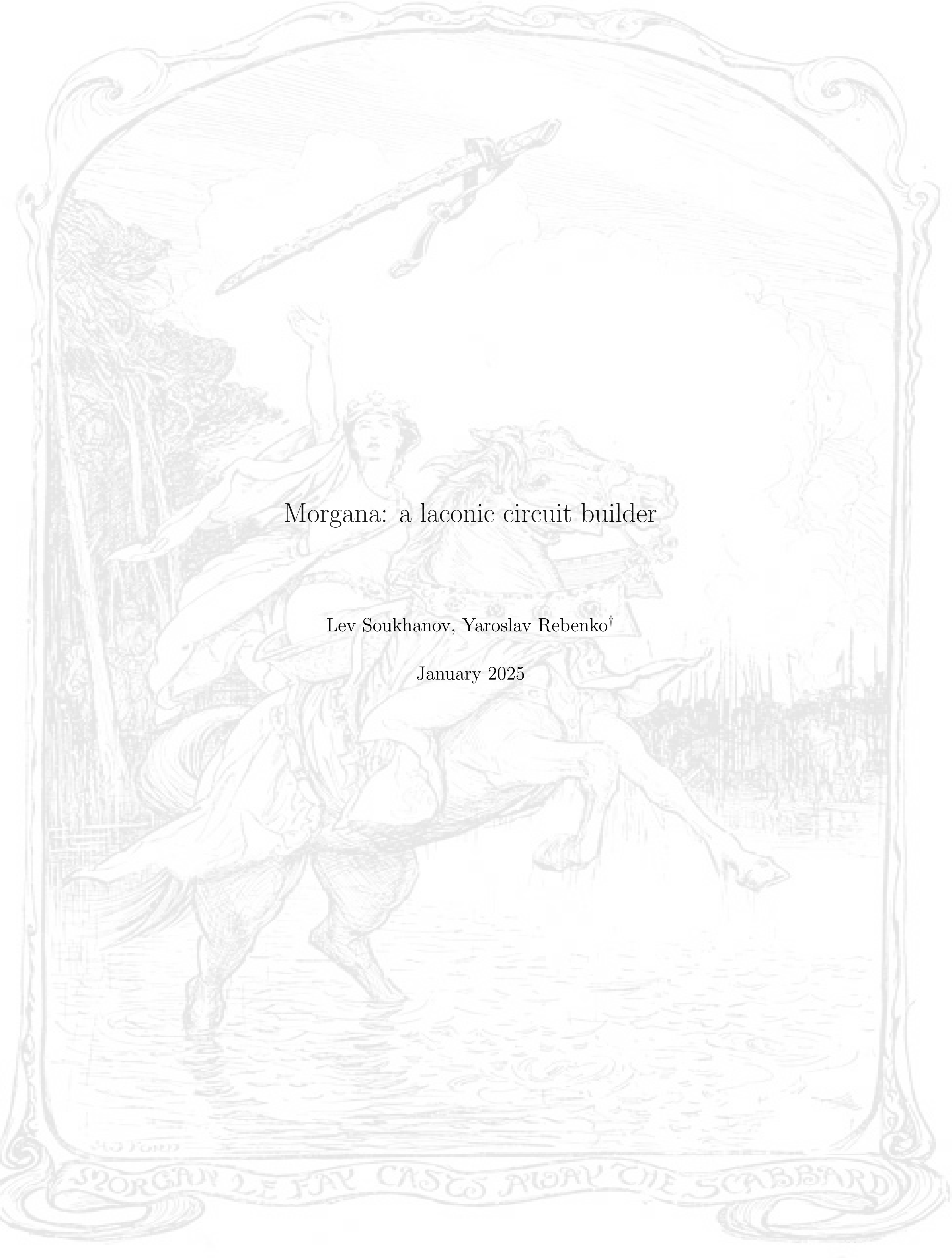

Morgana: a laconic circuit builder

Lev Soukhanov, Yaroslav Rebenko[†]

January 2025

Abstract

We construct a novel SNARK proof system, Morgana. The main property of our system is its small *circuit keys*, which are proportional in size to the *description* of the circuit, rather than to the number of constraints.

Previously, a common approach to this problem was to first construct a universal circuit (colloquially known as a zk-VM), and then simulate an application circuit within it. However, this approach introduces significant overhead.

Our system, on the other hand, results in a direct speedup compared to Spartan [19], the state-of-the-art SNARK for R1CS.

Additionally, small circuit keys enable the construction of zk-VMs *from* our system through a novel approach: first, outputting a commitment to the circuit key, and second, executing our circuit argument for this circuit key.

Contents

Introduction

State of the field

In recent years, we have seen impressive progress in the field of general-purpose SNARKs. It is difficult to do justice to this vast body of work in this brief summary, but the general outlook is as follows:

1. Trusted setup-specific approaches, such as Groth16 [10], are gradually being replaced by trusted setup-free approaches (or at least approaches with a universal trusted setup). Groth16 uses R1CS arithmetization.

2. The pivotal point in this transition was the introduction of PlonK [8]. Currently, dominant approaches, such as Halo2 [4] and STARKs [3] [18] [11], employ a very similar approach to arithmetization, known as "Plonkish."

3. Considerable effort has been dedicated to the development of efficient SNARK recursion primitives. In addition to the aforementioned Halo2 and STARKs (which are amenable to direct recursion), we highlight folding schemes [15] [5] [13] [14]. Importantly, all the folding schemes *obviate* the cost of linear combinations. As a result, they typically use either R1CS arithmetization or its generalization, CCS [20].

4. Recently, there has been a significant shift toward protocols that work with multivariate polynomials, allowing them to exploit the sumcheck protocol and, especially, the GKR [9] [22] protocol.

5. In applications, there has been a significant shift toward so-called zk-VMs (such as SP1, RISC-0, and Jolt [2]), universal circuits capable of simulating arbitrary programs. While the possibility of such an

† Ethereum Foundation, Privacy Scaling Explorations
0xdeadfae@gmail.com, yarebenko@gmail.com

approach has been understood for a long time, it is only recently that they have started to become feasible.

Circuit-based approaches and zk-VM approaches have a principal trade-off: while zk-VMs are slower due to simulation overhead, they enjoy **dynamic branching**. Indeed, in circuits, the expression

```
let y = if condition {option1} else {option2};
```

is interpreted as

```
let y = (1 - condition) * option1 + condition * option2;
```

which requires computing both branches before proceeding.

