Kaspa Verifiable Programs (vProg) Protocol Specification

Msutton[1], FreshAir08[1,2], and Hashdag[1]

[1]Kaspa Research
[2]Kaspa Ecosystem Foundation (KEF)

DRAFT v0.0.1

Abstract

Zero-knowledge proof systems enable verifiable off-chain applications, yet application sovereignty is inherently at odds with the need for synchronous composability. The vProg architecture is proposed to resolve this tension by enabling sovereign on-site execution, where the L1 sequencer guarantees data availability while metering the precise computational externality imposed on each vProg by its on-the-fly dependencies. The result is a synchronously composable, zk-based L1/L2 system, leveraging the high-throughput, low-latency Kaspa network as its shared sequencing layer. This document details the architecture, core components, and operational semantics that enable this system.

Contents

1 Introduction

1.1 Purpose and Scope

This document specifies the Kaspa vProg protocol. The primary objective is to scale the computational capacity of the Kaspa L1 by enabling off-chain execution through verifiable programs (vProgs), validated via zero-knowledge (ZK) proofs.

The protocol is designed to resolve the inherent tension between two critical requirements: maintaining the sovereignty of individual vProgs while enabling synchronous, atomic composability between them to prevent economic fragmentation.

This initial version of the specification covers the fundamental architecture; many implementation details are reserved for future revisions.

1.2 Architecture Overview

The vProg protocol is designed to solve the classic challenges of synchronous composability in a decentralized setting: enabling trustless, atomic interactions between sovereign applications without creating computational bottlenecks or sacrificing liveness. The core of the architecture rests on the principle of **metered, on-site execution**, ensuring vProg sovereignty.

This sovereign model is the protocol's core mechanism for partitioning its computational and state burdens. This provides a direct path to state-size scalability. It also enables the horizontal scaling of computation, as the depth of re-computation required for cross-program dependencies can be minimized by frequent ZK proof submissions.

In this model, vProg sovereignty is achieved by having each program's nodes locally execute the logic of any external vProgs they depend on. To facilitate this, the Kaspa L1 acts as a sequencer and data availability layer. It does not execute vProg logic itself, but critically, it guarantees that all necessary witness data for a transaction is available and calculates the full computational dependency, or "scope", that this on-site execution imposes on each vProg. By metering this scope, the L1 can regulate the computational throughput per vProg, preventing any single application from being overwhelmed by the demands of the wider network.

The witness data supplied by transactions is anchored in zk-proven commitments to the historical states of vProgs. The more frequent proof submissions become, the more "shallow" these anchors can become, and in turn the computational and execution externalities imposed by one vProg on the other reduce. This creates a powerful economic incentive for a healthy, active prover ecosystem to keep the system scalable.

Closely intertwined is the coordination of proving efforts. Cross-vProg transactions inherently require that one vProg await a proof for a computation of another before it can finalize proofs of its own state transitions. Indeed, our architecture shines the brightest where there is some degree of cooperation between provers of the distinct vProgs. Crucially though advancing the state commitment of a vProg never depends on any data that cannot be reconstructed locally (though doing so may be less efficient). While reconstructing the required data may not be as efficient as the "optimistic" cooperative option, this "sovereign" option crucially ensures the liveness of a vProg is never compromised by the fault of others.

The following sections detail the components that enable this model: the L1 consensus modifications, the structure of the Computation DAG (CD) that underpins both scope calculation and proof stitching, the operational flow of transactions, and the economic model that aligns incentives between all participants.

2 Framework Description

2.1 Global Time

L1 determines a global time by sequencing 3 distinct types of operations: $\mathsf{Op} := \{\mathsf{Tx}, \mathsf{Stitch}, \mathsf{CondBatch}\}$ detailed respectively in Section 2.3, Section 2.4 and Section 2.5. The global, ordered sequence of operations finalized by the L1 is denoted $T = \langle op_1, op_2, \dots \rangle$.

2.2 The vProgs Model

A verifiable program, or vProg, p, is a sovereign application defined by a state transition function exec_p. Each vProg exclusively owns a set of accounts, A_p, and is solely responsible for authorizing modifications to their state. From an architectural perspective, a vProg functions as an independent, verifiable state machine whose state progression is periodically committed to the Kaspa L1.

An account is the fundamental unit of state storage. We denote by $\mathcal{S}_p$ the state space of accounts owned by p. Only the executable exec_p of the owning vProg p has write-access to accounts in A_p, but its value may be read by any vProg. We denote $D_{p,a,t}$ the value of a in p at time t. Accounts are referred to as local if they belong to A_p, and foreign otherwise. The state of a vProg is the union of the latest state of all accounts owned by it, and the global state of the system is the union of the latest state of all accounts.

