EDEN - a practical, SNARK-friendly combinator VM and ISA

June 30, 2023

Logan Allen
logan@zorp.io
Zorp

Brian Klatt
brian@zorp.io
Zorp

Philip Quirk
phil@zorp.io
Zorp

Yaseen Shaikh
yaseen@zorp.io
Zorp

Abstract

Succinct Non-interactive Arguments of Knowledge (SNARKs) enable a party to cryptographically prove a statement regarding a computation to another party that has constrained resources. Practical use of SNARKs often involves a Zero-Knowledge Virtual Machine (zkVM) that receives an input program and input data, then generates a SNARK proof of the correct execution of the input program. Most zkVMs emulate the *von Neumann architecture* and must prove relations between a program's execution and its use of Random Access Memory. However, there are conceptually simpler models of computation that are naturally modeled in a SNARK yet are still practical for use. Nock is a minimal, homoiconic combinator function, a Turing-complete instruction set that is practical for general computation, and is notable for its use in Urbit.

We introduce **Eden**, an **E**fficient **D**yck **E**ncoding of **N**ock that serves as a practical, SNARK-friendly combinator function and instruction set architecture. We describe arithmetization techniques and polynomial equations used to represent the Eden ISA in an Interactive Oracle Proof. Eden provides the ability to prove statements regarding the execution of any program that compiles down to the Eden ISA. We present the Eden zkVM, a particular instantiation of Eden as a zk-STARK.

Contents

1 Introduction

1.1 Background

In recent years, work on and interest in zero-knowledge proof systems has been explosive. These are protocols in which a **prover** entity can convince a **verifier** entity of a computational claim while keeping certain information secret. Such systems work with computations encoded using an **arithmetization** process into the mathematical language of finite fields and polynomials. Given the delicate and specialized nature of this translation process, these proof systems are largely impractical for ordinary programmers who would ideally specify computation in a familiar language and not in terms of polynomials. Thus there is a natural demand to find a way to generate proofs without deviating too far from normal programming practices.

One way of accomplishing this is to target a specific instruction set architecture (ISA) which is the compilation target of a Turing-complete programming language, and to write polynomial constraints which represent the execution of an arbitrary program for this ISA. Since modern CPUs physically implement *von Neumann* architectures, it is natural that efforts to implement this solution have resulted in ISAs that substantially mirror modern CPU architecture.

However, combinatory logic and the functional programming paradigm provide an alternative mathematically-centered perspective on computation which stretches back to the duality between Turing's machines and Church's lambda calculus. Our attachment to functional computation flows from our enthusiasm for the elegant minimalism of **Nock**. Nock is an ISA given by approximately a dozen rules for transforming binary trees, which together define a practical Turing-complete combinator function. While Nock is expressive enough to be read and understood, code is typically written in higher level languages that compile down to Nock instead of in Nock directly. **Hoon** is at present the primary high-level language that compiles to Nock; Hoon exists as part of the Urbit project.

Natural and minimal arithmetizations provide a conceptually simple substrate to reason about and have numerous advantages, including efficiency and flexibility. Our uniquely compact and purely mathematical specification has the potential to be utilized in various existing and future cryptographic schemes.

1.2 Our Work

We have started from the basis of the Nock ISA, a Turing-complete combinator function built around binary tree transforms, and have optimized it for zkVM performance to create the Eden ISA. The Eden ISA is built entirely around reduction rules for a single universal data structure form-fitted for zero-knowledge proofs, binary trees with finite field-element leaves. We have created a particular instantiation of an Eden zkVM in the form of a STARK (Succinct Transparent Argument of Knowledge), however other instantiations of Eden zkVMs using other cryptographic commitment schemes are equally viable.

We present an arithmetization of Eden's fundamental data structures and supply a proof of concept for its utilization in cryptographic proof systems by describing the architecture of a zkVM for proving an Eden computation within the STARK framework. We will describe 3 key contributions.

1.2.1 Binary Tree Arithmetization

We describe a linearized arithmetic encoding of Eden's *noun* data structure, which is based on binary trees. This involves two vectors, one which describes the tree shape and another which encompasses the leaf values stored in the tree. For the shape encoding, we utilize the bijection between binary trees and *dyck words*, which is motivated by the simplicity of performing tree concatenation in this representation. It transpires that dyck words are fundamentally related to depth-first traversal of a tree. The tree leaves are then placed in the *leaf vector* in depth-first order. See Section 5.3 for more details.

1.2.2 Interactive Oracle Noun Fingerprints

The only fundamental binary tree operations used in Eden are tree concatenation, or *cons*, and its inverse, tree decomposition. These operations are expressed very simply in our arithmetic encoding which leads to a linear time algorithm, but practical limitations demand greater efficiency.

In a typical scheme like hash-consing, the prover has to prove the correct computation of the hash of the trees. In our case, we leverage the interactive aspect of the interactive oracle proof model which gives the prover access to random values from the verifier, allowing us to utilize a technique we call **Interactive Oracle Noun Fingerprinting**. In this technique, we generate a fingerprint by transforming the dyck word and leaf vector into polynomials which are then evaluated on the random challenges given by the verifier; this pair of evaluations is our *noun fingerprint*. This scheme allows for quick, collision-resistant, probabilistic checking of the *cons* and tree decomposition relations.

1.2.3 Eden zk-STARK VM Architecture

The process of computing and verifying Eden has several central and distinguishable components which are described in Section 4.1, and also makes use of various primitives and fundamental techniques as described in Section 5. These components motivate our construction of a zk-STARK architecture for proving the correctness of an execution of Eden. In a zk-STARK context, independent components of the system can be separated into various *tables*. Each table has a distinct set of responsibilities where it either directly enforces a subset of Eden's invariants, or provides secondary functionality required by another table to do so.

Our system makes use of the following tables:

- **Stack Table** - records a full Eden computation trace and decodes each instruction
- **Noun Table** - validates encoded nouns and stores them for use in other tables
- **Subtree Access Table** - validates subtree accesses for the Stack Table
- **Exponent Table** - precomputes constants used to enforce *cons* and inverse *cons* relations in an efficient manner
- **Pop Table** - proves the correctness of pop operations performed on our stack data structure

We will need to cover a number of prerequisites before delving into our system design. We'll begin with a discussion of proof systems.

2 Proof Systems

Proofs are demonstrations of mathematical or computational facts, e.g. "Assuming axiom system $\mathcal{A}$, statement X implies statement Y," or "The computation $\mathcal{C}$ with input x outputs y."

Traditional mathematical proofs are, ideally, pieces of text of reasonable length, composed by a prover after some non-deterministic process involving intuition and guesswork. The resulting text can then be checked by handing it to a verifier who engages in a straightforward check of logical implications under reasonably limited time constraints. This roughly corresponds to the NP computational complexity class of decision problems.

In the last several decades, complexity theorists have been exploring less traditional conceptions of proof: interactive proofs (IPs), multiprover interactive proofs (MIPs), and probabilistically checkable proofs (PCPs), as well as variations on and refinements of these models.

At the core of these innovations is a rather more flexible understanding of what proofs are. Traditionally, a proof purports to offer ironclad guarantees of truth or falsity if one merely examines some hypotheses and logical deductions. This newer, innovative conception, called a **proof system**, is any protocol by which a prover can convince a verifier of a claim, where the meaning of “convincing” depends on the proof system.

In such proof systems, the prover is not necessarily a trustworthy entity, and there exists a possibility that a prover can convince the verifier of a false statement, or can fail to convince the verifier of a true statement. These possibilities are quantified by the **soundness** – the probability that the verifier will accept a false statement – and **completeness** – the probability that the verifier will accept the proof of a true statement.

2.1 Interactive Oracle Proofs

One of the recently introduced families of proof systems is the interactive oracle proof (IOP) model of [BCS16]. The IOP model combines aspects of the earlier IP and PCP models. Like the PCP model, it is expressive enough to characterize the complexity class NEXP, which contains the NP class of traditional proofs. However, the injection of interactivity from the IP model provides efficiency gains over the PCP model (in e.g. proof length). This fusion works by allowing the verifier in the interaction to probabilistically query the prover’s messages.

In an IOP, the prover and verifier interact over a number of rounds. In each round, the verifier sends the prover a challenge. Then, the prover sends the verifier a set of data in response to the challenge, which the verifier can partially query in this or later rounds. At the end of an agreed upon number of rounds, the verifier either accepts or rejects the proof.

The surprise challenges are meant to dissuade a malicious prover from giving into the temptation to deceive. There is a low probability that the prover will provide a falsified response that passes the verifier’s challenge.

We remark that an IOP can be made into a non-interactive protocol via the **Fiat-Shamir heuristic** [FS87]. The Fiat-Shamir heuristic is a transformation which substitutes the verifier’s messages with the output of a random function – this is a purely theoretical construction called a **random oracle**; in practice, it’s a cryptographic hash function – where the input is the totality of previously exchanged messages. To allow the verifier access to parts of the prover’s messages, this procedure is augmented to utilize cryptographic commitments to the data. The proof in this non-interactive variant is a collection of information related to the commitments, and the verifier’s role is to check this information for validity without additional communication with the prover. For details see [BCS16].

2.2 zk-STARKs

A zk-STARK - **z**ero-**k**nowledge **S**calable **T**ransparent **A**rgument of **K**nowledge - is a way to realize an IOP model with some very desirable properties. Before discussing these features it’s useful to motivate the utility of this system.

Imagine that a verifier wants some computation performed but does not have the resources to do the computation itself. It would like to outsource the computation to a prover who has more resources but may be untrusted by the verifier. The verifier would want some procedure to check the work of the prover that takes vastly less time than running the computation and can be accomplished by the prover in a practical amount of time. This is what a zk-STARK provides alongside the following guarantees:

1. **Scalability**: Verifying the computation takes exponentially less time than running it, and proving the computation incurs marginal overhead to running the computation.
2. **Transparency**: The verifier only has to trust hash functions and mathematics, not the prover or anyone else.
3. **Perfect completeness**: The verifier is always convinced by the protocol if the prover is honest.
4. **High soundness**: The verifier is only convinced by a malicious prover with astronomically low odds.
5. **Zero-knowledge**: The proof can hide information that the verifier wants to keep private, while still proving statements about this information.

A zk-STARK uses a technique called **arithmetization** to encode a computation. Computation is modeled as a sequence of steps, where at each step we update a fixed finite amount of information according to a set of rules. The information recorded at each step is arranged in a horizontal row, and the sequential ordering of the steps is reflected by stacking the rows vertically into a two-dimensional table. For example, imagine you're at the bank to cash several checks, and are filling out a deposit slip. You record the check amounts in the appropriate column, and have to compute the total amount. To avoid having to do the entire computation in your head, you can make a new column next to the given one in the margin which keeps a *running* total. See Table 1. The first entry in this column is just the first check's amount. To compute the second entry of the running total, add the second entry of the check amount column to the *first* entry of the running total column. In general, to compute the nth running total, add the nth check amount to the $(n-1)$st running total. The equation implicit in this rule applies at every pair of rows and describes the correct execution of the computation; it is called a *constraint*, an important concept in zk-STARKs. If every step is computed correctly, then the last running total is the total amount.

check amount	**running total**
132	132
406	538
55	593
113	706

Table 1: Filling a deposit slip.

