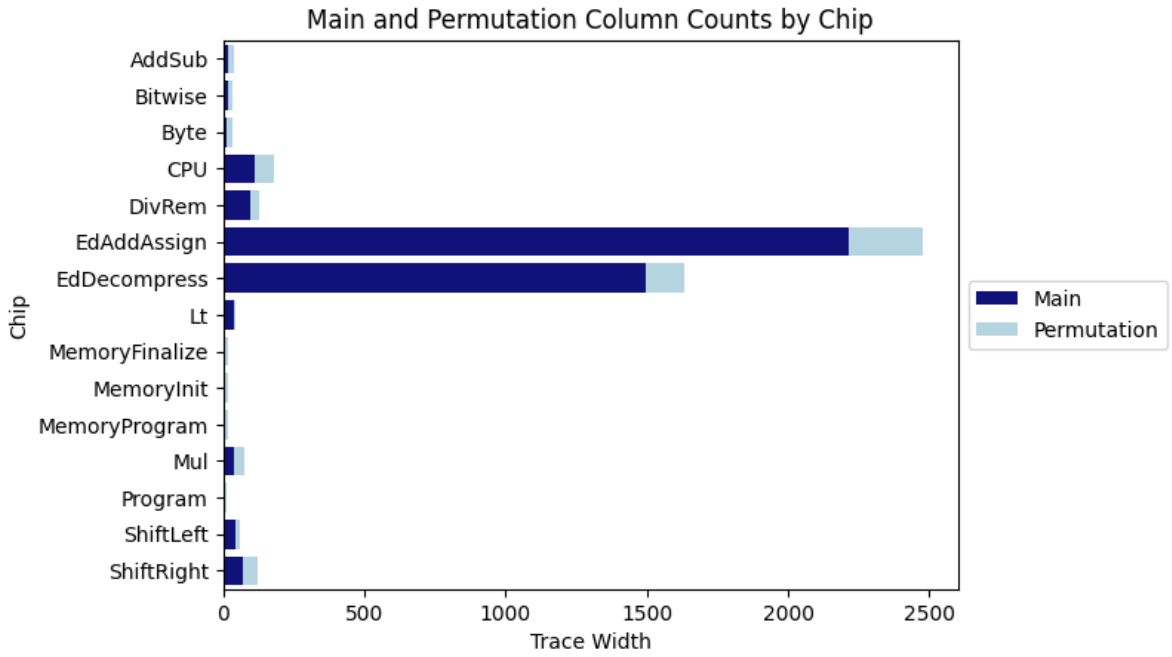

Figure 7: Column Counts (Main and Permutation) for Tendermint

the permutation trace computation and the quotient computation take the most time. Note also that the CPU table requires the most time, since it is the largest table by area.

Omitted from Figure 8 are the times involved in committing to and opening the evaluations, since all of the tables are combined into a single commitment. But Figure 9 aggregates the results of Figure 8, and also includes these opening/commitment times.

6 Appendix

The proofs created by SP1 employ several common cryptographic protocols. We give brief summaries of these protocols, giving more detail for the less standard protocols.

6.1 FRI

SP1 uses the Fast Reed-Solomon Interactive oracle proof of proximity (FRI) [Ben+18a; Ben+19] with the following parameters: the BabyBear field of order $15 \times 2^{27} + 1$, blowup factor 2 (Reed-Solomon code rate 1/2), arity-two folding (the size of the domain of evaluation is halved in each iteration), and batching.