In this revision of the yellowpaper, we will assume each vProg has a permanent static set of accounts, and will not dwell on questions of creation or deletion of accounts.

2.2.1 vProg Clients and States Cache

A client running a vProg's main purpose is to compute the latest state of all its owned accounts. However this latest state of an account may depend on the latest state of accounts in other vProgs, which in turn may themselves depend on others—including the past states of local accounts. A historical cache is preserved to facilitate execution of these dependencies. This historical cache consists of historical states of local accounts, and the states of foreign accounts.

Crucially, the addition and deletion of states from this cache is to be determined by L1 consensus. In this revision of the yellowpaper, we only describe the additions to the cache, but it will be relatively clear that historical states do not need be maintained indefinitely. The exact details of pruning these are deferred to a later revision of the yellowpaper.

We denote by K_p^t the set of account states that are maintained in the cache of the vProg client p at time t.

2.3 Transactions

A transaction is the atomic operation that drives state transitions in the system. Each transaction explicitly declares:

- the set of accounts it intends to read.
- the set of accounts it intends to write to.

We write $R(\text{tx})$ for the declared read set and $W(\text{tx})$ for the declared write set of transaction tx.

This declaration allows L1 to pre-compute the dependency graph and guarantees that all writes are controlled under a single state machine.

General model. In principle, a transaction could update accounts across multiple vProgs. Conceptually, this would amount to splitting it into smaller subtransactions, each executed by the relevant vProg. The outputs of one would serve as inputs to the next, until all declared writes are resolved.

Restriction in This Revision. To avoid the detail clutter of subtransactions syntax in this early revision, we restrict for now to the case where all writes belong to a single vProg, called the Writer vProg p_w.[1] The transaction may still read from many vProgs, but only p_w actually updates its accounts. Execution is simply:

$$T_{\text{txid}}(R, i) = \text{exec}_{p_w}(R, i),$$

where R are the declared read states, i represents auxiliary inputs (calldata, signatures, etc.), and exec_{p_w} is the deterministic state transition function of p_w. The result is a set of new values for the accounts in $W \subseteq A_{p_w}$.

2.4 Stitching Proofs and vProg State Index

Commitments of the form $(p, t) \mapsto C_p^t$ are mapped in a structure called the vProg state index. The commitments represent recent states of the various vProgs. New commitments can be added to this structure via an operation called a stitching proof (see Algorithm 1).

Commitment Submission. Each new submission to it satisfies:

1. *Monotonic time:* If the latest accepted commitment for p is C_p^s, then the new submission must have $t > s$.
2. *Full coverage of current ownership:* C_p^t must commit to the latest state at time t of every account owned by p.
3. *Historical continuity:* C_p^t must also commit to any intermediate states of p's owned accounts for every timestamp t' with $s < t' \leq t$ (inclusive of t, exclusive of s).
4. *Efficient attestation:* The commitment must support succinct membership/opening proofs (e.g., Merkle/accumulator based) so that an L1 verifier can check any claimed value for any $(\text{account}, t')$ with logarithmic (or similarly sublinear) witness size and verification cost.

Pruning Note. Once a commitment C_p^t is sufficiently buried, the prior entries and intermediate material in it is potentially redundant and thus the commitment can be pruned from the structure; exact policies are out of scope for this revision.

vProg Genesis Note. More discussion is required on the initialization of a slot for a vProg in this state index. This discussion is reserved for future versions.

[1]The general case is in a sense the essence of atomic composability, and will need to be described in detail. Nevertheless, the restricted model suffices to describe the core primitives, from which the generalization to the multi-writer case follows directly. This generalization conceptually involves a composition of conditional proofs, where an "outer" vProg's proof is conditioned on the certified outputs of an "inner" vProg. A sketch of describing the general case where multiple writers are present can be viewed in the research forum `https://research.kas.pa/t/a-basic-framework-for-proofs-stitching/323`.

2.5 Conditional Proof Batches

Conditional Proofs. A conditional zk-proof attests to the outputs of a transaction writing to vProg p, conditioned on its inputs. Several such standalone proofs of the same vProg can be batched together to a primitive called *conditional proof batch*, which arranges the commitments of each individual proof in a Merkle tree, and supplies a zk-proof attesting that the Merkle root forms the root of a valid tree where each leaf is individually zk-proven.