Additionally, the prover can make use of as many distinct tables as is necessary to express a computation: a fixed finite number of tables can be utilized, each with their own set of constraints, as long as the tables are appropriately linked. One can think of different tables as computing different subcomponents of the same computation. The collective totality of the tables is referred to as the **trace** of a computation.

The table model of computation described above aids in addressing the scalability requirement: how can the verifier tell that the step-by-step computation was done correctly without doing every step themselves? It seems counterintuitive that this is even possible, since even one small misstep in a computation can wildly affect the intended output. The verifier would need some way to magnify an error in a single location so that it's noticeable.

Since the discovery of Reed-Solomon codes, it is well-known that polynomials are good at this kind of *error detection*, which one can see in the following way. Suppose we fit the simplest polynomial possible to some data points over a particular input domain. If we corrupt the data by changing even a single one of the data points, then either the polynomial will look radically different outside of the input domain, or its degree – the largest exponent of a variable needed to express the polynomial – will potentially differ from the original polynomial.

The zk-STARK protocol takes advantage of this property by (a) requiring that constraints be represented as polynomials and (b) requiring that table data be interpolated into polynomials. This is done so that verification, (i.e. applying constraints to the computation data) becomes equivalent to polynomial composition, which in turn yields a final polynomial that can be subjected to the error detection process. It is in this way that deceit by the prover is magnified and made visible to the verifier.

The techniques of arithmetization and polynomial error correction sit at the heart of the zk-STARK protocol. In a zk-STARK, a prover and verifier engage in one or more rounds of roughly four distinct phases: **commitment**, **challenge**, **response**, and **Low Degree Testing (LDT)**.

- In a **commitment** phase, the prover interpolates polynomials over various columns of the trace tables, evaluates these polynomials over a large input domain, and constructs a Merkle tree of the evaluations. The root of the Merkle tree is sent down the proof stream, serving as the commitment of the polynomial.
- In a **challenge** phase, the prover receives random challenges from the verifier.
- In a **response** phase, the prover provides a direct response to random challenges that are specifically used to reveal evaluations of parts of the commmitted polynomials.
- In an **LDT** phase, the verifier checks that one or more polynomials are of low degree.

Specifically, the phases are ordered in the following way:

(1) **Base Commitment** - The prover populates the *base columns* of the tables and commits to them in the proof stream. Base columns are columns that do not require randomness to compute.
(2) **Extension Challenge** - The verifier sends the first round of random challenges.
(3) **Extension Commitment** - The prover uses the randomness to compute *extension columns* which are interpolated and then committed.
(4) **Quotient Commitment** - The prover generates and commits to *quotient polynomials*, which encode information about the evaluation of constraints on the data.
(5) **Combination Commitment** - The prover creates and commits to the *nonlinear combination codeword polynomial.* This polynomial is used as a means to mask information from the verifier and enables the system to provide zero-knowledge properties.
(6) **Evaluation Challenge** - prover receives random challenges which are interpreted as requested indices to be opened in the Merkle trees committed to hitherto.
(7) **Evaluation Response** - The prover provides the requested leaves from the Merkle trees through a series of openings. Using the revealed values, the verifier performs a number of consistency checks on the polynomials that have been committed to ensure that they are properly linked to one another.
(8) **LDT** - The verifier runs the *FRI* protocol on the decommitment of the nonlinear combination codeword to check that it is of sufficiently low degree; if it is, the verifier accepts the prover's work.

Note that the random challenges sent by the verifier serve distinct purposes depending on the stage of the proof: in (2), they are used primarily by the prover as a means of computing additional columns that make use of randomness, while in (6), they are used by the verifier to actually request data from the prover.

To make our system sound, we augment the sketch provided above with an extra round of the Challenge/Commitment phases described by (2) and (3). This extra round prevents a potentially malicious prover from taking advantage of an extra degree of freedom that could otherwise be exploited.

While an in-depth explanation of the zk-STARK protocol is out of scope, we urge the reader to refer to [Ben+18] or [KR08] for further information.

In order to use a zk-STARK to prove arbitrary computation, we need both a suitable Turing-complete model of computation, and a way to encode this model of computation in terms of finite field elements. Therefore, we will proceed to describe Eden, our chosen model of computation, and its arithmetization.

3 Eden ISA

The Eden ISA is a specification of a Turing-complete combinator function, comprised of a minimalist set of reduction rules that map a data-code pair to a product. All data and code are represented using the same data type, the *noun*, which makes Eden a homoiconic language. Eden is a minimally modified version of Nock that is optimized for zkVM performance; to describe the modifications tersely, Eden replaces natural number atoms with finite field atoms, and replaces regular increment with finite field increment. What follows is an informal introduction to the Eden specification. For a more complete and detailed introduction, see the Nock specification in [Noc23].

3.1 Nouns

A **noun** is defined as either an atom, which is a finite field element, or a cell, which is an ordered pair of nouns.

This definition corresponds in a straightforward way to a full rooted binary tree whose leaves are finite field elements, like so:

```
    •
   / \
  0   •
     / \
    •   1
   / \
  6  20
```

Full means each tree node has either no children – the node is an atom – or two children – the node is a cell. **Rooted** means the tree has a distinguished node which is singled out as the top of the tree. From now on, we will drop these two adjectives and just use the term *binary tree.*

We write `[a b]` to denote a cell that consists of two nouns, `a` and `b`. In a cell `[a b]`, we say that the **head** of the cell is `a` and that the **tail** is `b`. We also say that `a` is the **left subtree** of `[a b]` and that `b` is the **right subtree** of `[a b]`.

The expression `[a [b c]]` is a way to write a cell whose head is the noun `a` and tail is a cell of nouns `b` and `c`. As an example, the noun pictured above is denoted as `[0 [[6 20] 1]]`.

Note well that EDEN is implicitly **right associative**. For example, the preferred way to write the expression `[a [b c]]` is `[a b c]`; the `b` and `c` to the *right* of `a` are assumed to *associate* into a cell. Similarly, we have `[a b c d] = [a [b c d]] = [a [b [c d]]]`, etc. As a concrete example, the preferred way to write `[0 [[6 20] 1]]` is `[0 [6 20] 1]`. Note that one cannot drop any more brackets to reduce this to `[0 6 20 1]`; the latter is equivalent, by convention, to `[0 [6 [20 1]]]`, which looks like this picture:

```
  •
 / \
0   •
   / \
  6   •
     / \
    20  1
```

A fundamental operation on nouns is **concatenation**, or *cons*. This is defined by cons(`a`, `b`) $=$ `[a b]`. This is seen to be similar to list concatenation by drawing `a` to the left of `b`; then cons(`a`, `b`) is formed by creating a tree with `a` as the left-hand side and `b` as the right-hand side.

The inverse of *cons* is then cons^{-1}(`a`) = (head(`a`), tail(`a`)), the decomposition of `a` into the pair of its left and right subtrees.

3.1.1 A Theorem on Binary Trees

We pause here to state a theorem on binary trees that will be useful later; the proof is by induction on the number of leaf nodes and left to the interested reader.

Theorem: If there are n leaf nodes in a binary tree, there are $2n - 1$ total nodes, and thus $n - 1$ internal, or non-leaf, nodes. The number of edges between nodes is $2n - 2$.

3.2 Operators

We define now a few operators that accept a noun and produce a new noun. Do note that EDEN uses a non-standard convention in which 0 denotes `true` while 1 denotes `false`.

The operators are:

- the noun type logical operator `?` that can be applied to any noun `a`:
  - `?a` will evaluate to `0` for a cell, and `1` for an atom.
- the equivalence logical operator `=` that can be applied to any two nouns:
  - `=[a a]` will evaluate to `0`, and `=[a b]` where `b` is a different noun than `a` will evaluate to `1`.
- the finite field increment operator `+` that can be applied to any noun `a`:
  - `+a` will evaluate to `(1 + a) mod p` where `p` is a predefined finite field for EDEN to operate on if `a` is an atom, and will crash if `a` is a cell.
- the `/` operator (sometimes referred to as the *slot* operator) that may be applied to nouns:
  - `/(1 a)` will output `a`, and may be applied to any noun.
  - `/(2 [a b])` will output `a`, and may only be applied when the input noun is a cell. If the input noun is an atom, crash.
  - `/(3 [a b])` will output `b`, and may only be applied when the input noun is a cell. If the input noun is an atom, crash.
  - If the head atom is greater than 3, recursively apply the `/` operator using the following reduction:

    - `/(2a b)` will reduce to `/(2 /(a b))` which will then be further evaluated until it reaches one of the base cases.
    - `/((2a + 1) b)` will reduce to `/(3 /(a b))` which will then be further evaluated until it reaches one of the base cases.
- the `#` operator functions recursively like the `/` operator, but rather than accessing a subtree, it modifies a subtree at a particular axis.

The slot operator is essentially a numerical addressing scheme for subtrees of a tree. The full tree gets address `1`; if a subtree has address `a`, its left and right subtrees get addresses `2a` and `2a+1`, respectively. Thus, moving to the left child of a node with address `a` adds `0` to the binary expansion of `a`, and moving to the right adds `1`. This gives a straightforward way of translating a subtree address into a sequence of left/right moves from the root: convert the address to binary; ignore the leading `1`, which corresponds to the tree's root; the rest of the digits, read left-to-right, correspond to left/right moves according to whether they are `0`/`1`. This can also be used in reverse to turn a sequence of left/right moves into an address. Note that the slot operator accepts a noun as input, and will traverse that noun in depth-first order, recursively evaluating until said atom is `1`, then traversing to the next atom in the noun.

3.3 The Eden Function

We define the Eden function as a function that takes an input noun, and returns an output noun or crashes. Input is formatted as a cell of `[subject formula]`, where `formula` is the code that is to be executed and `subject` is the data that the formula can reference, which can contain code itself owing to Eden's homoiconicity. The Eden function is defined below:

`#eden(x)` may be written as `*x`.

Denote "evaluates to" by $\mapsto$.

A valid Eden formula is always a cell.

If the head of the formula is a cell, Eden treats both head and tail as formulas, evaluates each against the subject, and produces the cell of their products. That is,
- `*[a [b c] d]` $\mapsto$ `[*[a b c] *[a d]]`.

In this case, we say that the formula is *cons* as shorthand (which is related to but not the same as the **cons** *operation*).

If the head of the formula is an atom, then it must contain an atom acting as a numeric opcode from `0` to `11`. If the head of the formula is higher than `11`, it is not valid. Invalid Eden instructions result in a crash.