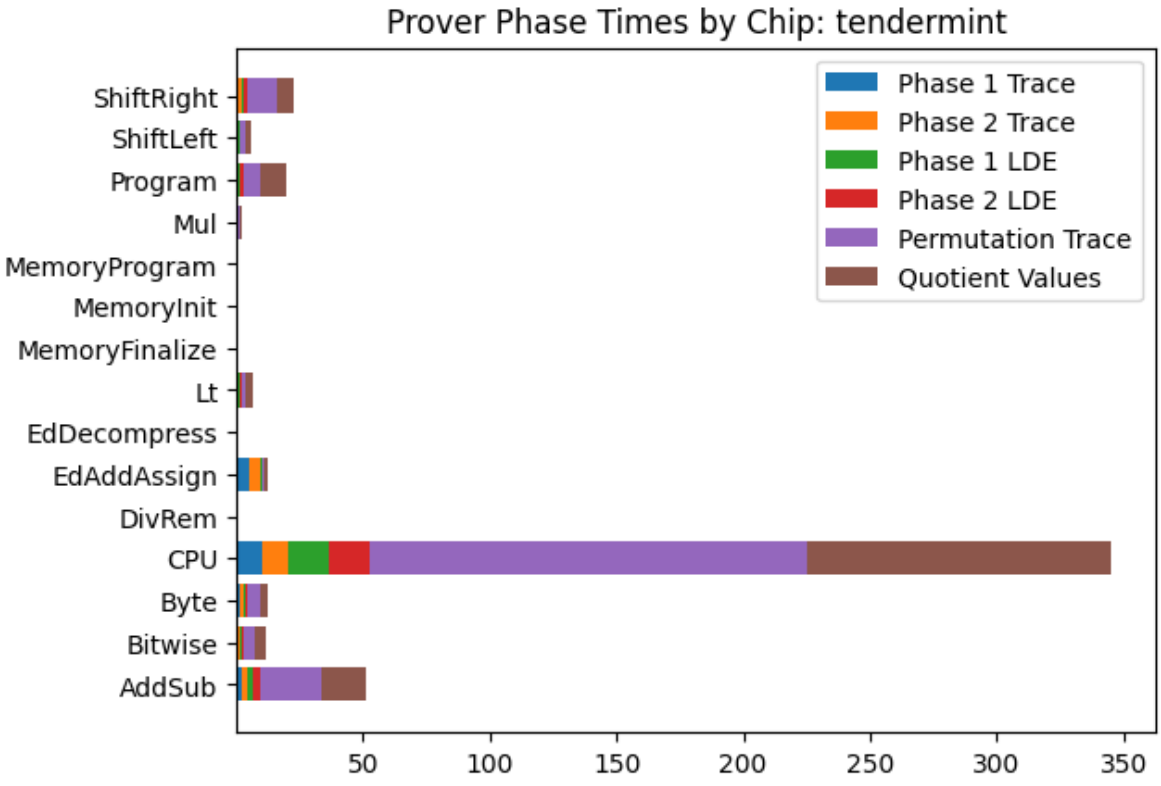

Figure 8: Proving Times by Table and Process

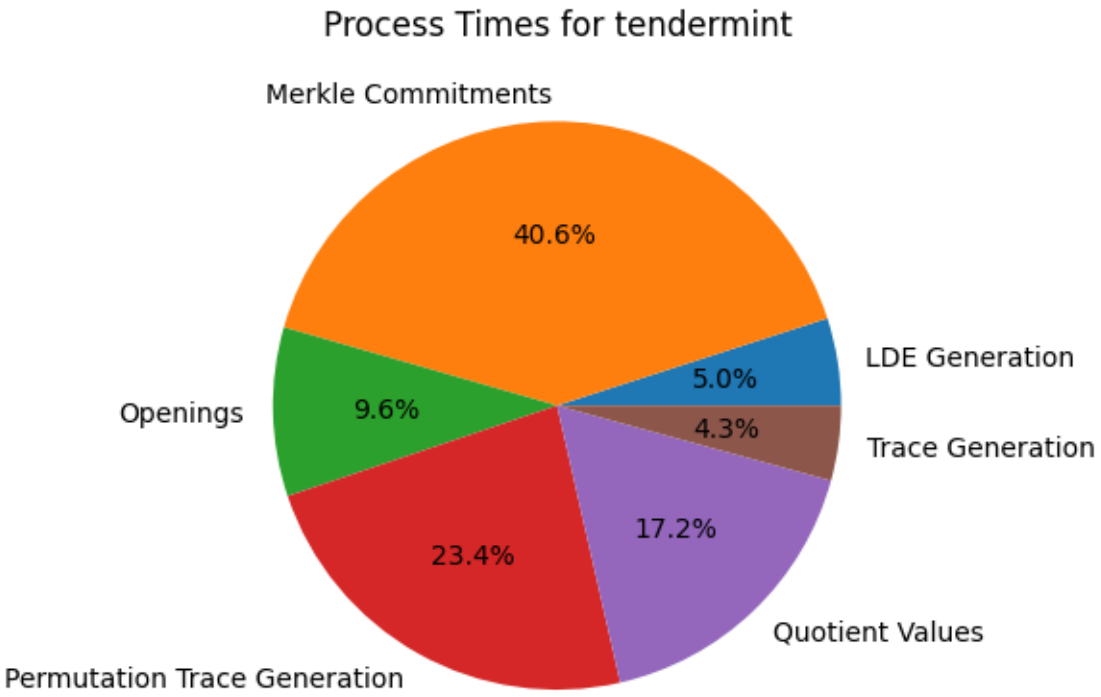

Figure 9: Proving Times by Process

Given a field $\mathbb{K}$, a subset $S \subseteq \mathbb{K}$ of size n, and a degree $d < n$, FRI is a protocol for proving that a function $f : S \to \mathbb{K}$ is the set of evaluations of a polynomial of degree less than d on S (the set of all such functions is denoted $\mathrm{RS}[\mathbb{K}, S, d]$ and is called the set of *Reed-Solomon codewords*).

The protocol proceeds by a series of reductions: given $f \in \mathrm{RS}[\mathbb{K}, S, d]$, one round of the FRI protocol produces $f' \in \mathrm{RS}[\mathbb{K}, S', d/2]$, where $|S'| = n/2 = |S|/2$. Similarly, if $f \notin \mathrm{RS}[\mathbb{K}, S, d]$, $f' \notin \mathrm{RS}[\mathbb{K}, S', d/2]$ (with high probability). Hence, after about $\log(d)$ iterations, the verifier will have a set of evaluations of a function on a set with two elements. In the honest case, this function should be constant; in the dishonest case, it is non-constant with high probability. In the former case, the verifier accepts the proof; in the latter, it rejects.

For further details of this reduction process, we refer the reader to the expository note [Hab22a].

6.2 AIR Arithmetization