2.6 The Computation DAG (CD)

The flow of state dependencies between accounts is modeled as a directed acyclic graph (DAG), the Computation DAG (CD). The structure of this graph is determined dynamically by the global sequence of transactions finalized by the L1 and serves as the canonical reference for all L2 proof verification. The CD will serve two distinct primary functions: facilitating static inference of (a bound on) the excess computation imposed on a given vProgs by computations in the jurisdiction of other vProgs (Section 3.2.2), and defining the dependency graph for L2 proof stitching, as expanded in Algorithm 1.

Vertices. The CD contains two types of vertices:

- *Account State Vertex* (V)*:* Represents the state of an account at a specific logical time. Its ID is a tuple $((pid, aid), t)$. We denote such a vertex by $v^{\text{acc}}_{pid,aid,t}$. For conciseness, we use the notation $v^{\text{acc}}_{pid,aid,\leq s}$ to refer to the latest state of an account *as of* a specific time s. This denotes the most recent state vertex of account (pid, aid) at or before logical time s. Formally, this means $v^{\text{acc}}_{pid,aid,\leq s} := v^{\text{acc}}_{pid,aid,t}$ where $t \leq s$ is the maximal timestamp for which such a vertex exists.
- *Transaction Vertex* (τ)*:* Represents a transaction, identified by its L1 transaction ID, (txid) and time t. We write $v^{\tau}_{\text{txid,t}}$ and view it as a function node consuming reads and producing writes.

Edges (E). A transaction vertex has incoming edges from the account state vertices it reads and outgoing edges to the new account state vertices it creates. Formally, for a transaction tx finalized at time t and represented by the vertex $v^{\tau}_{\text{txid,t}}$:

- For each account $(p, a) \in R(\text{tx})$, an incoming edge is added from $v^{\text{acc}}_{p,a,\leq t-1}$ to $v^{\tau}_{\text{txid,t}}$.
- For each account $(p, a) \in W(\text{tx})$, an outgoing edge is added from $v^{\tau}_{\text{txid,t}}$ to the newly created vertex $v^{\text{acc}}_{p,a,t}$.

Notation Note. A natural structure for this graph would be a hypergraph where transactions act as hyperedges connecting account state vertices. However for the purpose of describing the stitching process it is simpler to represent every transaction as a vertex, with outgoing edges to its write set, and ingoing edges from its read set.

2.6.1 Computational Scope and Transaction Anchoring

The value of an account vertex v in the CD is determined by its predecessor transaction vertex and the values of that transaction's inputs. However, a vProg client typically only holds a limited subset of account states, denoted K_p^{t-1}. This subset does not necessarily include all the vertices required to execute the transaction's declared read set. In some cases the missing values can be derived from K_p^{t-1} alone, but often they cannot.

To guarantee data availability for every vProg, transactions are therefore required to supply additional historical values sufficient to reconstruct their read set. These additional values are called *anchors*. A transaction is deemed invalid if it fails to provide anchors covering all missing dependencies. An anchor value must ground itself to a vertex familiar to a commitment in the vProg state index. This familiarity is enforced by the L1 (see Section 3.3.1).

Once a transaction's anchors are verified, a vProg node determines the computational scope by performing a backwards traversal on the CD, starting from the transaction's read set. The traversal along any path halts immediately upon reaching any vertex that is either (1) one of the transaction-supplied anchors, or (2) already present in the vProg's local cache, K_p^{t-1}.

The subgraph consisting of all vertices (both account state and transaction) and edges discovered during this traversal constitutes the **scope** for vProg p with respect to the transaction at time t, denoted $\text{scope}(p, t)$. It represents the exact dependency graph the vProg must process to derive the values of the transaction's inputs. This may involve the vProg locally executing the logic of parent transactions within the scope, even if those transactions belong to foreign vProgs. The set of new account state vertices within the scope, denoted $V_{\text{scope}}(p, t) := scope(p, t)\restriction_V$, corresponds precisely to the increment to the vProg's cache:

$$K_p^{t+1} = K_p^t \cup V_{\text{scope}}(p, t)$$

Finally, note that L1 nodes can and will calculate the set (but not the data) $\text{scope}(p, t)$ for each transaction and vProg. This allows L1 to act as a scheduler, regulating computational throughput per vProg and preventing any single vProg from being overloaded by cross-program dependencies.