Somewhere in between zk-VMs and application-specific circuits are various approaches exploiting recursion to deal with non-uniformity. On the circuit side, Stackproofs [7] use non-uniform folding to simulate the call stack and Nebula [1] uses non-uniform folding to fetch the opcodes by selectively deactivating parts of the circuit. On the zk-VM side, various approaches to ”precompiles” have emerged—de facto separate circuits capable of executing heavy operations requiring acceleration. The job of the zk-VM is simply to move data to these accelerated circuits, which handle the heavy lifting.

Our approach

We have discovered this construction by accident. Our original entry point was the idea that folding schemes can largely ignore the complexity of the linear layer (and only deal with linear constraints in a decider phase), and we wanted to reproduce a similar behavior in a monolithic SNARK (for linear combinations that are batched enough times in a circuit). What we have discovered is not only that it is possible, it suddenly unlocks a far larger degree of non-uniformity than traditional folding and recursive schemes.

What we construct, in fact, is a circuit proof system with an extremely lightweight and *provable* circuit compiler. This unlocks the ability to run a two-stage protocol to achieve dynamic branching - in the first stage, a virtual machine runs the circuit compiler and outputs the verification key. The second stage checks the proof with respect to this verification key.

To illustrate how this can work, we once again consider the example with the ternary operator. Consider the following expression, with `A`, `B` representing two static circuits.

```
let y = if condition {A(x)} else {B(x)};
```

a VM then will non-deterministically output one of the following two circuits in its place (note that inputs are known to the prover, so it can choose the appropriate branch):

```
assert!(condition);
let y = A(x);
```

or

```
assert!(!condition);
let y = B(x);
```

Circuit builder

Let us consider how circuit builders work. We will restrict our discussion to some large class of proof systems:

1. We only consider proof systems without circuit-specific trusted setup. Universal trusted setup is tolerated.
2. We assume that it uses the PCS + PIOP pattern. We fix some polynomial commitment scheme (or, maybe, a collection of commitment schemes) over our base field $\mathbb{F}$.
3. We assume that the verifying key of the circuit is a commitment to the circuit key, using our polynomial commitment scheme. We elaborate on this further:

First of all, we refer to **circuit key** when we mean circuit-specific information that is used for the proof. Typically, people consider a **proving key**, which consists of a **circuit key**, and a **commitment key**. The latter may or may not be present (for example, for FRI it is not present and for KZG it is a crucially important *universal* setup data).

Our (rather pedantic) requirement states that the **verifying key** is in fact a commitment (using our commitment scheme) to the **circuit key**. The reason why we insist on this is that formally a compressed high-level definition of a circuit would already qualify as a "circuit key" as long as the compiler is deterministic. We insist that the circuit key means the fully unrolled form - for example, in PlonK it would be values of all fixed columns, i.e. selector columns and columns encoding copy constraint permutation.

Operationally, our plan is to use an auxiliary VM proof system to output a circuit key and then use it's commitment for the circuit proof system. This, of course, requires property 3.

Almost all proof systems satisfying the 1st property that we know of satisfy the remaining properties.

Most circuit builders, then, work roughly as follows:

1. The circuit builder starts with a high-level description of a constraint system. Eventually, it typically constructs something similar to our definition of a **modular constraint system**, an intermediate representation which involves calls to a subcomponents, which are, themselves, modular constraint systems.

2. It then compiles this representation, obtaining a circuit key. In case of PlonK, this step is highly non-trivial, involving allocation of constraints, and computation of copy constraint permutation matrices. For R1CS / CCS, this is essentially just the flattening of an ordered graph of modular components.

We note that this 1-st stage representation typically has a much smaller size - the size cost of invoking a new component is $O(|\text{input}|)$ of this component. Our goal is to construct a proof system for which the 2-nd stage is essentially trivial.

V-SPARK

We construct a novel version of the SPARK argument, introduced in Spartan [19], which we call V-SPARK (”vectorized spark”). Using it as a drop-in replacement for SPARK in super-Spartan, we get a proof system with the desired property - a circuit key proportional not to the CCS constraint system size, but to the size of the modular description.

Dynamism

The most important application is that this approach enables far cheaper dynamic branching. Rather than building a zk-VM that *simulates* the circuit, it is now possible to build a zk-VM that *outputs* the modular description of the circuit and then proves it using the static part of the argument.

In a ”precompile” paradigm, this can be seen as essentially the freedom to make as many precompiles as desired, at no extra cost.

We avoid many preliminaries in this paper and assume familiarity with sumcheck-based protocols. We refer the reader to the excellent Spartan [19] and Lasso [21] papers for the necessary background on the use cases of the sumcheck protocol, the GKR protocol, and SPARK. We also refer to Ceno [16] as an example of an alternative approach to a similar problem set.

Acknowledgments

The name of our proof system "Morgana" comes from Morgan Le Fay, magician competitor of Merlin. By coincidence, it is also the internal name of our research group.

We thank Srinath Setty, John Chandler Burnham, Mamy Ratsimbazafy, Zhenfei Zhang and Misha Komarov for their initial review of our method, and Kevaundray Wedderburn, Chance Hudson, and Carlos Perez for their useful comments and suggestions.

1. V-SPARK Protocol

1.1 Preliminaries

1.1.1 Array notation

Let us begin by introducing some notation. We work in a field $\mathbb{F}$ such that $|\mathbb{F}| \geq\sim 2^{\lambda}$, where λ is a security parameter. By default, the values that we use are either field elements or small non-negative integer numbers (and we assume that small integers are embedded in $\mathbb{F}$).

We use Rust-like notation for the range:

$$[a..b] = \{x \in \mathbb{N} \mid a \leq x < b\}$$

The chunk of an array s is denoted as $s[a..b]$ (and one-sided chunks are denoted $s[a..]$ and $s[..b]$, respectively).

As we need to frequently deal with recursively defined structures, we will also use lists (note that this notation defines the set of all lists with values in T):

$$\mathrm{List}(T) = \{(k \in \mathbb{N}, f : [0..k] \to T)\}$$

For $l \in \mathrm{List}(T)$, we will denote it's i-th element l_i or $l[i]$ (if it exists). For $l \in \mathrm{List}(A \times B)$ we denote $\mathrm{unzip}(l) = (l_a, l_b)$ the pair of lists defined as $(l_a[i], l_b[i]) = l[i]$, and zip the inverse operation (valid only if the length of the lists coincides).

For two sets A, B their disjoint union (analogue of programming enums) is denoted $A \coprod_{x,y} B$. The subscripts x, y, if present, denote the names of the canonical embeddings (constructors):

$$i_x : A \to A \coprod_{x,y} B$$

$$i_y : B \to A \coprod_{x,y} B$$

1.1.2 Polynomials

Unless otherwise specified, we will work with multivariate polynomials.