In the following transformation rules, `a` is a subject and the following cell is the formula:

- `*[a 0 b]` $\mapsto$ `/(b a)`
- `*[a 1 b]` $\mapsto$ `b`
- `*[a 2 b c]` $\mapsto$ `*[*[a b] *[a c]]`.
- `*[a 3 b]` $\mapsto$ `?*[a b]`
- `*[a 4 b]` $\mapsto$ `+*[a b]`
- `*[a 5 b c]` $\mapsto$ `=[*[a b] *[a c]]`
- `*[a 6 b c d]` $\mapsto$ `*[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]`
- `*[a 7 b c]` $\mapsto$ `*[*[a b] c]`
- `*[a 8 b c]` $\mapsto$ `*[[*[a b] a] c]`
- `*[a 9 b c]` $\mapsto$ `*[*[a c] 2 [0 1] 0 b]`
- `*[a 10 [b c] d]` $\mapsto$ `#[b *[a c] *[a d]]`
- `*[a 11 b c]` $\mapsto$ `*[a c]`

The fundamental opcodes are 0-5, and 6-10 are essentially macros of 0-5. The fundamental opcodes can be divided into two categories: recursive and non-recursive. The non-recursive formulas are those with opcodes `0` or `1`, as their evaluations don't involve another invocation of the Eden function, while the recursive formulas are `2`, `3`, `4`, and `5`. Eden `11` is a formalism for dealing with extension instructions, or jets, which replace a formally specified computation with a faster version thereof.

When dealing with recursive formulas, the tail is either a noun `b` or a cell of nouns `[b c]`, and these nouns `b` and `c` are used as formulas themselves in defining the evaluations. We refer to them as **subformulas** of the outer formula.

Note that the evaluation of these subformulas against the subject must be computed in order to compute the original invocation of the Eden function; that is, `*[a b]` and/or `*[a c]` must be eagerly computed prior to the outer computation.

We observe that opcode `2` stands out since it involves *three* invocations of the Eden function `*` in its evaluation. The first subformula `b` is evaluated against the subject `a` and generates a *new subject*, while the second subformula `c` generates a *new formula*. Then, the new formula is evaluated against the new subject as follows: `*[new-subject new-formula]`.

For example, in the Eden expression `*[[0 4 0 1] 2 [0 2] 0 3]`, the subject is `[0 4 0 1]` and the formula is `[2 [0 2] 0 3]`. Thus to compute this expression, we compute a new subject via `*[[0 4 0 1] 0 2] = 0` and a new formula via `*[[0 4 0 1] 0 3] = [4 0 1]`. To complete the computation, we have to apply `*` one last time with this new subject and formula to find

`*[[0 4 0 1] 2 [0 2] 0 3] = *[0 4 0 1] = +*[0 0 1] = +0 = 1`.

4 Overview of Eden zkVM Design

The design of our Eden zkVM is a set of tables that communicate with each other and maintain Eden's computational invariants, which we discuss next.

4.1 Eden's fundamental invariants and operations

At a high level, the core loop of an Eden computation is as follows:

1. We are presented with a pair of nouns: a subject and formula.
2. The head of the formula has to be inspected, and depending on the result of the inspection, the formula is decoded into different subtrees.

3. If there are any subformulas, we execute them eagerly. Otherwise, we execute the decoded outer formula.

The following is a complete list of operations which are performed in the process of executing EDEN:
- the identity operation
- increment
- equality checking
- binary tree concatenation (*cons*)
- binary tree decomposition (inverse *cons*)
- *subtree access* (iterated tree decomposition)
- new recursive invocations of EDEN

Of these, only the operations involving *cons* require something beyond standard logical or arithmetic primitives.

From the description above, in order to capture the computational invariants of an EDEN computation, it is sufficient to have the following:

1. a formalism for manipulating nouns;
2. a means to inspect and decompose formulas;
3. a way to manage subcomputation order and a way to store intermediate results for completing earlier computations
4. a way to perform *cons*, *inverse cons*, and *subtree accesses* within our noun encoding;

It is based on these high level invariants that we motivate the design of our zkVM and its subsystems. Since the design is ultimately divided up into various tables, we now briefly describe each one and it's purpose.

4.2 Tables

For conceptual simplicity, our design involves multiple tables. Roughly, each table specializes in one of the core aspects of a generic EDEN computation that we enumerated above. The role of each table is to use its constraints to disallow traces that violate EDEN's invariants.

We now describe the core functionality of each table and how it relates to EDEN's invariants.

Stack Table

The stack table is the central table in our design. It models EDEN computation as a stack machine which directs the flow of computation according to the invariants of the EDEN specification. Accordingly, several other tables are its direct subsidiaries. The table makes use of various stacks as a means to keep track of subcomputation ordering.

Noun Table

All data expressed in proof tables must be encoded in terms of finite field elements, and nouns are no exception. The noun table's role is to store encoded nouns for later reference by other parts of the zkVM, and to prevent any improperly encoded nouns from being used in a computation. The table's design contains critical optimizations that allow us to validate nouns using our efficient linearized tree representation.

Subtree Access Table

The subtree access table performs Eden computations whose formulas have opcode `0` for the stack table, according to the semantics of the Eden specification for the `/` operator. Because the subject represents the totality of data accessible to the program, one can consider subtree access in Eden as an analogue of memory access found in traditional systems.

Decoder Table

The role of the decoder table is to serve the stack table with a means to inspect and decompose the formulas encountered throughout the computation, check that the formula nouns are correctly formed, and expose information such as the opcode and subformulas of the formula. Because of the tight coupling between the functionality that this table provides and what the stack table requires, it exists as a virtual table alongside the stack table.

Exponent Table

In order to perform *cons* and *inverse cons* correctly in our noun encoding, we require the computation of x^y for some quantities x and y that depend on the nouns being operated on. In practice, calculating x^y is non-trivial since x^y is not a polynomial quantity and ends up being quite costly. Thus, in order to amortize the costs, we batch computation of these exponentials into a single table. The exponent table's role is to compute these quantities correctly and share the results with other tables.

Pop Table

The pop table is a technical necessity that is required to ensure the validity of the pop operation on the various stacks used in the system, and is thus a subsidiary of the stack table. More detail about the stack data structure can be found in Section 6.2.

5 Arithmetization Fundamentals

5.1 Field Choice

We utilize the field $\mathbb{F}_p$ with prime cardinality $p = 2^{64} - 2^{32} + 1$ as our **base field**. This is the field we use to populate the cells in the tables before any randomness is generated. This field is widely referred to as the "Goldilocks field", despite the fact that the "Goldilocks" label was originally attached to the prime $2^{248} - 2^{224} - 1$ in [Ham15]. Much has been written about the virtues of this particular field, see e.g. [Por22], [Blo23], [Gou21]; they include fast arithmetic mod p, as well as support for efficient 32-bit arithmetic and fast Fourier transforms.

The main drawback of this field is its small size in relation to achieving our desired proof soundness. Thus, we use the unique up to isomorphism degree 3 extension $\mathbb{F}$ of $\mathbb{F}_p$. As a set this is $\mathbb{F} = \mathbb{F}_p^3$; note that $|\mathbb{F}| \approx 2^{192}$. Additive operations are performed component-wise according to the addition in $\mathbb{F}_p$, and multiplicative operations derive from $\mathbb{F}_p[x]$, modulo an irreducible cubic polynomial. Theoretically, any such polynomial gives rise to the same field up to isomorphism, but we use the depressed cubic $x^3 - x + 1$ because its coefficients are simple; irreducibility was verified in Sage.

Whenever we use the symbol $\mathbb{F}$ in the context of discussing our zkVM, $\mathbb{F}$ will denote this degree 3 extension.

5.2 Schwartz-Zippel Lemma

The main technical tool for proving the soundness of certain arguments throughout is the following:

Lemma (Schwarz-Zippel): Let $f(x_1, ..., x_k)$ be a nonzero multivariable polynomial with coefficients in a field $\mathbb{K}$, let d be the degree of f, and let S be a finite subset of $\mathbb{K}$. Then the number of zeroes of f in S^k is at most

$$d \cdot |S|^{k-1},$$

so the probability that $f(r_1, ..., r_k) = 0$ where $r_1, ..., r_k$ are drawn randomly and uniformly from S is at most

$$\frac{d \cdot |S|^{k-1}}{|S|^k} = \frac{d}{|S|}.$$

This lemma generalizes the elementary fact that the number of roots of a univariate polynomial over a field cannot exceed its degree; the proof is an uncomplicated induction from this fact.

A weaker version of the bound in this result was proved by Zippel in [Zip79]; in fact, he was bested by a year by DeMillo & Lipton in [DL78], whose names are sometimes attached to the result. The strong version given above was proved later by Schwartz in [Sch80].

5.3 Noun Encoding

Given that nouns are the fundamental data structure in Eden and we intend to prove Eden computations in the zk-STARK framework, our aim turns to the arithmetization of nouns.

Nouns have two distinct components: the shape of the tree, and the leaf data.

The fundamental connection between tree shape, leaf data ordering, and our arithmetization is depth-first traversal, which we briefly review.

5.3.1 Depth-First Traversal

Depth-first traversal is a fundamental algorithm for ordering the nodes of a binary tree. It can be described recursively in the following way: Add the current node to the list of nodes. Move to the left child node and do a depth-first traversal. Then move to the right child node and do a depth-first traversal.

As an example, here are the successive stages of the example noun we introduced above being ordered according to the depth-first algorithm:

```
         •          •          •
•       /          / \        / \
       0          0   •      0   •
                                /
                               •
```

```
  •           •           •
 / \         / \         / \
0   •       0   •       0   •
   /           /           / \
  •           •           •   1
 /           / \         / \
6           6  20       6  20
```

5.3.2 The Arithmetization of Nouns

Succinctly put, we use the binary sequence of left-child/right-child moves that are performed in a depth-first traversal of a tree as a description of the shape of the tree. Visually, the left/right moves correspond to an edge in the tree's graph from parent node to child node, so we naturally obtain a formal word w over the alphabet $\Sigma = \{L, R\}$, with L representing a leftward move and R representing a rightward move. Critically, the resulting language $D \subset \Sigma^*$ coincides with the well-known dyck language, which we will discuss shortly in Section 5.3.3. Given that our goal is to express these words in terms of finite field elements, we need to choose two field elements to represent L and R. Since all fields contain the two distinguished elements 0 and 1, and the addressing scheme for subtrees associated with the logical operator `/` already associates 0 with left and 1 with right, we use 0 and 1 as the arithmetic representations of L and R.

To encode the natural numbers at the leaves of the noun, we place them into a separate vector, where depth-first traversal naturally determines the order, in alignment with the shape encoding. With these two components in place, a noun can be completely and unambiguously characterized and represented in our encoding.

How to Arithmetically Encode a Noun:

- Initialize vectors for the binary word describing the tree shape and for the leaf data.
- Perform a depth-first traversal of the noun. Append a 0 or 1 to the word for every move from a parent node to a left or right child node, respectively. For each leaf encountered, append the data at the leaf to the vector.