2.6.2 Computation DAG Commitments

To allow attesting for continuity of its structure, each vertex in the Computation DAG is mapped to a hash via the following recursive rules:

$$\mathrm{H}(v_{\mathrm{acc}}) = \mathrm{H}\big(\mathrm{H}(v_{\mathrm{tx\,parent}}) \parallel (pid, aid) \parallel t\big),$$
$$\mathrm{H}(v_{\mathrm{tx}}) = \mathrm{H}\big(\mathrm{H}(v^{\mathrm{acc}}_{\mathrm{pred}1}) \parallel \cdots \parallel \mathrm{H}(v^{\mathrm{acc}}_{\mathrm{pred}n}) \parallel \mathrm{txid}\big).$$

This recursive hashing creates a unique, verifiable commitment for any vertex and its entire causal history, which will be fundamental for proof stitching. Unlike the potentially fragmented compute scopes calculated by L1, the proving process will require a complete and continuous segment of the CD to be covered by proofs, ensuring the integrity of the state transition between two L1-anchored commitments.

3 L1 Consensus Modifications

3.1 ZK Verify Capabilities

Kaspa will require new capabilities to allow it to run ZK verifications. The precise mechanics of how this capability will be enabled are under consideration. The cleanest option considered was the introduction of designated ZK opcodes, alongside opcodes for transaction inspection (covenants). See [1], [2]. Due to the need of L1 to be explicitly informed of state commitments of the various vProgs, at various times (to allow for anchors verification), this option is no longer as natural, and more intrusive options might be considered.

3.2 Resource Metering

Consensus will regulate two new types of masses, in addition to the existing compute mass (regulating computations done by the L1), permanent storage mass and transient storage mass. The new masses are referred to as L2 gas[2], and L2 scope gas. Both of these new types are in practice a sparse vector indexed by the various vProgs existing in the system, each entry regulated separately. Transactions will commit to a bound of spending on both types of gas. The implications of these commitments however are distinct: the scope gas commitment will be verified by the merging block, and cause a rejection of the transaction in case of a failure (in any coordinate)[3]. The L2 gas however is an L2 inner construct merely regulated by L1, and L1 itself will be oblivious to any failure to meet the L2 commitment.

3.2.1 L2 Gas

This is a vProg-specific resource metric, defined and priced by each sovereign vProg to manage its internal execution and state costs. The `GasPayments` map in a `Transaction` represents the user's payment for this L2 resource, which is ultimately to be claimed by the vProg's prover. This gas measure represents (a bound on) the internal execution in the vProg prover, excluding all the external calculations of foreign accounts occasionally required to derive a transaction's dependencies. In turn this measure represents the proving cost of the transaction in that vProg. L1 blocks regulate the L2 gas per vProg so it will not exceed a predetermined bound, possibly differing between different vProgs. We will refer to the sum of its L2 gas on every vProg as the total gas of a transaction.

3.2.2 ScopeGas

The scope gas represents the L1-verifiable computational load that a transaction's scope imposes on a full node of p for state reconstruction. Recall that τ denotes the set of transaction vertices. Then

$$\mathrm{ScopeGas}(\mathrm{tx}, p) \;=\; \sum_{v \in \left(\mathrm{Scope}(p,\mathrm{tx}) \cap \tau\right)} v.\mathrm{total\_gas},$$

The merging block validates the user-declared bound for scope gas; if exceeded in any coordinate, the transaction is deemed invalid.

3.3 vProg State Index and Stitching Covenant

The L1 maintains the vProg State Index. A canonical covenant manages the vProg State Index. This covenant specifies the rules under which a new state commitment C^t_p is accepted into the vProg State Index.

[2]In the restricted model, each transaction has only a single writer vProg p_w, so representing L2 gas as a vecotr is unnecessary. We nevertheless adopt this representation in order to preserve consistency with the full model, where the vector form is essential: the scope gas calculation depends on the aggregate contributions of L2 gas across all writer vProgs throughout the scope, and referring to L2 gas as a scalar fails to emphasize this aspect of its calculation.

[3]In upcoming revisions, this per-transaction commitment model may be revised. An alternative approach is for the merging block to deterministically calculate the scope gas for each transaction and include transactions up to the including block's own capacity limit, dropping those that do not fit, without requiring a user-declared commitment.

3.3.1 Transaction Anchors Verification

L1 consensus rules are extended to include the verification of transaction anchors, leveraging the *historical continuity* of state commitments. For each `Anchor` in a `Transaction`, L1 performs a two-step verification. First, it uses the `state_timestamp` (t') to look up the corresponding state commitment, $C^p_{t'}$, in its local vProg State Index. Second, it verifies the provided `proof` against this commitment. The proof must cryptographically attest that the commitment $C^p_{t'}$ includes the historical fact that the given `account` had the specified `value` at the intermediate `anchor_timestamp` ($t \leq t'$). A transaction is considered structurally valid only if all of its anchors are successfully verified.