SP1 arithmetizes its proofs of execution using the Algebraic Intermediate Representation (AIR) as implemented in Plonky3. This is a standard construction [Ben+18b], but let us do an example computation to schematically demonstrate how AIR works. Let $F_0 = F_1 = 1$, and define $F_{n+2} = F_{n+1} + F_n$, i.e. $F_0, \ldots, F_n, \ldots$ is the Fibonacci sequence. Suppose that we want to verify that a program correctly computed $F_5 = 8$ by sending over a sequence $x_0, \ldots, x_5$ of integers claimed to be $F_0, \ldots, F_5$. The essence of AIR is to notice that $x_0, \ldots, x_5$ are correctly computed if and only if they are solutions to the following set of polynomial equations:

$$X_0 - 1 = 0 \tag{1}$$
$$X_1 - 1 = 0 \tag{2}$$
$$X_{i+2} - X_{i+1} - X_i = 0, \quad i \in \{0, 1, 2, 3\}, \tag{3}$$

where Equations 1 and 2 are known as *boundary equations* and Equation 3 the *transition equation*. In this example, the polynomial equations are little more than a restatement of the *definition* of the Fibonacci sequence, but let us note that this formulation can encode a wide range of computations, even those that are not immediately described by polynomial equations. For example, if one expected that

$$x_{i+1} = \begin{cases} x_i^{-1}, & x_i \neq 0 \\ 0, & x_i = 0 \end{cases},$$

then one could encode this in the following polynomial equations

$$X_{i+1}(X_i X_{i+1} - 1) = 0$$
$$X_i(X_i X_{i+1} - 1) = 0.$$

In a finite field $\mathbb{K}$ with n elements, one could equivalently represent this operation using the polynomial equation $X_{i+1} - X_i^{n-2} = 0$, but this is a high degree polynomial, while the above equations are at most of degree 2 in their variables.