Thus, for the noun `[0 [6 20] 1]` pictured in Section 3.1, the arithmetization of the noun is the (word, vector) pair $(010011, \langle 0, 6, 20, 1 \rangle)$.

If the noun is merely an atom, then the word describing the shape is empty and we denote this with the symbol ϵ.

We note by Section 3.1.1 that if the length of the vector component of the noun encoding is n then the binary word component has length $2n - 2$, since the letters of the word correspond to the edges in the tree.

We call a word that describes the shape of a noun the **dyck word** of the noun and the vector containing the leaf data the **leaf vector** of the noun. We now turn to a deeper explanation of the various properties of dyck words.

5.3.3 Dyck Words

If T denotes a noun, we use T_d to denote the dyck word of T, and T_l to denote the leaf vector of T.

Now assume T is a cell, and let L and R denote the left and right subtrees, i.e. the head and tail, of T. Then the following fundamental relations hold:

$$T_d = 0L_d1R_d \tag{1}$$

$$T_l = L_lR_l \tag{2}$$

These follow immediately from the recursive description of depth-first traversal. To see (1), note that to depth-first traverse a tree (T_d), we first move to the left subtree (0), then depth-first traverse the left subtree (L_d), then move to the right subtree (1), then depth-first traverse the right subtree (R_d). To see (2), use the same recursive description of depth-first traversal: since we traverse T by first traversing L in depth-first traversal order, the left subtree's leaves will be first in the list of T's leaves, and then we traverse R and find its leaves in depth-first order.

Relation (1) was in fact our main motivation for anchoring our arithmetization to depth-first traversal, as opposed to other potential schemes, since it is intimately related to the fundamental tree operations *cons* and inverse *cons*, and it is imperative that these operations can be efficiently encoded. Relying on a dyck word-based encoding allows us to unlock performance characteristics that would otherwise not be possible under a scheme utilizing hash-consing.

We've seen that a binary tree gives rise to a word that satisfies (1). If we flip (1) on its head and use it to define a language we obtain the **dyck language** of **dyck words**:

A **dyck word** w over the alphabet 0, 1 is either the empty word ϵ or has the form

$$w = 0x1y \tag{3}$$

where x and y are dyck words.

We point out that, *a priori*, the set of dyck words could be larger than the set of dyck words *of binary trees*. However, it's easy to convince oneself that this is not the case. First, the simplest dyck word is ϵ and this is the dyck word of the trivial tree with one node. We can then argue inductively from (3): if a set of dyck words $\{x_i\}$ are the dyck words of some binary trees, i.e. $(T_i)_d = x_i$, then each word in the set of all words $0x_i1x_j$ is the dyck word of a tree, namely $[T_i \; T_j]_d = 0x_i1x_j$. But according to the definition, every dyck word is generated inductively from the single word ϵ via relation (3), so every dyck word represents the shape of a binary tree.

Now, we present a nice theorem which characterizes how to recognize a dyck word nonrecursively within $\{0, 1\}^*$ by merely reading it left-to-right, which will become relevant when we write constraints that recognize dyck words.

Theorem: The set of dyck words over the alphabet $\{0, 1\}$ is characterized by two properties:

1. While reading left-to-right, the number of 0's encountered so-far is at least as large as the number of 1's.
2. The total numbers of 0's and 1's are equal.

If $w = w_0 ... w_{n-1}$ then these properties are equivalent to the constraints

1. $\sum_{i=0}^{j} (1 - 2w_i) \geq 0$ for $j = 0, ... n - 1$.
2. $\sum_{i=0}^{n-1} (1 - 2w_i) = 0$

The fact that a dyck word has these properties is easy to verify by induction from the definition, and the fact that the empty word has these properties. The converse can also be proven by induction, but is a bit more involved.

5.4 Optimized Noun Encoding

While the encoding of nouns as $(\text{dyck-word}, \text{leaf-vector})$ pairs is a noteworthy and necessary step on the road to arithmetization, it is insufficient by itself. This is due to the cost within our zkVM of any manipulations of the literals in the undoubtedly large $(\text{dyck-word}, \text{leaf-vector})$ pairs that would occur in any realistic EDEN computation.

What is called for is a compression mechanism that would make the representation of nouns more succinct, but without compromising the ability to perform the fundamental tree operation, *cons*.

5.4.1 Polynomial Encoding of Nouns

To accomplish this compression, we begin by using an equivalent of the $(\text{dyck-word}, \text{leaf-vector})$ encoding, in the form $(\text{len}, \text{dyck-poly}, \text{leaf-poly})$. The dyck-poly and leaf-poly components are polynomial forms of the dyck word and leaf vector of the noun, created in the following way:

$$(a_0, ..., a_k) \mapsto \sum_{i=0}^{k} a_i x^{k-i} \tag{4}$$

where $(a_0, ..., a_k)$ is any finite ordered tuple of field elements. If the tuple is empty, the convention is that we map to the zero polynomial.

If T is a noun, we will write $T_d(x)$ and $T_l(x)$ for the dyck and leaf polynomials, respectively, and $\text{len}(T)$ for the length.

For example, if the noun is `[0 [6 20] 1]`, the encoding is $(4, x^4 + x + 1, 6x^2 + 20x + 1)$. On the other hand, the encoding of an atom `a` is $(1, 0, a)$.

The len component is the *length of the tree*, which we define to be the number of leaf nodes; i.e. $\text{len} = \text{length}(\text{leaf-vector})$. Recall from the theorem of Section 3.1.1 that the number of leaf nodes of a tree determines the number of total nodes and the number of edges. Through this, len implicitly encodes information about the length of the dyck word since it directly corresponds to the number of edges, thus making it a comprehensive metric for the size of the tree.

The len component is also convenient for restoring any leading 0's which become invisible when the dyck word and leaf vector are converted into polynomials; we will point out other ways in which it is useful later. However, including the length in the encoding is not strictly necessary. Briefly, one can always restore the correct number of 0's to recover the dyck word from the dyck polynomial since the number of 0's has to be equal to the number of 1's. Then, the length of the leaf vector can be recovered from the length of the dyck word, which can be used to restore any missing 0's to the leaf vector from the leaf polynomial.

5.4.2 Cons and Inverse Cons in the Polynomial Encoding

We can also write the *cons* relations of (1) and (2) in polynomial terms:

$$T_d(x) = 0 \cdot x^{2 \cdot \operatorname{len}(L) + 2 \operatorname{len}(R) - 3} + x^{2 \cdot \operatorname{len}(R) - 1} L_d(x) + 1 \cdot x^{2 \cdot \operatorname{len}(R) - 2} + R_d(x) \tag{5}$$

$$T_{l(x)} = x^{\operatorname{len}(R)} L_l(x) + R_l(x) \tag{6}$$

The powers of x are merely to shift the different components of the original *cons* formulas so there is no additive "mixing" of the word/vector literals.

We also have the straightforward identity

$$\operatorname{len}(T) = \operatorname{len}(R) + \operatorname{len}(L) \tag{7}$$

Now we present an algorithm for performing inverse *cons* (that is, the computation of L and R from T): Simply write down $(\operatorname{len}(L), L_d(x), L_l(x))$ and $(\operatorname{len}(R), R_d(x), R_l(x))$ and check that (5), (6), and (7) hold for the given T! This is a nondeterministic algorithm: we don't worry about how L or R were produced, we only care that we can efficiently check that they are correct. This is useful in an IOP context precisely because traces are computed *after* a program has been fully executed, allowing a prover to have access to such non-deterministic information.

5.4.3 Interactive Oracle Noun Fingerprint

We now use the random and interactive elements of the IOP model in a critical technique known as Interactive Oracle Noun Fingerprinting (ION Fingerprinting). After committing to the dyck word and leaf vector literals, the verifier supplies the prover with randomness. Say α_1 and α_2 are two such values. Let w_i and l_i denote the components of the dyck word and leaf vector of some noun with length λ, so that

$$(\text{len}, \text{dyck-poly}, \text{leaf-poly}) = \left(\lambda, \sum_{i=0}^{2\lambda-3} w_i x^{2\lambda-3-i}, \sum_{i=0}^{\lambda-1} l_i x^{\lambda-1-i}\right) \tag{8}$$

is the polynomial representation of the noun. Then we fingerprint the dyck and leaf polynomials by evaluating them at the α_i, producing the field elements (*felts*) that comprise the ION fingerprint:

$$(\text{len}, \text{dyck-felt}, \text{leaf-felt}) = \left(\lambda, \sum_{i=0}^{2\lambda-3} w_i \alpha_1^{2\lambda-3-i}, \sum_{i=0}^{\lambda-1} l_i \alpha_2^{\lambda-1-i}\right)$$

This fingerprinting scheme is homomorphic because for any felt α, the evaluation map $f(x) \mapsto f(\alpha)$ is a ring homomorphism $\mathbb{F}[x] \to \mathbb{F}$. We will make use of this property in the following section.

An advantage of including the length in the noun representation emerges here for security: the only possibility for collisions in this representation can occur between nouns of the same length. Since distinct polynomials evaluated on a random input will almost certainly give different outputs over a large domain, the probability that two different nouns have the same length, dyck felt, and leaf felt is negligible; see Section 8.1 for a more involved analysis.

5.4.4 Cons in the Felt Triple Representation

Since the polynomial *cons* relations are polynomial identities, these can, with high probability, be validated by checking that each side has identical evaluations on random field elements. Thus, if $\mathrm{len}(T) = \mathrm{len}(R) + \mathrm{len}(L)$, then

$$T_d(\alpha_1) = 0 \cdot \alpha_1^{2\cdot \mathrm{len}(L) + 2\cdot \mathrm{len}(R) - 3} + \alpha_1^{2\cdot \mathrm{len}(R) - 1} L_d(\alpha_1) + 1 \cdot \alpha_1^{2\cdot \mathrm{len}(R) - 2} + R_d(\alpha_1) \tag{9}$$

$$T_l(\alpha_2) = \alpha_2^{\mathrm{len}(R)} L_l(\alpha_2) + R_l(\alpha_2) \tag{10}$$

and it is overwhelmingly likely that $T = \mathrm{cons}(L, R)$. Note that, theoretically, the *cons* relations (9) and (10) have the same form as the relations (5) and (6) precisely because the evaluation map is a ring homomorphism.

For performing inverse *cons* in this representation (outlined in Section 5.4.2), we can use the same nondeterministic algorithm we employed in the polynomial encoding with the x variables replaced by the appropriate α_i.

6 Probabilistic Structures and Algorithms

6.1 Multiset Arguments

A multiset is a set where elements can have multiplicity, i.e. can appear multiple times. A multiset argument – called a permutation range check in [GPR21] – is a way to argue whether two multisets are the same. In our context, the multisets could be the contents of two columns (in the same or even different tables), or even the rows of two different tables.