3.4 DAG Maintenance and Scope Calculation

L1 nodes will be responsible for maintaining the topological structure of the Computation DAG based on the finalized transaction sequence. Upon validating a transaction tx_t, the L1 node adds a new transaction vertex v_{tx} and its corresponding new account state vertices v_a to the graph. As we detail below, L1 will also be responsible for managing the known set of vertices for each vProg and regulating the scopes of transactions, as well as computing and storing hashes for "tips" of the Computation DAG.

3.4.1 Transaction Scope Maintenance

A critical consensus rule is introduced to handle the dynamic nature of transaction ordering in the blockDAG. A merging block, which finalizes the order of transactions from parallel blocks, calculates the scope of each transaction and its ScopeGas based on its final position in the sequence. If a transaction's calculated ScopeGas for any vProg exceeds its corresponding commitment, it is deemed invalid. This rule prevents scope explosion attacks and ensures the deterministic regulation of vProg throughput.

L1 maintains only *metadata* sufficient for deterministic scope calculation; it does not store account values. Concretely, each account-state vertex v carries a list $v.\mathsf{stateful\_vprogs}$ (the vProgs that already know v's value). During sequencing, for each writer vProg p in the write set, the L1 traverses predecessors down the CD until (i) the transaction's ScopeGas commitment would be exceeded, (ii) a vertex with $p \in \mathsf{stateful\_vprogs}$ is reached, or (iii) an on-chain anchor supplied by the transaction is encountered. Successful traversal marks newly discovered vertices as known to p by updating $v.\mathsf{stateful\_vprogs}$.

3.4.2 DAG Root Maintenance

A vertex $v^{\text{acc}}_{p,a,t'}$ is *proven* once a commitment C^t_p accepted into the vProg State Index covers it, where $t' \leq t$. Prior to this, it is *unproven.* To maintain a succinct commitment to the set of all unproven vertices in the CD, each L1 block header contains a DAG Root.

This root is constructed from a two-tiered Merkle structure:

1. **Account Write Tree:** For each vProg p, an Account Write Tree tracks its set of *live accounts* (i.e., accounts with at least one unproven state). The leaves of this tree consist of the single **latest** unproven state vertex for each of these live accounts. At a given L1 time t, the leaf for a live account a is precisely the vertex $v^{\text{acc}}_{p,a,\leq t}$. The root of this tree is denoted R_p.
2. **vProg Tree:** The DAG Root is the Merkle root of all R_p across all live vProgs: $\Psi = \mathsf{MerkleRoot}\big(\{R_p\}_{p \in \mathcal{P}}\big)$.

Live Accounts Note. it will be worthwhile to consider regulating the amount of live accounts permitted per vProg. To be discussed in future revisions.

4 vProg Covenants

The pegging of a vProg in L1 will be enforced by two covenants with ZK capabilities:

4.1 Batches Verifier

This covenant is responsible for verifying conditional proof batches submitted with its id. The covenant is defined by a verification key vk created via standard methods to correspond to the executable $exec_p$.

4.1.1 Conditional Batches

A *conditional batch* publishes a Merkle root `ProofsRoot` committing to a set of conditional statements regarding the transactions of a single vProg p.

```
type Hash = [u8; 32];

struct ConditionalProof {
    ConditionHash: Hash,
    ResultHash: Hash,
    TxnHash: Hash,
    ProgID: vProgID,
}

struct ConditionalBatch {
    ProofsRoot: Hash, // Merkle root over encoded ConditionalProof leaves
}
```

Listing 1: Abstract Proof Structures

The `ConditionHash` is a commitment to the set of all account states read by the transaction, $R(tx)$. Let $D(a)$ be the state data of an account a. The hash is computed as:

$$C = H(\dots \parallel (p_i, a_i, D(a_i)) \parallel \dots) \tag{1}$$

where the tuples are concatenated in a deterministic, lexicographical order. The `ResultHash` is computed similarly over the written accounts $W(tx)$.

4.1.2 Conditional Batch Proof Verification

A conditional batch proof operation must supply along it a zk-proof for its validity. The submitted zk-proof must establish that:

1. **Well-formed Merkleization.** the public input `ProofsRoot` is the Merkle root of a tree whose leaves are byte-encodings of `ConditionalProof` records with fields `(ConditionHash, ResultHash, TxnHash, ProgID)`.
2. **Single-vProg scope.** Every leaf represents a transaction with `ProgID` $= p$.
3. **Per-leaf statement shape.** Each leaf represents "the execution of the program $exec_p$ embedded on the verification key on the transaction with id *txid conditioned* on the declared reads results in the declared writes" i.e., the proof system enforces the conditional-validity semantics tied to `ConditionHash` and `ResultHash`, without asserting anything about other vProgs or global finality.