Since multilinear polynomials are defined by their values in the boolean hypercube $\mathbb{B}^n$, it is convenient to use the following notation: for a multilinear polynomial P and a number i, we will denote

$$P[i] = P(i_0, ..., i_{n-1})$$

where

$$(i_0, ..., i_{n-1}) \in \mathbb{B}^n, \sum 2^k i_k = i$$

As a slightly unconventional notation, we declare that standalone $[i]$ denotes the decomposition $(i_0, ..., i_{n-1})$. This is because that way conveniently $P([i]) = P[i] = P(i_0, ..., i_{n-1})$.

We also denote by $\mathrm{eq}(x; y)$ a multilinear polynomial in two sets of variables that represents a diagonal matrix: $\mathrm{eq}[i][i] = 1$, and $\mathrm{eq}[i][j] = 0$ for $i \neq j$. We will occasionally use the shorthand notation $\mathrm{eq}_r(x) = \mathrm{eq}(x; r)$ when the variable $y = r$ is fixed.

The following closed-form expression is well known:

$$\mathrm{eq}(x; y) = \prod_{i \in [0..n]} (x_i y_i + (1 - x_i)(1 - y_i))$$

A simple implication of this formula is the following **tensor decomposition property**:

$$\mathrm{eq}(x; y) = \mathrm{eq}(x[..k]; y[..k])\, \mathrm{eq}(x[k..]; y[k..])$$

Definition 1. *The* ***multilinearization*** *of a table P is a multilinear polynomial P' such that $P[i] = P'[i]$ for all i.*

Multilinearization can be computed as follows:

$$P'(y) = \sum_{x \in \mathbb{B}^n} P(x)\, \mathrm{eq}(x; y)$$

We will typically skip the multilinearization notation and denote a multilinearization of an array P with the same letter. We hope this does not cause any confusion.

1.1.3 Pullback and pushforward

Next, we introduce two operations on arrays. Assume that we are given two sets, identified with $[0..n]$ and $[0..m]$.

Let $b : [0..m] \to \mathbb{F}$ (represented by an array of length m), and let $f : [0..n] \to [0..m]$ (represented by an array of length n with values in $[0..m]$).

Definition 2. *The* ***pullback*** *of b along f is:*

$$(f^*b)[i] = b[f[i]]$$

In the Lasso paper, the same concept is referred to as an indexed lookup.

Now, assume f is as defined above and a is an array of size n.

Definition 3. *The* ***pushforward*** *of a along f is:*

$$(f_*a)[i] = \sum_{j|f[j]=i} a[j]$$

This concept is somewhat obscured in SPARK-related papers, but really it describes a sparse array using a set of (value, index) pairs, with the added feature that indices can repeat and values for repeating indices are summed together.

Proposition 1. *The following well-known observation establishes the duality between pullback and pushforward:*

$$\langle a, f^*b \rangle = \langle f_*a, b \rangle$$

Here, $\langle \cdot, \cdot \rangle$ denotes the inner product.

1.2 SPARK

Assume that we are in the following setting:

- We are given a multilinear polynomial P in n variables and a multilinear polynomial f in n variables, with values living in $0 \dots 2^m \subset \mathbb{F}$.
- Both of these are committed (or, in an interactive setting, the oracles to these polynomials are sent to the verifier), and the prover tries to convince the verifier of the claim of the form:

$$(f_*P)(r) = c$$

The SPARK protocol family follows the following blueprint:

$$c = (f_* P)(r) = \langle f_* P, \mathrm{eq}_r \rangle$$

$$\langle f_* P, \mathrm{eq}_r \rangle = \langle P, f^* \mathrm{eq}_r \rangle$$

Therefore, it is sufficient to check the sum:

$$\sum_{x \in \mathbb{B}^n} P(x)(f^* \mathrm{eq}_r)(x) = c$$

The methods for computing this sum diverge:

1. The prover can commit to $f^* \mathrm{eq}_r$ and validate that it is an indexed lookup using any off-the-shelf lookup argument. While this method is conceptually the simplest, it is not efficient for normal SPARK.

2. Alternatively, one can commit to the bit decomposition of f and compute $f^* \mathrm{eq}_r$ directly using a closed-form expression. This method is more efficient when $2^n \ll 2^m$.

3. As a middle-ground (and the most practically used method), one can split f into multiple limbs of the form:

   $$f = f_0 + 2^{\alpha} f_1 + \cdots + 2^{(q-1)\alpha} f_{q-1}$$

   where $q = \lfloor m/\alpha \rfloor$. Then, one observes that:

   $$f^* \mathrm{eq}_r = \prod_{j \in 0..q} f_j^* \mathrm{eq}_{r_j}$$

   where $r_j = r[\alpha j \ldots \alpha(j+1)]$, due to the tensor decomposition property of eq_r.

One of the reasons why the third method is efficient is that, while the prover needs to perform more indexed lookups for all the pullbacks, the set of values taken by $f_j^* \mathrm{eq}_{r_j}$ is small (of size 2^q). This is crucial for elliptic curve-based commitment schemes. Curiously, FRI-based commitment schemes do not benefit from this speedup and are forced to stop at roughly $2^q \sim 2^n$ for optimal efficiency.

Various minor optimizations are possible (for example, the sumcheck for the computation of the sum can be run in parallel with the lookup argument), but this is not important for our high-level exposition.

Virtual machine description

One can interpret a SPARK as an execution of a following (very simple) VM.

The program is a sequence of pairs `(ADDR, VALUE)`. It takes as an input the evaluation point r, initializes a single register `RET = 0` and executes this sequence using the following state transition:

```
RET += EQ(ADDR, r) * VALUE
```

Obviously, this interpretation is rather trivial (and in practice, optimizations described above are employed to save the costs), but it makes sense to use it as a specification of a necessary functionality of SPARK.

1.3 V-SPARK

1.3.1 Admissible sets

The idea of **V-SPARK** is to add caching to the SPARK. For this, we need to define a special class of embeddings that are friendly to evaluation properties of multilinear polynomials.

Definition 4. ***Admissible segment*** *is a range* $[0..2^n]$*. The set of admissible segments is denoted* Adm*.*

This is just a notion of an evaluation domain - the n-dimensional hypercube will always be identified with this admissible segment. The space $\mathbb{F}^s$ of functions on the admissible segment (i.e. vectors of length 2^n) correspond to multilinear polynomials in n variables through multilinearization.