The algorithm used is probabilistic, and works like this. Two multisets $\{a_i\}$ and $\{b_j\}$ are encoded as the roots of polynomials $p(x) = \prod(x - a_i)$ and $q(x) = \prod(x - b_j)$. These polynomials are equal if and only if the multisets are equal, since the polynomial ring $\mathbb{K}[x]$ over a field $\mathbb{K}$ is a unique factorization domain. If the field is large, we can tell with high probability whether these polynomials are different by evaluating them on a random field element and checking whether the evaluations differ. The only possibility of error is a false positive, where two different polynomials happen to have the same evaluations. By the Schwartz-Zippel lemma, the probability of such a false positive is bounded by the maximum degree of the two polynomials, divided by $|\mathbb{K}|$.

The random field element, say β, is generated by the verifier after the elements of the multisets are committed to by the prover. It is then very easy to compute the evaluation of the polynomial in linear time by repeated accumulative multiplication of the linear factors $(\beta - a_i)$ corresponding to the multiset elements a_i.

By an abuse of terminology, we sometimes refer to the evaluation $p(\beta) = \prod(\beta - a_i)$ as "a multiset."

6.1.1 Compressing Tuples into Multisets

If we want to argue that multisets of tuples are equal, we have to reduce the tuple to a field element in an acceptable way to utilize the argument above. The standard technique is to use more verifier randomness to compress the tuple into a field element.

If the tuple of field elements in a particular row is, say, (x, y, z), then we can use verifier randomness to form the single field element $ax + by + cz$. Doing the same in each row, we can then use the argument above on the resulting multisets of field elements.

For the soundness of this procedure, which uses the Schwartz-Zippel lemma in the multivariate linear case, see [Sze].

6.1.2 Multisets in the Eden zkVM

We link various tables together using multiset arguments to ensure that the tables are maintaining invariants relating to the same computation. Some tables want to offload computational tasks to other tables and assume the answers to certain questions; they put the questions and answers in a multiset and check that it matches the **computed** question-and-answer from another table.

For example, every table that uses ION fingerprints needs to know that these fingerprints are validly constructed. Thus the noun table produces multisets of valid ION fingerprints, while other tables make multisets of felt triples which they claim are ION fingerprints. The zkVM then includes constraints that enforce the equality of the multisets, which implies the validity of the claimed fingerprints. As another example, the stack table assumes knowledge of formula decompositions (computing them non-deterministically), which are then checked by the formula decoder table.

The multisets in our design are as follows:

- The noun table creates a multiset for every table except the exponent table to give tables correct noun representations.
- The decoder table creates a multiset for the stack table containing tuples of the form $(\text{formula},\ \text{decomposition})$, where the decomposition component is itself a tuple, describing the decomposition of formula nouns.
- The pop table creates a multiset for the stack table containing valid stack states. (See Section 7.1.5 for more details.)
- The subtree access table creates a multiset for the stack table containing tuples of $(\text{subject}, \text{axis}, \text{output-of-subtree-access})$.
- The exponent table creates multisets of tuples of the form $(\text{power}, \text{randomness-to-power})$ for each table which has to perform *cons* or *inverse cons* (this includes the stack, decoder, and subtree access tables).

6.2 Stacks in the Eden zkVM

Since stacks have an ordering just like the terms in polynomials and polynomials are a primary object in proof systems, it seems natural to try to model the dynamics of a stack using polynomials.

Imagine a sequence of items (i_j) being pushed to and popped from a stack:

item	push	pop
i_1	1	0
i_2	1	0
i_3	0	1
i_4	1	0
i_5	1	0
i_6	0	1
i_7	0	1

Here, 1 means we perform the operation in that column. In order for this to be correct, we must have $i_3 = i_2$, $i_6 = i_5$, and $i_7 = i_4$. Assuming this to be the case, we can show the state of the stack as a polynomial in each row after the operation in that row has been performed:

item	**push**	**pop**	**poly**
i_1	1	0	i_1
i_2	1	0	$i_1x + i_2$
i_3	0	1	i_1
i_4	1	0	$i_1x + i_4$
i_5	1	0	$i_1x^2 + i_4x + i_5$
i_6	0	1	$i_1x + i_4$
i_7	0	1	i_1

The push operation in polynomial terms is as follows,

$$\text{push}(i, s(x)) = x \cdot s(x) + i \tag{11}$$

while pop is

$$\text{pop}(i, s(x)) = \frac{s(x) - i}{x} \tag{12}$$

provided that the constant term of $s(x)$ is i, i.e. $s(0) = i$.

Obviously, the pop operation is more difficult to verify, given the condition on its evaluation. Note that $s(0) = i$ if and only if there is a *polynomial* $\hat{s}(x)$ such that $s(x) = x \cdot \hat{s}(x) + i$, and that $\hat{s}(x)$ is *unique* and precisely $\text{pop}(i, s(x))$.

Thus, we can reconsider pop as a nondeterministic algorithm that returns the new state of the stack and the item popped:

$$\text{pop}(s(x)) = (\hat{s}(x), i)$$

The above equation is subject to $s(x) = x \cdot \hat{s}(x) + i$ where $\hat{s}$ is a polynomial and both $\hat{s}$ and i are "guessed."

Now return to our table but add an $\hat{s}$ column:

item	**push**	**pop**	**poly**	$\hat{s}$
i_1	1	0	i_1	
i_2	1	0	$i_1x + i_2$	i_1
i_3	0	1	i_1	
i_4	1	0	$i_1x + i_4$	
i_5	1	0	$i_1x^2 + i_4x + i_5$	$i_1x + i_4$
i_6	0	1	$i_1x + i_4$	i_1
i_7	0	1	i_1	

Let v' be the variable which denotes the value of the variable v in the next row. Whenever pop' is 1, we imagine there to be a constraint that says $\text{poly} = x \cdot \hat{s} + \text{item}'$ which guarantees that item can be popped and that $\hat{s}$ is the next stack state; thus we also imagine a constraint forcing $\text{poly}' = \hat{s}$ in this instance.

In the actual zkVM we have to use field elements in cells and not polynomials involving an indeterminate variable x, but we can do this by replacing the x in the previous figure with a verifier-provided random value γ. The identities

$$\text{poly}(x) = x \cdot \hat{s}(x) + i$$

are now verified with high probability via

$$\text{poly}(\gamma) = \gamma \cdot \hat{s}(\gamma) + i.$$

provided that the $\hat{s}$ polynomials are validly accumulated and made available. This is the purpose of the Pop Table, explained in the next chapter.

7 Eden zkVM Design

7.1 Tables

In what follows, if x denotes the value of a variable corresponding to a column at some row, then x' denotes the value of the variable at the successive row.

7.1.1 Stack Table and Decoder Table

7.1.1.1 Background

As mentioned previously, the core functionality of the stack table is to model valid Eden computation and reject any computation that does not follow the proper semantics.

It accomplishes this by modeling a stack machine that executes Eden, always eagerly processing the subformulas of the current computation and then processing the outer formula with the results. When non-recursive formulas are encountered, their result is computed directly.

In other words, it can be described as an improved version of a tree-walking interpreter that has been linearized in order to to fit more neatly within the tools that the AIR formalism has to offer.

While the table is central to the overall zkVM, it outsources a number of responsibilities to other tables, namely
- noun validation to the noun table,
- Eden 0s calculation to the subtree access table
- stack popping validation to the pop table

7.1.1.2 Description

The stack machine has 3 stacks:

- a **compute stack** (CS), which is used to hold the remaining subcomputations to execute, which come in subject/formula pairs
- a **product stack** (PS), which is used to hold the results of both non-recursive (i.e. terminal) and intermediate subformula executions
- an **operation stack** (OS), which stores **machine-specific** operations as well as deferred Eden operations to be done after subformulas have been computed.

Deferred opcodes are either lone atoms or tuples containing additional context to be used after processing subformulas of the opcode in question. The machine-specific operations allow the stack machine to perform critical tasks in addition to executing EDEN opcodes themselves.

The primary machine-specific operation used is pop2, which takes a subject/formula pair off of the compute stack and sets it as the next computation to run. When combined with the OS, pop2 allows the machine to delineate subcomputation boundaries in the otherwise flat compute stack and allows the machine to descend into a subcomputation while preserving the context of earlier computations.

The other machine-specific operation is cons, which signals to the machine that the formula is *cons* and needs to be handled accordingly.

The stack machine also has a few registers:
- `subj` - the current subject
- `form` - the current formula
- `op` - the current operation to perform, taken from the OS

Moreover, we can think of the machine as having 3 states:

- **start** - this is the initial state of the machine. No computation has been executed, but the subject and formula are loaded on the computation stack
- **middle** - this is the state during which EDEN is recursively evaluated
- **end** - this is the final state of the machine, when no further computation is left to complete on the computation stack, and the result of the EDEN computation is available on the product stack.

These states are helpful for thinking about the machine but do not need to be explicitly encoded in the trace, and can be implemented through constraints implicitly.

7.1.1.3 Stack Machine Execution

The machine has a core loop that it executes until completion, which we now describe as follows:

1) Start off with
   - `subj` set to empty
   - `form` set to empty
   - CS set to the initial computation `[s f]`
   - PS set to empty
   - OS set to pop2
   - `op` set to empty
2) Then, pop an operation off of the OS and move it into `op`.
3) Switch on `op`:
   a) If the operation is pop2,
      i) Pop a value off of the CS and assign that to `subj` and pop another value off of the CS and assign that to `formula`.
      ii) Then, deconstruct the current formula using *inverse cons* into the subformulas, zip them up into subject/formula pairs, and push them onto the compute stack to queue them up for execution. If the formula is a *cons* cell, then the left and right subtrees of the formula will be zipped with the subject and pushed onto the compute stack instead.

    iii) Push the opcode of the formula to the OS such that when the subformulas have been processed, we can come back to the outer instruction and finish processing it. If the formula was a *cons* cell, push the cons operation to the OS.
    iv) Push a pop2 on the OS for each subcomputation that will need to be evaluated.
  b) If the operation is an Eden opcode, finish executing the instruction by manipulating the CS and PS appropriately based on the opcode using the intermediate results stored on the PS.
  c) If the operation is cons, finish processing it by *cons*ing the top two items on the product stack to produce a cell.

4) If the OS is empty, then the final result of the computation is the single remaining value on the PS. Otherwise, the machine returns to Step 2, and advances by recording state in the next row of the trace.

7.1.1.4 Formula Deconstruction

Formula deconstruction is the process of taking the full formula noun and extracting out opcodes and subformulas, in addition to detecting *cons* formulas when encountered. This process is constrained by the formula decoder table which exists as a virtual table attached to the stack table.