It is emphasized that this covenant does not require the off-chain leaves for verification; it only checks the zk-proof that binds them to `ProofsRoot` and p.

4.2 Stitching Covenant

This covenant's main responsibility is to sanction commitment submissions to the vProg index. New commitments will only enter the index if they are authorized by this covenant.

4.2.1 Stitching Proof

A *stitching proof* operation publishes a new state commitment for a vProg, together with evidence that the transition is consistent with conditional proofs and previously attested anchors.

```
type Hash = [u8; 32];
type MerkleProof = Vec<byte>;

struct BatchRef {
    ProofsRoot: Hash,          // batch Merkle root
    HeaderRef: Hash,           // block header reference
    TxInclusion: MerkleProof, // proof of inclusion in HeaderRef
}

struct RefPointer {
    ProgID: vProgID,           // q_j
    Time: u64,                 // r_j
}

struct StitchingProof {
    ProgID: vProgID,           // primary vProg (p)
    NewCommitment: Hash,       // C^p_t
    DAGRoot: Hash,             // psi^p_t
    StartTime: u64,            // s
    EndTime: u64,              // t
    VProgTreeWitness: MerkleProof, // psi^p_t \in Psi_t witness

    RefPointers: Vec<RefPointer>,  // {(q_j, r_j)} references
    Batches: Vec<BatchRef>,        // conditional batch roots + inclusion


    ZkProof: Vec<byte>,            // proof of stitching predicate
}
```

Listing 2: Abstract Stitching Proof Structure

The covenant validates the proof inputs as follows:

- **Primary start.** Fetch the latest commitment C^p_s to the primary vProg from the index. Verify `StartTime` $= s$, and bind this to the zk input.

- **End time and vProg-tree inclusion.** Require `EndTime` $:= t > s$; verify `VProgTreeWitness` fetch the vProg tree commitment of the block merging time t and verify $\psi_t^p =$ `DAGRoot` is included in that commitment.
- **Foreign/historical commitments.** For each $(q_j, r_j) \in$ `RefPointers`, fetch $C_{r_j}^{q_j}$ from the index, compute the hash `RefCommitmentsHash` $= H(\{C_{r_j}^{q_j}\}_j)$ in canonical order.
- **Batch roots.** For each `BatchRef`, verify `TxInclusion` attests that `ProofsRoot` is on-chain in the block with `HeaderRef`. Hash all proofsRoots together to derive `CondsRoot`.
- **Stitching predicate.** Finally, verify `ZkProof` under the stitching circuit with public inputs

$$(s, C_s^p, C_t^p, \psi_t^p, \texttt{RefCommitmentsHash}, \texttt{CondsRoot}).$$

vProg MetaState Note. The structural checks of both `RefCommitmentsHash` and `BatchRef` in the covenant, consisting of supplying and hashing many items together, may not conform to bandwidth and computational feasibility requirements[4] if many distinct commitments or conditional batches will be required. We consider this scenario likely. We sketch out one option to address it, to be detailed in future versions. The main idea is to delegate the computational and bandwidth load required to the stitching predicate. To enable this, each vProg commitment should commit not to its client state directly, rather to a "metastate" enveloping it. This metastate will keep track of the proofs (stitching and conditional batches) submitted on chain, and maintain corresponding inner commitments for the proofs within its metastate. These inner commitments could then be deciphered using the private witnesses supplied during proof stitching. To ensure the metastate is itself updated in a deterministic manner, it may be required that L1 maintain a separate "sequencing commitment" for proof operations. This too shall be detailed in future versions.

Stitching Algorithm. The logic of the stitching predicate, detailed in Algorithm 1, consists of three main phases. First, the predicate performs structural validation on the private inputs, ensuring all provided witnesses and proofs are valid against their corresponding public commitments. Second, it reconstructs the computational history of the segment by processing the conditional proofs in order, using the supplied witnesses to resolve all dependencies. Finally, it verifies that this fully reconstructed history correctly results in the proposed new state commitment (C_p^t) and DAG root (ψ_t^p).

[4]Concretely, these feasibility requirements arise from the design of L1 scripting, which does not support features like loops required to iterate over a variable number of inputs.