Equations 1, 2, and 3 are polynomials in multiple variables; in order to produce polynomials which can be passed into FRI, we need to reduce the many variables into a single one. To do this, we choose a generator $o \in \mathbb{K}$ of a multiplicative subgroup whose order equals the number of cycles (6, in the Fibonacci example) and find the unique univariate polynomial p of degree 5 such that $p(o^i) = x_i$ for all i.

Making this substitution, we simply need to check that $p(1) = 1$, $p(o) = 1$, and

$$f(x) := p(o^2 \cdot x) - p(o \cdot x) - p(x) = 0, \quad \forall x \in \{1, o, o^2, o^3\}. \tag{4}$$

In other words, we have reduced the verification of the computational claim to a statement about the values of a univariate polynomial derived from p on a given domain (namely the powers of o).

The FRI protocol is meant to verify claims about the low degree of certain polynomials, not necessarily about the values of those polynomials on a specific domain, which is what one has obtained by the process so far.

To this end, note that we can define a polynomial

$$\tilde{f}(x) = \frac{p(o^2 \cdot x) - p(o \cdot x) - p(x)}{(x-1)(x-o)(x-o^2)(x-o^3)};$$

the function $\tilde{f}(x)$ is a polynomial because the roots of the denominator are precisely the vanishing locus of the polynomial f in Equation 4. Because the polynomial p has degree 5 in its variable, the polynomial $\tilde{f}$ has degree 1. In general, for a trace with $T+1$ rows and degree m polynomial constraints in the X_i variables, $\tilde{f}$ will have degree less than $mT - T = (m-1)T$ (in our example, we've elided a detail that would allow us to divide also by $(x-o^4)(x-o^5)$ in the above equation). The FRI protocol requires us to provide a Reed-Solomon code-word, i.e. we need to evaluate $\tilde{f}$ on a domain of size 2β, where β is the blowup factor (a "hyperparameter" of the protocol, which in the SP1 implementation is 2). The factor of 2 appears because $\tilde{f}$ is of degree 1 and so is determined by its evaluations at 2 points. In the general case, we need to evaluate $\tilde{f}$ on $\beta(m-1)T$ points; it's easiest to evaluate at points where the denominator is non-zero, so we should choose a domain disjoint from the one generated by o. In practice, we choose a coset of the multiplicative subgroup generated by o^β in $\mathbb{K}$, say the coset containing $g \in \mathbb{K}$. (For the less mathematically inclined, this means we evaluate $\tilde{f}$ on $S := \{g, go^\beta, go^{2\beta}, ...\}$.) The inverse Fast Fourier transform allows for the efficient computation of the polynomial p, and the Fast Fourier Transform makes it easy to compute the evaluations of $\tilde{f}$ on the above-mentioned domain. This collection of evaluations is known as the "low-degree evaluation" of the polynomial $\tilde{f}$, and is by definition a Reed-Solomon codeword in the code $\mathrm{RS}[\mathbb{K}, S, (m-1)T]$; therfore, we can apply the FRI protocol to it.

6.3 Log Derivative Lookup Argument (LogUp)

To ensure consistency between the tables, SP1 employs a version of the log derivative lookup argument (LogUp) described in [Hab22b]. It is easiest to explain how LogUp works with an example. Let us suppose that the `CPU` looks as in Table 1:

Cycle	**Instruction**	**Output Value**	**Operand 1 Value**	**Operand 2 Value**	⋯
0	ADD	3	2	1	⋮
1	MUL	6	2	3	⋮
2	ADD	4	2	2	⋮

Table 1: CPU Table

(So, for example, the table records an `ADD` operation in Cycle 2 which adds the values 2 and 2 to get 4. The notation ⋯ indicates that the `CPU` table contains many more columns.)