It can be summarized by the following table, where f is the formula and the remaining columns to the right are its deconstruction.

f	**op**	**is-cons**	**sub-1**	**sub-2**
`[0 b]`	0	N	`b`	
`[1 b]`	1	N	`b`	
`[2 b c]`	2	N	`b`	`c`
`[3 b]`	3	N	`b`	
`[4 b]`	4	N	`b`	
`[5 b c]`	5	N	`b`	`c`
`[[b c] d]]`	cons	Y	`[b c]`	`d`

Table 5: Formula deconstruction

7.1.1.5 Translating from Eden Execution to Stack Manipulation

Each operation that exists on the OS has a strictly defined set of transformations that it applies to the CS, PS, and OS. We will now describe these semantics.

Below are the semantics of the pop2 operation.

s	**f**	$\mathbf{CS}_{\text{new}}$	$\mathbf{PS}_{\text{new}}$	$\mathbf{OS}_{\text{new}}$
`a`	`[0 b]`		/(b a)	
`a`	`[1 b]`		`b`	
`a`	`[2 b c]`	`a,c,a,b`		pop2, pop2, 2
`a`	`[3 b]`	`a,b`		pop2, 3
`a`	`[4 b]`	`a,b`		pop2, 4
`a`	`[5 b c]`	`a,c,a,b`		pop2, pop2, 5
`a`	[[b c] d]]	`a,d,a,[b c]`		pop2, pop2, cons

Table 6: pop2 semantics

To keep the table terse, we employ a shorthand: for the stack related columns, an empty cell means the value remains unchanged, while a filled cell means that the included value(s) are to be pushed to the stack.

Below are the semantics of the deferred EDEN opcodes, including the cons instruction.

op	$\mathbf{PS}_{\text{old}}$	$\mathbf{PS}_{\text{new}}$	$\mathbf{CS}_{\text{old}}$	$\mathbf{CS}_{\text{new}}$
2	n_1, n_2, P	P	CS	n_1, n_2, CS
3	n, P	$?\,(n), P$	CS	CS
4	n, P	$+(n), P$	CS	CS
5	n_1, n_2, P	$=[n_1\ n_2], P$	CS	CS
cons	n_1, n_2, P	$[n_1, n_2]$, P	CS	CS

Table 7: Deferred opcode semantics

In other words, the above table describes the behavior of the stack table when it sees any of the deferred EDEN operations.

Table 6 and Table 7 define the totality of how the stack machine operates. The polynomial constraints for the stack table are constructed to ensure that all transformations are properly applied, and that any invalid transformation is rejected.

7.1.1.6 Inverse Cons *is* Memory Access

One key optimization that can be made is by taking advantage of EDEN's homoiconicity. As mentioned earlier, we deconstruct formulas using *inverse cons*. For instance, a formula such as `f = [4 b]` gets deconstructed into `op=4`, `form=b` by proving that an *inverse cons* relation `f` $=$ cons(4, b) holds. However, we observe that the *inverse cons* relation above can be phrased as proving the memory accesses `*[f 0 2]` $\mapsto$ 4 and `*[f 0 3]` $\mapsto$ `a`.

This observation generalizes to deconstructing other formulas with 2 or 3 subformulas, and thus the whole formula deconstruction process can be implemented in terms of memory accesses operating on formulas as the subject. In light of this, we can completely obviate any additional constraints in the table that are used to prove the *inverse cons* relation when deconstructing formulas and instead reuse the logic already present in the subtree access table.

7.1.2 Noun Table

To compute EDEN, one needs a model of its fundamental data structure, the noun, and, as discussed, our model is the ION Fingerprint: a length component, and two evaluations on verifier randomness of polynomial forms of dyck words and leaf vectors. The noun table encodes nouns into the ION Fingerprint, and rejects any attempts at passing ill-formed nouns into the computation. For example, a malicious prover would not be able to pass 001 of as the dyck word of a noun. The main purpose is to share these fingerprints with other tables so that they have a compact representation of nouns for computation. The sharing is accomplished via multiset arguments with other tables. *Every* noun that is utilized in the program has to be validated by the noun table.

To understand the noun table, it's helpful to mentally divide it into *segments*, each of which is a contiguous set of rows. Each segment is responsible for encoding one noun and adding it to the multiset with a certain multiplicity. Additionally, keep in mind that each segment has two *phases*, the accumulation phase and the multiplicity phase, which are elaborated below. In the accumulation phase, the dyck word is checked for validity and the dyck-felt and leaf-felt are accumulated in extension columns. In the multiplicity phase, the result of this accumulation is placed repeatedly in a multiset.

One of the primary variables in the noun table is dyck, which denotes the column where the dyck words are input by the prover. At the beginning of a segment of the noun table, the variable rem, for "remaining," is initialized by the prover. It indicates the number of letters remaining until we reach the intended end of the dyck word for this segment, i.e. until the accumulation phase is completed.

The dyck column is constrained to only contain 0's and 1's in each cell. We can recall from the theorem of Section 3.1.1 that the constraints

1. $\sum_{i=0}^{j}(1-2w_i) \geq 0$ for $j = 0, ...n-1$.
2. $\sum_{i=0}^{n-1}(1-2w_i) = 0$

characterize dyck words, where w_i denotes a binary letter. In the noun table, the partial sum $\sum_{i=0}^{j}(1-2w_i)$ is denoted by the column variable ct for "count" (though it's properly a *weighted* count). We enforce the constraint $\text{ct}' = \text{ct} + (1 - 2\ \text{dyck}')$ throughout the accumulation phase to maintain ct's character as a running sum. It's easy to encode the second constraint, which is that $\text{ct} = 0$ at the end of the accumulation phase. To encode the first constraint above we need to work harder since we're encoding an inequality in terms of polynomial equalities. The basic idea is that since the weighted count can only increment or decrement, if ct ever went below 0 it would have to hit -1, and so we encode $\text{ct} \neq -1$ instead. We do this by introducing the variable ict-i, for "increment-of-count's inverse", which is the true inverse of $\text{ct} + 1$ when $\text{ct} + 1 \neq 0$, and 0 otherwise. Then the constraint $(\text{ct} + 1) \cdot \text{ict-i} = 1$ prevents $\text{ct} = -1$.

The column leaf contains the components of the leaf vectors, which aren't constrained during the accumulation phase. The column len counts up from 1 for each 1 in the dyck word; since a tree of length n has a dyck word with $2n - 2$ letters, half of which are 1's, this increments the variable len $n - 1$ times from 1 to the correct final value of n.

In the extension columns dyck-felt and leaf-felt, the values in the dyck and leaf columns are accumulated into the evaluations of the dyck and leaf polynomials at the random values α_1, α_2. While in the accumulation phase,

$$\text{dyck-felt}' = \alpha_1 \cdot \text{dyck-felt} + \text{dyck}$$

$$\text{leaf-felt}' = \text{dyck}' \cdot (\alpha_2 \cdot \text{leaf-felt} + \text{leaf}) + (1 - \text{dyck}') \cdot \text{leaf-felt}$$

The second constraint here is a conditional on the next character of the dyck word, and works because of some tree combinatorics: we're currently at a leaf if we can't go down into the tree any further from our current position, which means the next symbol of the dyck word is 1. So we accumulate leaf components when the next dyck character is 1.

Once this accumulation phase is complete, we enter the multiplicity phase. The multiplicity mult is a variable whose value is set by the prover at the top of a segment and remains constant throughout the accumulation phase. In the multiplicity phase it decrements from row to row. With each decrement, the values in the len, dyck, and leaf columns after the accumulation phase are compressed and put into a multiset mset:

$$\text{mset}' = \text{mset} \cdot (\beta - (a\ \text{len}' + b\ \text{dyck-felt}' + c\ \text{leaf-felt}'))$$

where β, a, b, and c are verifier-supplied random values.

Atomic nouns are an edge case due to their empty dyck words and have to be handled specially, but this is not terribly difficult and we omit the details here.

We close out this section by mentioning that this basic design can be optimized. The basic problem with the design above is that there's substantial redundancy. For example, imagine two nouns with large identical left subtrees but distinct right subtrees; these left subtrees would both be present in the table twice, taking up double the number of rows. In the optimized design, we pass over the literals of certain nouns in a first phase and record them as above. Then in a second phase, we *cons* the nouns recorded in the first phase to create larger composite nouns and record them in a multiset. This *cons* operation occurs in one row, yielding a significantly more efficient algorithm.

7.1.3 Subtree Access Table

Recall that the first EDEN formula opcode is `0`, and is formally specified as `*[a 0 b]` $\mapsto$ `/(b a)`. Informally, `b` is an axis number and `/(b a)` returns the subtree of `a` with that axis number. As `a` is the subject in the EDEN expression `*[a 0 b]`, and the subject/formula dichotomy in EDEN is analogous to the familiar data/code distinction, we can conceptualize an EDEN 0 computation as the access of stored data, i.e. a memory access. The subtree access table is our dedicated instrument for these memory accesses. It receives "requests" from the stack table when EDEN 0 computations are encountered, and pipes answers back.

Just as with the noun table, it is convenient to mentally divide the subtree access table into *segments*, each with two phases. Each segment is dedicated to a single EDEN 0 computation with a particular subject and axis. For a given $(\text{subject}, \text{axis})$ pair, the first phase is the *traversal phase*, wherein the path into the subject determined by the axis is traversed and the correct subtree identified. The second phase is the *multiplicity phase*, wherein the output subtree is placed in a multiset repeatedly.

The basic idea is simple: the target axis contains within its binary digits, after stripping the leading 1, left/right instructions encoded as 0/1 for reaching the target. So, starting with the given noun, we continually decompose nouns into left/right subtrees and select a new noun according to whether we see a 0 or 1 as the next digit. This new noun becomes the target of further decomposition until we exhaust the available digits.

To understand this, it will be helpful to jump into a schematic of the mechanism, as seen in the following figure.

path	**axis**	**tarax**	**tree**	**left**	**right**
1	1	13	S	S_L	S_R
0	3	13	$T = S_R$	T_L	T_R
1	6	13	$U = T_L$	U_L	U_R
	13	13	$V = U_R$		

Table 8: Idea of subtree access table

Note that in this figure the tree, left, and right columns are each aggregates of three columns in the actual subtree access table, representing the three components of the ION Fingerprint.

The tarax column contains the target axis. The path column contains the binary digits of this target stripped of the leading 1. These digits must be entered correctly because the axis column starts at 1 and uses the binary path digits to build back up to the target:

$$\text{axis}' = 2 \cdot \text{axis} + \text{path}$$

Since binary expansions are unique, any wrong path digit will lead to the axis never hitting the target value. If the axis value never hits the target, the result of the axis lookup is never recorded.

In each row, we have a tree and its left and right subtrees. This decomposition is enforced by *cons* constraints verifying that the left and right subtrees are correct.

The 1 in the first row of the path column determines, via constraint, that the right subtree of the initial tree will become the new tree of the next row. Then the 0 in the second row of the path column determines that the left subtree of T will become the new tree in the third row. Finally, the 1 in the third row determines that the right subtree will become the tree in the final row, which is the target subtree of S since we're out of digits.