Definition 5. ***Admissible embedding*** *is an embedding of admissible segments* $f^{(a)} : [0..2^k] \hookrightarrow [0..2^n]$ *defined as* $f^{(a)} : x \mapsto x + a \cdot 2^k$*. It's image (a segment* $[a \cdot 2^k..(a+1) \cdot 2^k]$*) will be called an admissible sub-segment, or an admissible subset.*

For $s', s \in \mathrm{Adm}$, denote $\mathrm{Emb}(s', s)$ the set of admissible embeddings from s' to s.

Lemma 1. *(Properties of admissible embeddings):*

1. ***(Composition)*** *For a pair of admissible embeddings*

$$s \xhookrightarrow{f} s' \xhookrightarrow{f'} s''$$

their composition $s \xhookrightarrow{f' \circ f} s''$ *is admissible.*

2. ***(Evaluation)*** *For an admissible embedding* $f^{(a)} : [0..2^k] \hookrightarrow [0..2^n]$*, a polynomial* $Q(x_0, ...x_{k-1})$ *and a point* $r = (r_0, ...r_{n-1})$ *it holds that*

$$(f_*^{(a)}(Q))(r) = Q(r[..k]) \cdot \mathrm{eq}([a], r[k..])$$

Proof. The first property is obvious - if $s = [0..2^k], s' = [0..2^{k'}], s'' = [0..2^{k''}]$ for $k \leq k' \leq k''$, and f, f' are shifts $f^{(a)}, f^{(a')}$ respectively, the composition $f' \circ f$ performs a total shift by $a \cdot 2^k + a' \cdot 2^{k'} = 2^k(a + a' \cdot 2^{k'-k})$. Therefore, it is an admissible embedding $f^{(a+a'2^{k'-k})}$.

For the second property observe that $f^{(a)}$ can be defined on a hypercube as a linear embedding $(x_0, ..., x_{k-1}) \mapsto (x_0, ..., x_{k-1}, [a])$. We know how the eq polynomial behaves w.r.t. these embeddings.

$$\begin{gathered}(f_*^{(a)}(Q))(r) = \langle f_*^{(a)}(Q), \mathrm{eq}_r \rangle = \\ = \langle Q, (f^{(a)})^* \, \mathrm{eq}_r \rangle = \langle Q, \mathrm{eq}_{r[..k]} \rangle \, \mathrm{eq}_{r[k..]}([a]) \\ = Q(r[..k]) \, \mathrm{eq}(r[k..], [a])\end{gathered}$$

□

The lemma 1 lists all the required properties of admissible embeddings. In the later parts of the article, we will discuss alternative *admissibility structures* - for different kinds of polynomials, these will be bases satisfying the embedding property, and a form of evaluation property (which allows one to evaluate the direct image in a point using a closed formula).

We avoid axiomatizing the concept of admissibility structure, but will use it as an informal concept - a set of bases satisfying a version of the lemma 1. Many of our results (formulated in terms of composition and evaluation) then will easily transition to these other settings.

1.3.2 Sparse arrays

Definition 6. *For an admissible segment* s : Adm*, we define sparse arrays:*

$$\mathrm{Sp}(s) = \mathrm{List}(s \times \mathbb{F})$$

This is just a normal definition of a sparse array that we have always used - a sequence of (address, value) pairs. Alternatively, for a sparse array $u \in \mathrm{Sp}(s)$, one can define a pair of lists $f \in \mathrm{List}(s), a \in \mathrm{List}(\mathbb{F})$ via unzipping: $(f_i, a_i) = u_i$.

The dense form of an array u is defined as

$$\mathrm{df}(u) = f_* a$$

.

1.3.3 Cached-sparse arrays

We generalize the sparse arrays to allow recursive definition on admissible subsegments.

Definition 7. *For $s = [0..2^n] \in \mathrm{Adm}$, we define the set of cached-sparse arrays using the following recursive rule:*

$$\mathrm{CSp}(s) = \mathrm{Sp}(s) \coprod_{(sp,rec)} \mathrm{List}(\{(s', f, u) | s' \in \mathrm{Adm}, f \in \mathrm{Emb}(s', s), u \in \mathrm{CSp}(s')\})$$

Or, in plain language, a cached-sparse array hosted on s is either

1. A sparse array hosted on s (base case).
2. A list of triples (s', f, u) where $f : s' \hookrightarrow s$ is an admissible embedding and u is a cached-sparse array hosted on s'.

The dense form of the cached-sparse array is also defined recursively:

1. For the base case $u = i_{sp}(c)$, $\mathrm{df}(u) = \mathrm{df}(c)$.
2. For the recursive case $u = i_{rec}([(s'_0, f_0, u_0), (s'_1, f_1, u_1), ...])$, we define

$$\mathrm{df}(u) = \sum_i (f_i)_* (\mathrm{df}(u_i))$$

Lemma 2. *The cached-sparse polynomial u can be evaluated in a point r using the following recursive algorithm:*

1. For the base case $u = i_{sp}(c)$, the normal SPARK applies:

$$(\mathrm{df}(u))(r) = (\mathrm{df}(c))(r) =$$

(denote $\mathrm{unzip}(c) = (f, a)$)

$$= (f_* a)(r) = \sum_i a_i \cdot (f^* \, \mathrm{eq}_r)_i$$

2. For the recursive case $u = i_{rec}([..., (s'_i, f_i, u_i), ...])$

$$(\mathrm{df}(u))(r) = \sum_i ((f_i)_* u_i)(r) =$$

and by evaluation property from lemma 1 we get

$$\sum_i (u_i(r[..k]) \cdot \mathrm{eq}(r[k..], [a]) \text{ where } s'_i = [0..2^k], f_i = f^{(a)})$$

1.3.4 Directed graph interpretation

Consider an array of admissible segments $s \in \text{List}(\text{Adm})$ (called "shapes"), and an array u of cached-sparse arrays $u[i] \in \text{CSp}(s[i])$, such that each recursive $u[i]$ only involves $u[j]$-s for $j < i$. They form a topologically ordered directed graph (with multiple edges, but without loops) in the following way:

1. The vertices of the graph are indices i.
2. For a vertex i corresponding to the recursive $u[i] = i_{rec}(\ell)$, and $j < i$ we define the list of edges $e(j,i) \in \text{List}(\text{Emb}(s[j], s[i]))$ as all f-s that map $u[j]$ to $u[i]$:

$$e(j,i) = [f \text{ for } (s_{\_}, f, u_{\_}) \text{ in } \ell \text{ if } u_{\_} = u[j]]$$

It is a nicer interpretation from a computational standpoint because it reflects the actual concept of caching.