Suppose also that the `ADD` table looks as in Table 2:

Cycle	**Output Value**	**Operand 1 Value**	**Operand 2 Value**
2	4	2	2
0	3	2	1

Table 2: ADD Table

A cursory glance at the tables reveals that the `ADD` table simply consists of the rows of the `CPU` table whose instruction is `ADD`, except that the rows have been rearranged. The lookup protocol is designed to verify precisely this kind of fact, and it is based on the following observation:

Lemma 2. *Suppose that $n < p$, where p is the characteristic of $\mathbb{K}$. Then, the matrices $\{A_{ij}\}_{i\in[n],j\in[m]}$ and $\{B_{ij}\}_{i\in[n],j\in[m]}$ are the same, except for a possible permutation of the rows, if and only if*

$$\sum_{i=0}^{n-1} \frac{1}{X + \sum_{j=0}^{m-1} Y^j A_{ij}} = \sum_{i=1}^{n-1} \frac{1}{X + \sum_{j=0}^{m-1} Y^j B_{ij}} \tag{5}$$

as rational functions in the variables X, Y.

Proof. For fixed $y \in \mathbb{K}$, Lemma 3 of [Hab22b] gives that Equation (5) holds if and only if there exists a permutation σ on n elements such that, for all i,

$$\sum_{j=0}^{m-1} y^j A_{ij} = \sum_{j=0}^{m-1} y^j B_{\sigma(i)j}.$$

Letting y vary now, we obtain an equality between two polynomials of degree at most m in the Y variable. Hence, we must have Equation (5) hold if and only if there exists a permutation σ such that for all i and j, the equation

$$A_{ij} = B_{\sigma(i)j}$$

holds, which is to say that A and B are the same up to a permutation of the rows. □

We remark that the constraint $n < p$ is important; if this requirement is not carefully checked in the protocol, this could lead to potential vulnerabilities. If possible, we would like to adapt the LogUp protocol to avoid this requirement.

SP1 uses a variant of the above discussion. Instead of checking that the rows of A_{ij} are permutations of the rows of B_{ij}, one instead has a collection of matrices $A_{ij}^{(1)}, A_{ij}^{(2)}, \dots, A_{ij}^{(k)}$ and wants to establish that the i-th row of of B appears with multiplicity m_i among the rows of the A matrices for all i. One modifies the LogUp protocol as follows: Equation (5) is replaced by the equation

$$\sum_{\ell=1}^{k}\sum_{i=0}^{n-1} \frac{1}{X + \sum_{j=0}^{m-1} Y^j A_{ij}^{(\ell)}} = \sum_{i=0}^{n-1} \frac{m_i}{X + \sum_{j=0}^{m-1} Y^j B_{ij}},$$

and the prover now needs to commit to the multiplicities m_j.

The preceding lemma suggests the following protocol, summarized also in Figure 10:

1. The prover commits to $A_{ij}^{(\ell)}, B_{ij}$, and m_i.
2. The verifier samples and sends random elements $\gamma, \beta \in \mathbb{K}$.
3. The prover sends $s_i^{(\ell)}$ and r_i (in the honest case, $1/(\gamma + \sum \beta^j A_{ij}^{(\ell)})$ and $1/(\gamma + \sum \beta^j B_{ij})$, respectively).
4. The verifier checks that:

$$s_i^{(\ell)}(\gamma + \sum_j \beta^j A_{ij}^{(\ell)}) = 1, \quad \forall i, \ell$$
$$r_i(\gamma + \sum_j \beta^j B_{ij}) = m_i, \quad \forall i$$

5. The prover and verifier run a protocol to prove $\sum_i r_i = \sum_{i,\ell} s_i^{(\ell)}$.

In practice, the first two equations in Step 4 are checked by adding a permutation column keeping track of $1/(\gamma + \sum_j \beta^j A_{ij})$ and the last equation is checked via a running sum column $\sum_{k=1}^{i} r_k$ to A, and similarly for B. The first two equations of Step 4 are then added as constraints in the AIR arithmetization, as is a constraint verifying proper computation of the running sum. By this process, Tables 1 and 2 are augmented to become Tables 3 and 4.

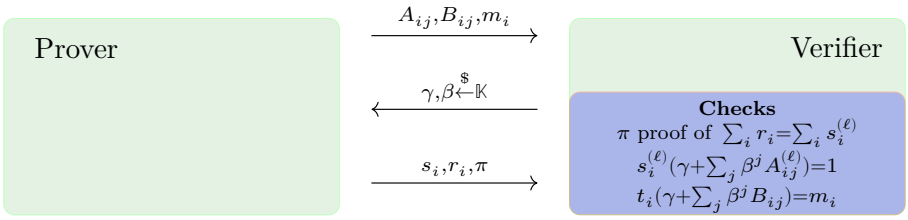

Figure 10: A schematic depiction of the LogUp protocol.