Once the target subtree is found, we enter the multiplicity phase, where the results of the traversal phase are recorded with multiplicity. There is an mset variable such that during the multiplicity phase

$$\text{mset}' = \text{mset} \cdot (\beta - (p \cdot \text{subject}' + q \cdot \text{tarax}' + r \cdot \text{tree}'))$$

until the multiplicity variable mult, which is constant throughout the traversal phase, decrements to 0. Here, β, p, q, r are verifier-generated randomness. Note that the input subject and axis to the Eden 0 computation are included in the multiset, as well as the output. This is the same form as the query from the stack table: the stack table effectively asks, "Does running an Eden 0 computation with this subject and axis yield this output?" and the subtree access table responds, "Running an Eden 0 on this subject and axis yields this output."

The variable subject is similar to tarax in that it is part of the input to the computation, and it remains constant throughout each segment. Its value is constrained in the first row of a segment to be $a \cdot \text{tree-len} + b \cdot \text{tree-dyck} + c \cdot \text{tree-leaf}$ where $(\text{tree-len}, \text{tree-dyck}, \text{tree-leaf})$ is the disaggregation of the tree variable in Table 8, and a, b, c are verifier randomness.

7.1.4 Exponent Table

Recall the *cons* formulas (see (9) and (10)):

$$T_{d(\alpha_1)} = 0 \cdot \alpha_1^{2 \cdot \text{len}(L) + 2 \cdot \text{len}(R) - 3} + \alpha_1^{2 \cdot \text{len}(R) - 1} L_{d(\alpha_1)} + 1 \cdot \alpha_1^{2 \cdot \text{len}(R) - 2} + R_d(\alpha_1)$$

$$T_l(\alpha_2) = \alpha_2^{\text{len}(R)} L_l(\alpha_2) + R_l(\alpha_2)$$

In order to make use of these, we need to compute the exponents $\alpha_1^{2 \cdot \text{len}(R) - 1}$, $\alpha_1^{2 \cdot \text{len}(R) - 2}$, and $\alpha_2^{\text{len}(R)}$. Since $\text{len}(R)$ is a variable these are exponential functions which need to be computed in polynomial terms. This is the sole purpose of the exponent table. It then shares its results via multiset with other tables that perform *cons*.

It's easy to describe a naive design for computing exponents a^x of a value a (a could be either of the α_i above). Starting at a, we multiply, one step at a time, new factors of a into a variable exp, and increment the power variable pow. We then enter a secondary phase in which we accumulate (x, a^x) pairs into the multiset mset according to the value of the multiplicity mult. This multiset evolves as

$$\text{mset}' = \text{mset} \cdot (\beta - (a \cdot \text{pow} + b \cdot \text{exp}))$$

while the multiplicity decrements. Once mult reaches 0, pow increments and the accumulation process begins again.

A more involved optimization of this design that we implement is to initially compute a, a^2, a^4, ... , a^{256} by squaring so that one can increment the power by any integer in the range $[1, 512)$, and skip ranges of exponents which aren't needed and would take up unnecessary table space.

7.1.5 Pop Table

As discussed in Section 6.2, we model a stack as a dynamic polynomial $s(x)$, such that the current top of the stack is the constant term of the polynomial. We can push whatever we want to the stack without controversy, but repeated pops are problematic since we have to verify that what we're popping is a particular element that we previously pushed, and the number of intervening push/pop operations could be arbitrarily large. Due to the local nature of STARK constraints, the relevant information needed for stack operations occurring in disparate rows could potentially be unreachable. As discussed in the aforementioned section, the prover must produce a polynomial $\hat{s}(x)$ and a field element i such that $s(x) = x \cdot \hat{s}(x) + i$. The validity of this identity is equivalent to i being at the top of the stack s, and $\hat{s}$ being the state of the stack post-pop, and is checked via evaluation on a verifier-generated random value γ. The role of the pop table is to produce polynomial candidates for the various post-pop stack states $\hat{s}$ that arise.

At a naive level, all we have to do in the pop table is put the coefficients of each desired $\hat{s}$ polynomial in a column with a variable named item and accumulate via $\text{stack}' = \gamma \cdot \text{stack} + \text{item}'$. We can then place the final result in a multiset multiple times with a mult variable in a pattern that is now familiar from the noun and subtree access tables. When the multiplicity is exhausted, we switch to accumulating a new $\hat{s}$.

However, this would be inefficient. To see why, imagine two different stack states s_1 and s_2 which are equivalent except for their top elements. In the scheme described above, we'd have to repeat the shared portion which would increase the number of rows in the trace. Instead, we begin the accumulation from an $\hat{s}$ that has already been computed previously by taking it off the multiset and using it as the starting point, eliminating duplicate work. We also make sure to adjust the multiplicity for this $\hat{s}$ accordingly. In other words, the multiset now contains copies of stack states that will be used internally by the pop table for efficiency purposes, in addition to copies used by the stack table.

7.2 Input/Output Linking

As it stands, the system as described only proves the much more limited statement *there exists some program P and input I that produce some output.* A critical functionality that a zkVM must provide is the ability to prove statements of the form "*program P and input I produce output O*". Various techniques are employed in differing systems to achieve this functionality. In our case, we make a modification to the zk-STARK protocol described previously to allow for prover and verifier to generate constraints at proof-time according to a known protocol. This modification paves the way for generating input- or output-dependent constraints that can then be used to limit program inputs or program outputs to specific values.

The modification is as follows: firstly, before the base columns are interpolated and committed to, the prover sends the full computation and its result (i.e, subject, formula, and product) in plaintext over the proof stream, and the verifier accepts and stores the computation. Then, the prover and verifier continue the protocol as normal, with the verifier sending over random challenges to the prover. Critically, it is at this point that calculating ION Fingerprints is possible. Before proceeding, both prover and verifier augment the existing set of polynomial constraints with additional constraints. These constraints force the appropriate cells to contain the fingerprints of the input computation and output respectively. From there, the zk-STARK protocol proceeds normally. This addition allows the proofs generated from the system to be tied to the input and output as desired.

7.3 Extra Commitment Phase

As we've discussed, in order to use ION Fingerprints across the various tables of the Eden zkVM, these tables create multisets of the $(\text{len}, \text{dyck-felt}, \text{leaf-felt})$ triples that they use and check via a multiset argument that these coincide with ION Fingerprints constructed by the noun table.

However, there's a technicality to be aware of here. In a multiset argument for the equality of multisets $\{x_i\}$, $\{y_j\}$, the value β which is used to check whether $\prod(t - x_i) = \prod(t - y_j)$ must be *random*. In practice, this means that all values $\{x_i\}$, $\{y_j\}$ must be cryptographically committed to before the verifier generates the random value β. If the prover knew the value of β and was not committed to $\{x_i\}$, $\{y_j\}$, i.e. could change them after knowing β, it would be trivially easy to satisfy $\prod(\beta - x_i) = \prod(\beta - y_j)$ with unequal multisets $\{x_i\} \neq \{y_j\}$.

In the Eden zkVM, the prover needs randomness to construct the ION fingerprints, and this randomness is obtained from the verifier after an initial commitment to base columns. However, if the random value β used in multiset arguments as above is released at the *same* time as the random values used to construct the ION Fingerprints, there is a problem of the type indicated in the previous paragraph. This is because in every table except the noun table the fingerprint has not been committed to, and now the value β is known. This would give the prover leverage to forge inappropriate fingerprints that would pass the multiset argument.

Our solution is to add an extra commitment phase in the construction of our tables. First, base columns are committed to, and the verifier generates the randomness α_1 and α_2 used to construct ION Fingerprints. Then, columns that only depend on the values α_1 and α_2 are filled and committed to; these are the *primary extension columns*. The verifier then generates the remaining randomness, which includes the randomness used to compress tuples and make multiset and stack arguments. Only then does the prover fill the *secondary extension columns* to complete the construction of the tables.

7.4 Large-Noun Elision

Among various proof system performance metrics, proof size is often a primary concern. Because of the way that Eden organizes computation, *all* code that a given computation wishes to reference or make use of must be present in the subject, *including the standard library.* In combination with the input/output linking technique described above, this would necessitate the prover having to send large nouns in every proof, paying for the cost each time.

To avoid this, we make another modification to the zk-STARK protocol by taking advantage of the fact that certain large nouns such as the standard library are "well-known", i.e. are known *a priori* to both the prover and verifier. When sending the subject and formula over the proof stream, we elide well-known nouns by replacing each noun with a special tag that contains its hash instead. The verifier then scans the noun for these tags and replaces them with the appropriate noun that it has associated to the hash. The normal zk-STARK protocol proceeds from this point.

8 Security Analysis

8.1 Noun Collision

In this section we investigate the probability that two distinct nouns `a` and `b` encoded by the noun table have the same ION Fingerprint. Recall that the ION Fingerprint of a noun `a` is $\iota(a) = (\text{len}(a), d_a(\alpha_1), l_a(\alpha_2))$, where α_1, α_2 are random values from the field. Thus `a` and `b` have the same ION Fingerprint if and only if $\text{len}(a) = \text{len}(b)$, α_1 is a root of $d_a - d_b$, and α_2 is a root of $l_a - l_b$.

For each length l, and each unordered pair $\{a, b\}$ of distinct nouns with length l, there is a set of roots $r_d(\{a, b\})$ of $d_a - d_b$ in $\mathbb{F}$ and a set of roots $r_l(\{a, b\})$ of $l_a - l_b$ in $\mathbb{F}$, such that `a` and `b` have the same ION Fingerprint if and only if $\alpha_1 \in r_d(\{a, b\})$ *and* $\alpha_2 \in r_l(\{a, b\})$. Since the α_i are independently chosen, we can equivalently say that `a` and `b` have the same ION Fingerprint if and only if $(\alpha_1, \alpha_2) \in r_d(\{a, b\}) \times r_l(\{a, b\})$. Let S denote the set of pairs $\{a, b\}$ such that $a \neq b$ and $\text{len}(a) = \text{len}(b)$. Then, there is an ION Fingerprint collision between some pair $\{a, b\} \in S$ if and only if $(\alpha_1, \alpha_2) \in \cup_S\, r_d(\{a, b\}) \times r_l(\{a, b\})$.

Let N be the number of total nouns in the program; note that $N \leq 2^{32}$ since 2^{32} is the maximum allowable height of the noun table. For a length l, let N_l denote the number of nouns of length l, so that $N = \sum_l N_l$.

Note that for $\{a, b\} \in S$, either $r_d(\{a, b\}) = \mathbb{F}$ or $r_l(\{a, b\}) = \mathbb{F}$ but not both. The equality $r_d(\{a, b\}) = \mathbb{F}$ is equivalent to the case where `a` and `b` have the same tree shape but different leaf vectors, whereas $r_l(\{a, b\}) = \mathbb{F}$ is equivalent to the case where `a` and `b` are distinct nouns with the same leaf vectors on different tree shapes. If neither are equal to $\mathbb{F}$, then $|r_d(\{a, b\}) \times r_l(\{a, b\})| \leq 2(l - 1)l$. In the worst case, $|r_d(\{a, b\}) \times r_l(\{a, b\})| \leq 2(l - 1)|\mathbb{F}|$.