1.3.5 Sample implementation

Consider an array u of descriptions. Each description $u[i]$ has a field k representing the corresponding admissible segment $[0..2^k]$ (i.e. k is number of variables in the corresponding polynomial), and the field entries, which is either Sp or Rec. Sparse entries are lists of $(\text{addr}, \text{val})$ pairs, while recursive descriptions are lists of (id, a) pairs, where $\text{id} < i$ is the id of one of the previously constructed polynomials, and a defines the embedding $f^{(a)}$.

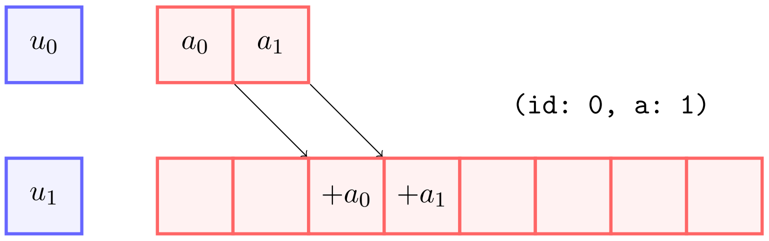

Figure 1.1: Recursive entry adds an already constructed array to an admissible segment

```
fn evaluate_cached_sparse<F: Field>(
    r: &[F], // evaluation point
    u: CachedSparseDescription<F>,
        // as defined above
) -> F {
    let n = u.len();
    let mut evals = vec![F::zero(); n];

    for i in 0..n {
        match &u[i].entry {
            // evaluation point of u[i]
            let r_ctx = &r[..u[i].k];

            Sp(arr) => {
                for (addr, val) in arr {
                    evals[i] += eq(r, addr) * val
                }
            }
            Rec(arr) => {
                for (id, a) in arr {
                    let r_hi = r_ctx[u[id].k..]
                    evals[i] += evals[id] * eq(r_hi, a)
                }
            }
        }
    }
return evals[n-1];
}
```

Definition 8. *A SNARK implementing this function is called V-SPARK protocol.*

The details of the SNARK itself may vary. The most efficient approach likely involves a write-once memory for values of `evals`, but we do not go into these details here - an implementation using a fully dynamic RAM is enough for our purposes.

1.4 V-SPARK modifications

In our actual application, we need various slight modifications of V-SPARK.

1.4.1 Matrix version

In the Spartan protocol, SPARK is applied to the polynomials representing structure matrices that define the R1CS circuit. We therefore need a version

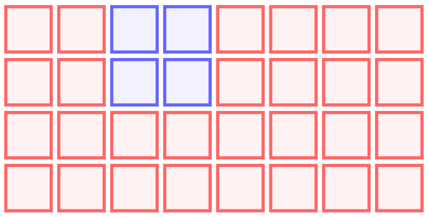

Figure 1.2: An admissible submatrix is a product of two admissible subsegments

of V-SPARK that is applicable to admissible submatrices (i.e., products of two admissible subsegments).

All the definitions can be tweaked to accommodate these changes. Importantly, the evaluation formula from the lemma 1 now states that for the admissible embedding

$$f^{(a,a')} : [0..2^k] \times [0..2^{k'}] \hookrightarrow [0..2^n] \times [0..2^{n'}]$$

a matrix-representing polynomial $P(x_0, ..., x_{k-1}; y_0, ..., y_{k'-1})$, and an evaluation point $(r; r') = (r_0, ..., r_{n-1}; r_0, ..., r_{n'-1})$ we have

$$(f_*^{(a,a')} P)(r; r') = P(r[..k]; r'[..k']) \cdot \mathrm{eq}(r[k..], [a]) \cdot \mathrm{eq}(r'[k'..], [a'])$$

The rest of the definitions and formulas are modified in an obvious way to accomodate this.

Definition 9. *We state the required notation for **cached-sparse matrix graph description** explicitly, as we will need it later:*

1. *Vertices are indices i. Each vertex has an associated matrix shape $s[i] \times s'[i]$.*

2. *Some vertices correspond to sparse descriptions. For these, we have an associated sparse matrix.*

3. *For a recursive i, and $j < i$, there is a list of embeddings $e(i, j) \in \mathrm{Emb}(s[j] \times s'[j], s[i] \times s'[i])$.*

1.4.2 Vector-valued arrays

Sometimes, it is useful to consider *vector-valued* arrays. This is useful if multiple arrays are being constructed using the same combinatorial description.

An important practical example for which this happens are the R1CS matrices (A, B, C).

2. Modular Constraint Systems

2.1 MCS definitions

Now, our goal is to construct a circuit compiler that outputs cached sparse matrix descriptions as circuit keys. Obviously, there is no chance of doing this efficiently for a general circuit. Instead, we want to exploit modularity, which naturally exists in most realistic circuits constructed by programmers (with the notable exception of neural networks and other poorly structured circuits).

Definition 10. *CCS shape is a collection of admissible segments*

$$s = (s_{source}, s_0, ..., s_{k-1})$$

.

Definition 11. *A Generalized CCS (customizable constraint system) of shape s is a tuple $(M_0, ..., M_{k-1})$ of matrices, with matrix M_i defined on $s_{source} \times s_i$, and an additional set of non-linear conditions, which is kept opaque in this definition.*

The witness W is an element of $\mathbb{F}^{s_{source}}$. The results of application $T_i = M_i(W) \in \mathbb{F}^{s_i}$ are called virtual targets. The T_i-s are subject to some non-linear conditions.

The witness W satisfies the CCS if the T_i-s satisfy these non-linear conditions.

(The original CCS definition given by Srinath Setty says that T_i-s all have the same length, and satisfy a single non-linear equation. We do not insist on this — and, practically, a similarly generalized version of CCS is already used in one of the newer folding papers [14].)

A particular simple case of this definition is R1CS. In this case, we only have three matrices, and the virtual targets are subject to non-linear relation $M_0(W) \cdot M_1(W) - M_2(W) = 0$.

For our purposes, we need a more refined notion of **components**, which can involve multiple admissible segments that are *to be allocated* in either W or virtual targets.

We denote Tag the set of tags - the tag "source", corresponding to the witness, and indices of virtual targets.

Definition 12. *Component shapes* CSh *are defined as lists of tagged admissible segments:*

$$\mathrm{CSh} = \mathrm{List}(\mathrm{Adm} \times \mathrm{Tag})$$

Note that in contrast to CCS shapes, in the component shape the tags can repeat - for example, we can have multiple segments tagged "source". This reflects the idea that the component insists that each of these witness chunks is allocated contiguously, but does not care if the chunks themselves are in arbitrary positions.