Let us demonstrate how this procedure works as applied in the given `CPU` and `ADD` tables. For concreteness, let $\gamma = 1, \beta = 2$, and suppose $\mathbb{K} = \mathbb{F}_7$. It is very important to note that the prover first commits to Tables 1 and 2; then the verifier sends γ, β. Finally, the prover commits to the permutation column and the running sum column.

Program Counter	**Instruction**	**Output Value**	**Operand 1 Value**	**Operand 2 Value**	**Permutation Column**	**Running Sum**	⋯
0	ADD	3	2	1	5	5	⋮
1	MUL	6	2	3	0	5	⋮
2	ADD	4	2	2	6	4	⋮

Table 3: Augmented CPU Table. Computations are done mod 7, with $\gamma = 1, \beta = 2$.

Program Counter	**Output Value**	**Operand 1 Value**	**Operand 2 Value**	**Permutation Column**	**Running Sum**
2	4	2	2	6	6
0	3	2	1	5	4

Table 4: Augmented ADD Table. Computations are done mod 7, with $\gamma = 1, \beta = 2$.

Note that the bottom entries in the "Running Sum" column of both tables are equal.

In practice, multiple lookups will need to be performed. For each comparison, one should add a "Permutation Trace" and "Running Sum" column to both tables, one of which will be the "sender" and one the "receiver". (An optimization, whereby one needs only add one permutation column per two lookups, is employed in SP1.)

Figure 7 shows the breakdown between the main and permutation columns by table for the Tendermint example.

One final, but important, techincal note: Equation (5) can be rearranged into a polynomial equation of degree at most $(2n-1)(m-1)$ in its variables. By the Schwartz-Zippel lemma,

the probability that it holds at a randomly chosen point is bounded by

$$\frac{(2n-1)(m-1)}{|\mathbb{K}|},$$

where n is the number of rows in the matrices to be compared, and m is the number of columns. A typical value of n is 2^{22}, and for the BabyBear field, $|\mathbb{F}|$ is about 2^{31}, which gives around 9 bits of security for the LogUp protocol (even less if m is large). This is typically too small for cryptographic purposes, so the lookup arguments in SP1 happen over the degree 4 extension of the BabyBear field, which gives an additional 96 bits of security. So, SP1's proofs are actually generated using this degree four extension $\mathbb{F}(4)$.

References

[Blu+91] Manuel Blum et al. "Checking the correctness of memories". In: *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*. SFCS '91. San Juan, Puerto Rico: IEEE Computer Society, 1991, pp. 90–99. ISBN: 0818624450. DOI: 10.1109/SFCS.1991.185352. URL: https://doi.org/10.1109/SFCS.1991.185352.

[Ben+18a] Eli Ben-Sasson et al. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity". In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Ed. by Ioannis Chatzigiannakis et al. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 14:1–14:17. ISBN: 978-3-95977-076-7. DOI: 10.4230/LIPIcs.ICALP.2018.14. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2018.14.

[Ben+18b] Eli Ben-Sasson et al. *Scalable, transparent, and post-quantum secure computational integrity*. Cryptology ePrint Archive, Paper 2018/046. https://eprint.iacr.org/2018/046. 2018. URL: https://eprint.iacr.org/2018/046.

[Ben+19] Eli Ben-Sasson et al. *DEEP-FRI: Sampling Outside the Box Improves Soundness*. Cryptology ePrint Archive, Paper 2019/336. https://eprint.iacr.org/2019/336. 2019. URL: https://eprint.iacr.org/2019/336.

[Hab22a] 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.

[Hab22b] Ulrich Haböck. *Multivariate lookups based on logarithmic derivatives*. Cryptology ePrint Archive, Paper 2022/1530. https://eprint.iacr.org/2022/1530. 2022. URL: https://eprint.iacr.org/2022/1530.