The worst case *a priori* upper estimate for $|\cup_S\, r_d(\{a, b\}) \times r_l(\{a, b\})|$ is thus

$$\sum_l \frac{1}{2} N_l (N_l - 1) \cdot 2(l - 1)|\mathbb{F}|$$

In a naive implementation of the noun table we have $N_l \leq \frac{2^{32}}{2(l-1)}$ since the dyck word of each noun is stacked vertically in the noun table. Using this naive esimate as a heuristic, we'd find

$$|\cup_S\, r_d(\{a, b\}) \times r_l(\{a, b\})| < N \cdot 2^{31} \cdot |\mathbb{F}| \leq 2^{63}|\mathbb{F}|$$

which puts the probability of sampling (α_1, α_2) from $\cup_S\, r_d(\{a, b\}) \times r_l(\{a, b\})$ as at most $2^{63-192} = 2^{-129}$.

If a performance-optimized version of the noun table were to cause the previous upper bound in the inequality $N_l \leq \frac{2^{32}}{2(l-1)}$ to swell by a factor of 2^{32}, the resulting inequality would then be $N_l \leq \frac{2^{64}}{2(l-1)}$. This would still only put the probability of sampling appropriate (α_1, α_2) at 2^{-97} in the worst case. Thus we conclude that the ION Fingerprint scheme within our zkVM is sufficiently secure against collision.

8.2 Cons & Inverse Cons Validation

Recall that the way we perform $\text{cons}(a, b)$ of two nouns in our zkVM is by obtaining the ION Fingerprints $\iota(a)$, $\iota(b)$ with randomness α_1, α_2 and then performing *cons* in this representation (which we denote by $[\iota(a)\ \iota(b)]$). This latter quantity is a triple of field elements that we always validate (using a multiset argument) to be equal to $\iota(c)$ for some noun `c`. Now, it is *a priori* possible that there is a `c` such that $\iota(c) = [\iota(a)\ \iota(b)]$ despite `c` $\neq$ `[a b]`. But if `[a b]` exists in the noun table, then we have the collision $\iota([a\ b]) = \iota(c)$, which we know is highly unlikely by Section 8.1. If `[a b]` is not already in the noun table, then the noun table augmented with this single noun would produce a collision. This can be demonstrated to be highly unlikely by making the same argument as in Section 8.1 with one extra noun and a noun table of height at most 2^{33} (since the extra noun can occupy at most 2^{32} rows).

Inverse *cons* is somewhat different; to perform inverse *cons* on a noun `a` in the zkVM, which is represented in the form $\iota(a)$, two triples of field elements are produced, which are validated to be equal to $\iota(b)$ and $\iota(c)$ for nouns `b` and `c`, and it is checked that $\iota(a) = [\iota(b)\ \iota(c)]$. The nondeterministic nature of the algorithm now gives the prover more latitude: they can survey all combinations of $\iota(b)$ and $\iota(c)$ such that $\mathrm{len}(a) = \mathrm{len}(b) + \mathrm{len}(c)$ and hope to find one such that `a` $\neq$ `[b c]` but $\iota(a) = [\iota(b)\ \iota(c)]$.

However, the prover is still heavily constrained, as they must commit to the particular multiplicities for each noun before they ever see any randomness. To illustrate, suppose that after the α_i are generated by the verifier, the prover identifies appropriate `a`, `b`, `c` such that `a` $\neq$ `[b c]` but $\iota(a) = [\iota(b)\ \iota(c)]$. Either the multiplicities of `b` and `c` were committed to based on an honest run of the program, or they were incorrect and committed to maliciously. In the former case, a malicious prover would be forced to repurpose an existing instance of `b` and `c` in the program execution in order to take advantage of $\iota(a) = [\iota(b)\ \iota(c)]$ to prove the false inverse *cons* relation `a` $\neq$ `[b c]`. This is because the prover cannot modify the multiplicities of `b` and `c` given their prior commitment. But now the original rows where `b` and `c` were used can no longer make reference to them, and have to be filled by some other nouns. This creates a cascade of difficulties for the prover who now has to reshuffle nouns into new roles and hope not to violate constraints. This adds robustness to our security analysis, as the variety of layers of constraints drastically reduces the prover's degrees of freedom. In the remaining case, the prover simply commits to false multiplicities with the expectation that `a` $\neq$ `[b c]` and $\iota(a) = [\iota(b)\ \iota(c)]$ would hold. But this involves predicting ahead of time that the verifier's random challenges α_1 and α_2 will be the roots of two given polynomials, which by Schwartz-Zippel is completely improbable.

Leaving such intuition to the side, what is the probability that `a` $\neq$ `[b c]` but $\iota(a) = [\iota(b)\ \iota(c)]$ holds? Firstly, based on Equations (5) and (6), there are two categories of ways in which `a` $\neq$ `[b c]` can occur: (1) $d_a \neq [d_b\ d_c]$ *and* $l_a \neq [l_b\ l_c]$, and; (2) exactly one of $d_a \neq [d_b\ d_c]$, $l_a \neq [l_b\ l_c]$ hold. We can easily argue that for the dishonest prover, looking for instances of `a` $\neq$ `[b c]` in category (1) where $\iota(a) = [\iota(b)\ \iota(c)]$ is futile. For a given `a`, there are at most 2^{64} pairs (`b`, `c`), so even if `a` had length 2^{32}, the probability of a single such instance is at most $\left(\frac{(2^{32})^3}{|\mathbb{F}|}\right)^2 \approx 2^{-192}$, by using Schwartz-Zippel once for each random value α_i. So, even if there were 2^{32} such inverse *cons* equations, the probability of a single false positive would be in the neighborhood of 2^{-160}. This is negligible, so for all practical purposes such cases can be ignored.

Thus, we turn to the odds of $\iota(a) = [\iota(b)\ \iota(c)]$ for instances of `a` $\neq$ `[b c]` in category (2). The odds of the prover seeing `a` $\neq$ `[b c]` because only one of $d_a = [d_b\ d_c]$ or $l_a = [l_b\ l_c]$ hold, *and* seeing $\iota(a) = [\iota(b)\ \iota(c)]$, depends on:

1. the number of inverse *cons* relations
2. for any given `a` to which the inverse *cons* will apply, the number of `b` and `c` nouns available in the program such that only one of $d_a = [d_b\ d_c]$ or $l_a = [l_b\ l_c]$ hold, and
3. the length of `a`, which determines the number of roots of $l_a - [l_b\ l_c]$ that can be targeted by α_2 (in the case where $d_a = [d_b\ d_c]$) and the number of roots of $d_a - [d_b\ d_c]$ that can be targeted by α_1 (in the case where $l_a = [l_b\ l_c]$).

In light of this, we will now provide a security argument using the naive implementation of the noun table. Let N be the total number of nouns in a given zkVM program, so $N \leq 2^{32}$. For any noun length l, let N_l be the number of nouns of length l. Of course, $N = \sum N_l$, but critically we also have

$$\sum N_l \cdot 2(l-1) \leq 2^{32} \tag{13}$$

since each noun of length l takes up a vertical space of $2(l-1)$. Now, for each noun `a` of length l and each decomposition $l = k + (l-k)$ (where $k = 1, ..., l-1$), there are at most N_k nouns `b` of length k and N_{l-k} nouns `c` of length $l-k$ available to construct a false equation `a` $\neq$ `[b c]` of category (2) such that $\iota(a) = [\iota(b)\ \iota(c)]$ holds. Thus, for each noun `a` of length l, there are at most P_l total ways to attempt a false construction, where P_l is the convolution-type sum $P_l = \sum_{k=1}^{l-1} N_k \cdot N_{l-k}$. Since there are N_l nouns of length l, and the number of roots of either the dyck *cons* or leaf *cons* equation cannot exceed $2(l-1)$, the maximum possible number of roots available that would allow a category (2) false *cons* is

$$\sum P_l \cdot N_l \cdot 2(l-1) \tag{14}$$

But $N_l \cdot 2(l-1) \leq 2^{32}$ by (13), which means that (14) is bounded by

$$2^{32} \sum P_l = 2^{32} \sum_l \sum_{i=1}^{l-1} N_i N_{l-i} \tag{15}$$

This double sum is further bounded by $\left(\sum N_l\right)^2 = N^2 \leq 2^{64}$ by expanding $\left(\sum N_l\right)^2$ and the positivity of the terms. Putting this together, we conclude that the maximum possible number of roots available that would allow a false *cons* in category (2) is at most 2^{96}, yielding a probability of at most $\frac{2^{96}}{|\mathbb{F}|} \approx 2^{-96}$ of such an event.

References

[Sze] A. Szepieniec. [Online]. Available: https://web.archive.org/web/20230630200925/https://github.com/aszepieniec/stark-brainfuck/issues/43

[KR08] Y. T. Kalai, and R. Raz, "Interactive pcp," in *Automata, Languages Program.*, Berlin, Heidelberg, 2008, pp. 536–547.

[Ham15] M. Hamburg, "Ed-448 Goldilocks, a new elliptic curve," 2015. [Online]. Available: https://eprint.iacr.org/2015/625.pdf

[BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive Oracle Proofs," 2016. [Online]. Available: https://eprint.iacr.org/2016/116 (Cryptology ePrint Archive, Paper 2016/116)

[Ben+18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," 2018. [Online]. Available: https://eprint.iacr.org/2018/046 (Cryptology ePrint Archive, Paper 2018/046)

[Gou21] A. P. Goucher, "An efficient prime for number-theoretic transforms," 2021. [Online]. Available: https://cp4space.hatsya.com/2021/09/01/an-efficient-prime-for-number-theoretic-transforms/

[GPR21] L. Goldberg, S. Papini, and M. Riabzev, "Cairo – a Turing-complete STARK-friendly CPU architecture," 2021. [Online]. Available: https://eprint.iacr.org/2021/1063.pdf (Cryptology ePrint Archive, Paper 2021/1063)

[Por22] T. Pornin, "EcGFp5: a Specialized Elliptic Curve," 2022. [Online]. Available: https://eprint.iacr.org/2022/274.pdf

[Noc23] "Nock Definition," 2023. [Online]. Available: https://developers.urbit.org/reference/nock/definition

[Blo23] R. Bloemen, "The Goldilocks Prime," 2023. [Online]. Available: https://xn--2-umb.com/22/goldilocks/

[DL78] DeMillo, and Lipton, "A probabilistic remark on algebraic program testing," 1978.

[Zip79] R. Zippel, "Probabilistic algorithms for sparse polynomials," 1979.

[Sch80] J. T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," 1980.

[FS87] A. Fiat, and A. Shamir, "How to prove yourself: practical solutions to identification and signature problems," in *Advances Cryptology --- CRYPTO' 86*, Berlin, Heidelberg, 1987, pp. 186–194.