Definition 13. *For a pair of shapes $v', v \in \mathrm{CSh}$, we denote $\mathrm{Emb}(v', v)$ the set of tag-preserving admissible embeddings. I.e., the embedding from v' to v is a mapping that takes $(s', t') \in v'$ and returns $((s, t), f) \mid (s, t) \in v, f \in \mathrm{Emb}(s', s)$ such that $t = t'$.*

We will use it to define allocation, or "gluing data" - i.e. describe the way that embeds a particular component into the larger one.

Definition 14. *For the shape $v \in \mathrm{CSh}$, we define a **primitive component** of shape v. Denote $(s_i, t_i) = v_i$ segments and tags, respectively. Then, the primitive component is a collection of sparse matrices from source to target segments:*

$$(M_{ij} \in \mathrm{Sp}(s_i \times s_j) \mid t_i = \mathit{source}, t_j \neq \mathit{source})$$

The set of primitive components of shape v is denoted $\mathrm{PrComp}(v)$.

Definition 15. *The **recursive components** of the shape v are defined recursively:*

$$\mathrm{RecComp}(v) =$$

$$\mathrm{PrComp}(v) \coprod_{pr,rec} \mathrm{List}(\{(v', g, u) | v' \in \mathrm{CSh}, g \in \mathrm{Emb}(v', v), u \in \mathrm{PrComp}(v')\})$$

We call g gluing data.

Definition 16. *A **modular constraint system** is a topologically ordered graph of recursive components $(u_0, ..., u_{n-1})$ - i.e. definition of each component only involves previously defined components u_j for $j < i$.*

Additionally the component u_{n-1} is assumed to have the CCS shape.

Similar to previous cases, we can interpret it as a graph of shapes and embeddings:

1. Each vertex i has an associated shape $v[i] \in \mathrm{CSh}$
2. Sparse vertex has an associated primitive component.
3. Recursive vertex i is connected with $j < i$ by a list of gluing mappings $g(j, i) \in \mathrm{List}(\mathrm{Emb}(v[j], v[i]))$.

Every modular constraint system can be interpreted as the cached-sparse matrix system, by replacing each vertex i with the list of products

$$\ell[i] = [s \times s' \text{ for } (s, t), (s', t') \text{ in } v[i] \text{ if } s = \text{source}, s' \neq \text{source}]$$

. Then, each $g(j, i)$ can be interpreted as a bipartite graph of admissible matrix embeddings between $\ell[j]$ and $\ell[i]$ - recall that each gluing is an embedding of $v[j] \hookrightarrow v[i]$. The corresponding lists of embeddings of products are united by all $g \in g(j, i)$.

Proposition 2. *For an MCS u, the CCS matrices corresponding to u_{n-1} can be computed using V-SPARK protocol on the graph defined above.*

Spartan protocol for MCS (with V-SPARK used to compute evaluations of structure matrices instead of SPARK) is a proof system with the circuit key proportional in size to the modular description of a constraint system.

2.2 Approaches to an allocator

In theory, the previous description is already enough to write circuits. For example, if one creates a modular component, representing some re-usable functionality, they can then allocate it manually multiple times and construct a modular constraint system. In this process, the target allocations, likely, never intersect (because targets represent constraints, if they intersect for multiple components it would mean that constraints got added to each other). Witness allocations can and sometimes should intersect, to facilitate I/O between the components.

However, it is extremely inconvenient - not only the programmer will need to allocate the witness manually, but the alignment requirements are also much more difficult than in normal programming! De facto, only the structures of size 2^k are allowed.

To understand more about different tradeoffs that can arise during the allocation, consider the following explicit example:

2.2.1 Circom example

As an example, we will use Circom[12], a language for R1CS circuits (the syntax is very intuitive, and in our examples, knowledge of the full language is not really necessary). As we are working with R1CS circuits, we have three virtual targets: L, R, and O, bound by the equation $L[i] \cdot R[i] = O[i]$. Circom has the constraint operator `===`, which represents the R1CS constraint.

Here is an example circuit:

```
/// this component checks that a^(2^20) = b^(2^20)
component Component1() {
    signal input a;
    signal input b;

    signal a_powers[20];
    signal b_powers[20];

    a_powers[0] <== a * a;
    b_powers[0] <== b * b;

    for(var i = 0; i < 19; i++) {
        a_powers[i + 1] <== a_powers[i] * a_powers[i];
        b_powers[i + 1] <== b_powers[i] * b_powers[i];
    }

    a_powers[19] === b_powers[19]; // constrains lhs = rhs
}
/// This component does it for each a[i], b[i]
component Component2() {
    signal input a[64];
    signal input b[64];

    // this cycle is statically unrolled at compilation time
    for(i=0; i<64; i++) {
        var tmp = Component1();
        tmp.a <== a[i];
        tmp.b <== b[i];
    }

}
```

Consider the witness of `Component1`. In total, we need to allocate 42 elements to the witness. The way they are allocated is irrelevant because we consider the `Component1` to be primitive. It also needs to allocate 40 elements in each of the L, R, and O targets to accommodate constraints (in R1CS, these three targets are always used together).

`Component2` needs to allocate 64 instances of `Component1`. It also needs to separately allocate its inputs (as they are contiguous arrays) and spend 128 constraints to copy the data between instances of `Component1` and the inputs.

The question here is: what strategy should be used for the allocation of `Component1`? There are a few possible options (we will speak only about the witness here; analogous problems also exist for the targets):

1. Allocate 64 elements to the witness, wasting $64 - 42 = 22$ elements.
2. Allocate $42 = 32 + 8 + 2$ elements to the witness in 3 separate segments.
3. Do something in between.

2.2.2 Tradeoffs

First strategy is wasteful - because each level of sub-component in a circuit leads to the wasted witness. These losses accumulate multiplicatively, leading to inefficient allocation after just few layers of indirection.

The second strategy is much better in terms of witness size; moreover, it is actually optimal:

Lemma 3. *Let $n = n_1 + ... + n_k$. Denote s_i the admissible segments of such lengths that $\sum\limits_i l(s_i) = n$ (i.e., a binary decomposition). Analogously, define $s_{i,j}$ to be $\underset{i}{l}(s_{i,j}) = n_j$.*

Then, it is possible to tightly pack the disjoint union of $s_{i,j}$ in the disjoint union of s_i.

Proof. The following greedy algorithm then does the job:

1. If there exists an odd segment $l(s_i) = 1$, then there exists some $l(s_{i,j}) = 1$. Pack $s_{i,j}$ in s_i, it removes them from the process.
2. If there is no odd segment s_i, but there is $l(s_{i,j}) = 1$, there exists another odd segment $s_{i',j'}$. Glue them together and treat them as a single segment (i.e. they will always be allocated contiguously).