Algorithm 1 Stitching predicate

```
Require:
 1: $s$: starting sequence time of the segment
 2: $C^p_s$: state commitment for principal vProg $p$ at time $s$
 3: $C^p_t$: proposed state commitment for $p$ at time $t$
 4: RefCommitmentsHash: state commitment hash for secondary vProgs, or historical states of the primary
 5: $\psi^p_t$: the DAG_Root of $p$ at time $t$
 6: CondsRoot: Merkle root of conditional proof batches
 7: Private Inputs:
 8:    State Commitments $\{C^{q_j}_{r_j}\}_j$, the unhashed state commitments
 9:    seg: CD segment data from time $s$ up to tips covered by $\psi^p_t$
10:    anchors: map indexed by $(p', t')$ with state value and opening witness under the corresponding commitment
11:    conds: ordered set of conditional proofs with inclusion witnesses under CondsRoot
12:    InitialStateView: A set of Merkle proofs against the public state root $C^s_p$, revealing the leaves and branches for all state portions relevant to the segment's execution.
Ensure:
13: A valid ZK proof that the transition from $C^p_s$ to $C^p_t$ is correct
14: procedure GenerateStitchingProof
      /* Phase 1: Verify private input integrity */
15:     Verify that the state commitments hash correctly into RefCommitmentsHash
16:     Verify that all proofs in InitialStateView are valid against the public state root $C^s_p$.
17:     Verify seg obeys hash rules (Section 2.6.2)
18:     for all $c \in$ conds do
19:         Verify inclusion witness of $c$ under CondsRoot
20:     end for
21:     for all $(key, val) \in$ anchors do                          ▷ Verify anchors validity
22:         Verify opening witness attests that vertex $key$ has value $val$ under its referenced commitment[a]
23:     end for
      /* Phase 2: Construct a continuous stitched set of account vertices */
24:     Initialize seg_map $\leftarrow \emptyset$
25:     for all $c \in$ conds do
26:         Verify $c$ is structurally consistent with seg: it commits to a txn $tx$, read set ReadVertices($tx$), write set WriteVertices($tx$), and ProgID $= p$
27:         for all $u \in$ ReadVertices($tx$) do
28:             if $u \in$ seg and some parent of $u$ is not in seg then
29:                 Require $u \in$ anchors; set seg_map[$u$] $\leftarrow$ anchors[$u$]
30:             else
31:                 Require seg_map contains $u$ and its value matches $c$'s ConditionHash opening
32:             end if
33:         end for
34:         for all $w \in$ WriteVertices($tx$) do
35:             Set seg_map[$w$] $\leftarrow$ value opened from $c$.ResultHash
36:         end for
37:     end for
      /* Phase 3: Verify final public outputs */
38:     Verify that the key set of seg_map is equal to seg
39:     Let upd be the set of local accounts written to within seg.
40:     Verify that $C^t_p$ is the resulting Merkle root after using the InitialStateView as a template, applying the latest writes from upd, and propagating the updated hashes to the root.
41:     Verify that the latest writes of any account in upd hash to $\psi^p_t$
42: end procedure
```

[a]In particular, this verifies *key* is an account-state vertex

It is emphasized that the logic of individual transactions is already assumed verified in the conditional batches. The stitching covenant is hence oblivious to features of the vProgs, beyond their ID. The verification key of advancing the various vProgs can very well be a shared one.

5 vProgs Specification

5.1 vProg Code Availability

The on-site execution model assumes that any vProg node has access to the contract code of any other vProg it needs to interact with. A protocol for vProg contract publication and P2P synchronization will be required. The specification for this protocol will be provided in a future version of this paper.

5.2 Deterministic State Derivation Rules

A vProg's state transition function must be deterministic and based solely on the L1 transaction sequence. Each vProg defines its own internal gas model and fee structure, allowing it to regulate its own resources and create a market for its blockspace.

5.3 Composability

Each vProg is free to determine which foreign vProgs it considers acceptable sources of read data. Ideally, this vetting should not be a static whitelist of individual programs, but rather a structured approval of standard vProg infrastructures that the vProg is willing to interoperate with. The resulting pattern of approvals naturally forms a directed graph of read dependencies between vProgs.

Importantly, the Kaspa L1 Computation DAG is agnostic to these choices. From the perspective of L1, all transactions are valid as long as they obey structural rules: a transaction may declare reads from any vProg and writes to its writer vProg. Whether or not such a read is *semantically* acceptable depends entirely on the writer vProg's own rules.