3. If all segments have even length, rescale every segment 1/2 times and continue.

$\square$

Therefore, the second strategy allocates everything without overhead. There is, however, another issue - the total amount of matrices involved in a definition of the component blows up quadratically (as we allocate $\sim$ log(size) segments for both the source and each of the targets). We did not perform the exact estimates, but it looks like a potentially significant overhead for a component call.

There are also other reasons to desire simplicity (even if this strategy works). For example, in a dynamic circuit builder (which is a desired target of Morgana), the circuit is not known beforehand. That means that any allocation strategy will need to be executed during proving time — a serious argument against using "smarter" approaches to allocation.

In the next chapter, we explain an approach that solves this issue altogether (at the cost of a marginally larger prover). We believe it is also likely the simplest approach from a dynamic standpoint (i.e., it minimizes the amount of work done by an allocator).

3. Alternative Admissibility Structures

In this chapter, we adapt the techniques from V-SPARK to a hybrid univariate - multivariate setting.

Specifically, univariate polynomials in a **coefficient** basis are very desirable because they can be shifted by k using multiplication by x^k. This means that admissible segments in this case are **any** contiguous segments:

3.1 Admissibility in monomial basis is trivial

We change the notation slightly - $P[i]$ now means i-th coefficient of a polynomial. We still identify the polynomial P with the array $P[\_]$.

Lemma 4. ***(Evaluation formula analogue to Lemma 1)***
Let $f : [0..k] \hookrightarrow [0..n]$ be any shift embedding $f : x \mapsto x + a$ (in the context of this chapter, we will call these "admissible"). Let P be an array of length k (corresponding, through monomial basis, to a polynomial $P(x) = \sum\limits_{i \in [0..k]} P[i] \cdot x^i$). Then,

$$(f_* P)(t) = t^a P(t)$$

Proof. Obvious. □

V-SPARK protocol naturally transitions to monomial basis, and can be used for any segments and shift embeddings - the alignment restrictions no longer apply. However, many other questions remain unclear — main one being, how does one run a sumcheck in this basis?

3.2 Inner product argument in a monomial basis

Inspired by the cohomological sumcheck argument from [6], we suggest an inner product argument in a *coefficient* basis. Suppose that we are given a pair of oracles to univariate polynomials $P(x)$ and $Q(x)$ of degree $n-1$. Then, their inner product in coefficient basis is equal to:

$$\langle P, Q \rangle = (P(x)Q(x^{-1})x^{n-1})[n-1]$$

Lemma 5. *Assume ζ is nonzero and not a root of unity of order $< n$. Then, for a polynomial T of degree $2n-2$, its $n-1$-th coefficient equals 0 if and only if there exists a polynomial M such that:*

$$M(\zeta x) - \zeta^{n-1}M(x) = T(x)$$

Proof. Let's consider a linear operator $M(x) \mapsto M(\zeta x) - \zeta^{n-1}M(x)$. It acts diagonally on monomials, sending a monomial $x^k \mapsto (\zeta^k - \zeta^{n-1})x^k$. It obviously vanishes for $k = n-1$, so it is impossible to find such an M if $(T(x))_{n-1} \neq 0$. Additionally, all other coefficients are multiplied by $\zeta^{n-1}(\zeta^{k-(n-1)} - 1)$. By our assumption, this is nonzero for k ranging from 0 to $2(n-1)$, which means that it is possible to reverse it on all other monomials. $\square$

The argument (which we call **Knuckles**, for its resemblance to the Sonic[17] commitment scheme) works as follows:

1. Assume that the prover has already sent oracles to P, Q, and claimed inner product c.
2. Send commitment to M, receive random challenge t.
3. Validate that $P(t)Q(t^{-1})t^{n-1} - c = M(\zeta t) - \zeta^{n-1}M(t)$ by opening $M(t)$ and the oracles.

This argument is quite versatile; for example, it can be used to construct a multivariate commitment scheme from a univariate one by setting Q to represent the coefficient-form cast of the evaluations of a multilinear eq-polynomial.

3.3 Univariate to Multivariate Transition

In the simplest version, we will keep the targets multivariate but make the witness univariate (in coefficient form). Let us consider a target $T[y]$, equal to $\sum\limits_{x} M[y,x]W[x]$. The matrix M is represented by a polynomial which is multivariate in the variables y (and uses an evaluation basis) and univariate in x (and uses a coefficient basis).

When an evaluation claim $T(r) = c$ is received, one first needs to pass the following sumcheck:

$$\sum_{j} (M(r,x))_j (W(x))_j = c.$$

This is done using Knuckles.

The second stage is the evaluation of $M(r,r')$. Here, we use a modified version of V-SPARK (in the target dimension, the admissible subsets are as before, but in the source dimension, any subsegment is admissible).

3.4 Hybrid Approach

Importantly, the Knuckles argument requires us to perform an additional commitment to a $\sim 2N$-sized polynomial (where N is the witness size). This is a significant overhead, which can be decreased by instead using polynomials of the form $P(x_0, ..., x_{u-1}, \widetilde{x})$, which are multivariate in the first few variables and univariate in the last one.

When asked to compute the inner product of two such polynomials, we can first run the sumcheck over the multivariate coordinates. This way, we reduce the claim

$$\sum_{(x_0,...,x_{u-1}) \in \mathbb{B}^u} \sum_{i} (P(x_0, ..., x_{u-1}, \widetilde{x}))_i (Q(x_0, ..., x_{u-1}, \widetilde{x}))_i = c$$

to the claim of the form

$$\sum_{i} (P(r_0, ..., r_{u-1}, \widetilde{x}))_i (Q(r_0, ..., r_{u-1}, \widetilde{x}))_i = c',$$

to which we can then apply Knuckles.

No other changes are required. Realistically, most univariate commitment schemes already include batching, so they naturally support this hybrid commitment scheme. For practical purposes, u can be chosen between 4 and 6, trading proof size for prover performance.

From the standpoint of V-SPARK, there is an additional choice to be made: do these multivariate coordinates correspond to the least or most significant bits? We think that convenient choice is to make most significant bits multivariate. In this case, the memory is split into 2^u **pages**, and an admissible segment is either a subsegment which fits into a page or an admissible collection of pages.

3.5 Univariate Targets

It is also possible to use the univariate (or hybrid) system for the targets. It is unclear if this is really necessary, and u for the targets likely should be large (as we really want to avoid committing to them). This is done by splitting M into two matrices - first is a univariate-to-univariate layer performs the actual transformation, and the second layer is a univariate-to-multivariate layer representing a diagonal matrix.

4. Adding dynamism

We now explain the killer feature of the Morgana proof system — the fact that, in addition to a circuit builder, we get dynamic branching (almost) for free! In our opinion, this is very important, as it completely nullifies the tradeoff between a circuit-based approach and a VM approach. We get something very close to the normal programming flow (arguably, even better, as the ability to use non-determinism is retained), without losing performance (and even gaining performance compared to Plonkish approaches).

4.1 Design space

There are multiple ways of adding dynamism. We have only preliminary understanding of the different tradeoffs that these approaches pose.

The most principled one (and the one that we didn't investigate too much) would be allowing the circuit to output the code of other circuits. This should be enough to construct a form of λ-calculus.

A simpler approach, which we outline here, is to make a provable VM capable of outputting the circuit key of the circuit. A minimal such VM has normal RAM, stack, and a write-once memory that hosts the circuit description.

4.2 Minimal VM

A minimal virtual machine should be capable to perform the following operations:

1. Basic arithmetic over field $\mathbb{F}$ (likely only additions are enough), reads / writes to RAM, conditional jumps.

2. Non-deterministic guess - an operation that synthesizes unconstrained prover input on top of a stack.

3. Ability to write in write-once memory representing the cached-sparse matrix system. The semantics of this process are not that important, but there is a mission critical property - the VM **must not** be able to edit the already constructed matrices after finalizing them.

4. An allocator, capable of allocating cached-sparse matrices on the main component. Allocated components **must not** be able to reuse elements of the targets. (in reality, the intermediate components will likely use the same allocator, but this property is only crucial for the main component)

Definition 17. *Given such VM, the (base)* ***Morgana VM protocol*** *performs the following sequence of operations:*

1. *A VM is ran on the code / input of the program and constructs an MCS circuit key. The corresponding verifier key is exposed as one of the commitments of the proof.*

2. *A V-SPARK-based Spartan protocol is applied to MCS. The proof is correct if both of these are correct.*

Lemma 6. ***(Circuit transformation)*** *For a circuit f it is possible to write a Morgana-VM program of size $= O($size of the modular description$(f))$ which executes the circuit.*

Proof. The program works as follows:

1. Construct the circuit (write the required cached matrix description in write-once memory).

2. Allocate it to the main component.

3. For every stack input / output c and corresponding circuit input / output x add a linear constraint $x - c \cdot 1 = 0$. Here, c is treated as a constant from the perspective of the circuit - i.e. this constraint binds the inputs and outputs of the circuit to the values on a stack. From a VM perspective, outputs are obtained non-deterministically - but it enforces the constraint that will then be checked during the circuit validation phase.

Due to conditions (3), (4), the circuit stays intact (since further actions of VM can not alter an already allocated circuit). This means that the prover trying to supply an incorrect output will fail the second phase of the protocol. □

Moreover, if the circuit is called multiple times, the cost of the call becomes O(input/output size). The reason for this is that after the circuit is allocated, it becomes a modular component that is callable. Therefore, the only work that VM needs to do is add a call and constraints binding the I/O.

Importantly, the conditions (3) and (4) are crucial here - if a VM is able to retroactively edit the description of f, it might be possible to attack it using the following pattern: first non-deterministically output wrong result of f and then use it to retroactively change the definition of f. Similar issue happens if the allocation integrity is violated - the prover can retroactively add value to some of the constraints of f to meddle with the definition of f.

4.3 Extension: adding RAM accesses to circuits

It is important to differentiate between branching and random access - branching manipulates the program counter of the VM. We do not treat RAM accesses as branching.

This can be done by allowing the circuit to access (and constrain) the execution trace of the VM. The exact design of this can vary depending on the design of the VM itself - the most straightforward option is introducing (or simulating using guess) non-deterministic read / write opcodes that do not interact with the stack but are reflected in the RAM access trace.

When invoking the circuit, a VM now also does a sequence of n non-deterministic read / write opcodes and passes the corresponding chunk of the execution trace ([(timestamp$-n$)..timestamp]) as an input to the circuit. The circuit simulates these accesses by constraining the input.

This allows circuit components to access the memory without overhead.

4.4 Extension: adding dynamic lookup accesses to circuits

For arrays defined on the circuit-level of the system (i.e. existing as witnesses to the main component, not in RAM) it is desirable to have a separate lookup argument to simulate (read-only) random access. The reason for this is that the RAM simulation argument is significantly heavier than a normal indexed lookup.

4.5 Dealing with the control flow

Once again, we reiterate that the maximal expressiveness of our model can be achieved by embracing the functional paradigm. However, we also explain how to transform the imperative code to achieve a significant speedup:

Assume that our code is represented as separated blocks of instructions, starting with jump entry point and ending with a conditional jump. This is a technical condition. Every block of instructions then gets transformed to a single circuit, with inputs and outputs representing the action of this block on a stack. The corresponding block is then replaced by the following sequence of operations:

1. Allocate the circuit.
2. Read the input from the stack, nondeterministically write the output to the stack, add constraints as in lemma 6.
3. Do n non-deterministic RAM accesses (where n is a total amount of RAM accesses in a circuit).
4. Pass these accesses as RAM input as suggested in Section 4.3.

This completely eliminates the concept of a ”precompile” - any branchless code block becomes a precompile in this paradigm.

5. Bibliography

[1] Arasu Arun and Srinath Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. Cryptology ePrint Archive, Paper 2024/1605, 2024.

[2] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023.

[3] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.

[4] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019.

[5] Benedikt Bünz and Binyi Chen. ProtoStar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023.

[6] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Paper 2019/1047, 2019.

[7] Liam Eagen, Ariel Gabizon, Marek Sefranek, Patrick Towa, and Zachary J. Williamson. Stackproofs: Private proofs of stack and contract execution using protogalaxy. Cryptology ePrint Archive, Paper 2024/1281, 2024.

[8] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[9] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015.

[10] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016.

[11] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. Cryptology ePrint Archive, Paper 2024/278, 2024.

[12] iden3. Circom language. https://docs.circom.io/.

[13] Abhiram Kothapalli and Srinath Setty. HyperNova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023.

[14] Abhiram Kothapalli and Srinath Setty. NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive, Paper 2024/1606, 2024.

[15] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370, 2021.

[16] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. Cryptology ePrint Archive, Paper 2024/387, 2024.

[17] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Paper 2019/099, 2019.

[18] Mir protocol. Plonky2. https://github.com/mir-protocol/plonky2, 2022.

[19] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2019/550, 2019.

[20] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023.

[21] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023.

[22] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. Cryptology ePrint Archive, Paper 2013/351, 2013. `https://eprint.iacr.org/2013/351`.