Concretely, a transaction that attempts to write to a vProg p but also reads from a foreign vProg that p has not approved may still appear structurally valid to L1, and may even force p to compute scope for those reads. However, upon execution, p will reject the transaction internally as invalid, and its declared writes will fail.

Finally, by the directed nature of dependencies, no vProg will ever be forced to compute or rely on the value of an unapproved account in order to derive the correct value of any approved account. Composability thus remains strictly under the control of each vProg.

5.4 Account Creation

To be detailed in future revisions.

5.5 Clients Cache Pruning

To be detailed in future revisions.

6 Provers

Provers are for-profit entities responsible for creating stitching proofs and conditional batch proofs. Provers are expected to specialize in the proofs of a particular vProg or set of vProgs.

6.1 Economic Model

Provers' compensation is to be managed by the vProgs themselves. We suggest it be in correspondence to the L2 gas consumed by the transaction. This section will be expanded upon in future revisions.

6.2 Sovereign and Optimistic Paths

Advancing the state commitment of a vProg interacting with others will typically require stitching together transactions of several vProgs. Indeed provers submit proofs of conditional batches on-chain, and these proven conditionals are in principle usable by all. However the individual conditionals are not transparent on-chain themselves. Hence to make use of the conditionals proven by another prover, provers must communicate with each other at the time of stitching to allow deciphering and extracting of the individual predicates within a batch. We refer to this flow as the optimistic path. It is emphasized that the speed of this communication only affects proof latency, not the sequencing latency of the system.

The sovereign path describes the scenario where, for any reason whatsoever, prover communication malfunctions (either completely, or just suffers unsatisfactory delays). In this path unknown batches by other provers cannot be deciphered. However a prover still has the capabilities to advance the state commitment of their local vProg: the segment to be stitched itself is derivable from the DAG, and the anchoring values are by the design of the system known to the vProg clients (which provers are always expected to run). A prover hence always is capable of submitting conditional proofs for the missing batches by their own, and stitching them together to advance their vProg state commitments. It is noted that required witnesses for foreign commitments are available by design, as they must have been supplied by the transactions.

Edge cases regarding pruning may apply though. The details of the sovereign path will be expanded upon in future revisions.

Acknowledgments

We wish to thank Hans Moog for extensive discussions regarding atomic composability and zero knowledge technologies.

A TransactionV1 Rust Specification

To illustrate a potential concrete implementation of the abstract transaction structure, this section includes a detailed Rust specification for 'TransactionV1'.

```rust
// Type alias for a 32-byte public key used for IDs.
type Pubkey = [u8; 32];

// Structure for transactions' provided anchors
struct Anchor {
    /// L1 logical time t' of C^{p}_{t'}.
     state_timestamp: u64,

    /// Anchor logical time t <= t' of C^{p}_{t'} s.t. t is an intermediate state of C^{p}_{t'}
     anchor_timestamp: u64,

    /// Canonical, opaque encoding of the account state value at (p, account, t).
    value: Vec<u8>,

    /// Merkle inclusion proof that (account, value, t) is in C^{p}_{t'}.
    proof: MerkleWitness,
}

// Main transaction structure for vProg interactions.
struct TransactionV1 {
    version: 1,

    // --- V1 Fields ---

    // A single registry of all unique vProg and account Pubkeys referenced in the transaction.
    // Max 128 items.
    keys: Vec<Pubkey>,

    // Maps a vProg index from 'keys' to a list of account indices from 'keys'.
    relations: Vec<(u8, Vec<u8>)>,

    // Indices into 'keys' identifying vProgs that are written to.
    writing_vProgs: Vec<u8>,

    // Indices into 'keys' identifying vProgs that are only read from.
    readonly_vProgs: Vec<u8>,

    // Indices into 'keys' identifying accounts that are written to.
    write_accounts: Vec<u8>,

    // Indices into 'keys' identifying accounts that are only read from.
    readonly_accounts: Vec<u8>,

    // Anchors for scope calculation. The 'u8' is an index into 'keys'.
    anchors: Vec<(u8, Anchor)>,

    // Maps a gas commitment to each vProg in 'writing_vProgs'.
    gases: Vec<(u8, u64)>,

    // Maps a scope gas commitment to each vProg in 'writing_vProgs'.
    scope_gases: Vec<(u8, u64)>,


    // --- V0 Compatible Fields ---
    inputs: Vec<TransactionInput>,
    outputs: Vec<TransactionOutput>,
    lock_time: u64,
    payload: Vec<u8>,
}
```

Listing 3: TransactionV1 Rust Structure