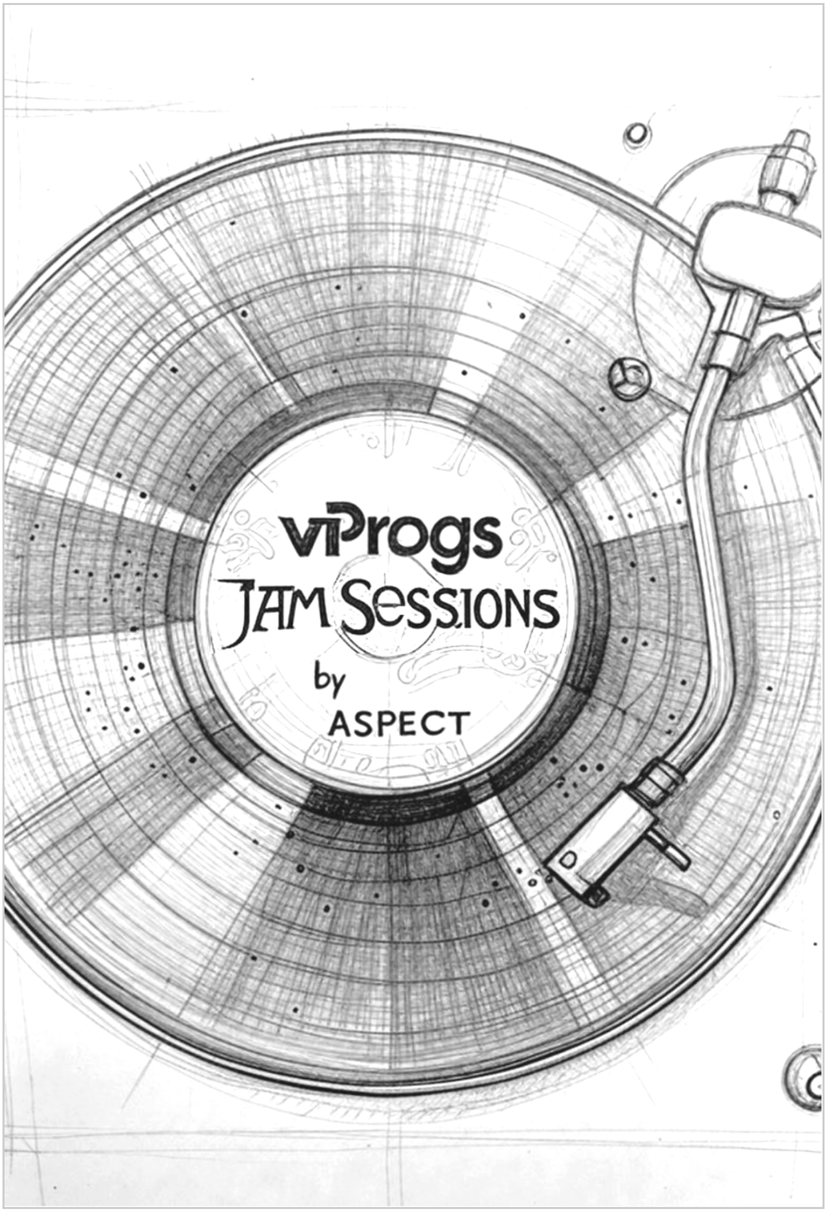

vProgs JAM SESSIONS by ASPECT
Version v1.0.0
December 1, 2025

This document is a compilation of software development concepts, ideas, general research, and feedback on the Kaspa vProgs Yellow Paper v0.0.1.

Soundproofing required.

Table of Contents

1 Introduction

There exist different views on what vProgs represent. Are they sovereign Layer-2 networks, or are they more akin to smart contracts? Due to inherent ZK composability features, this line becomes blurred because vProgs inherit properties of both. The question becomes, from the standpoint of the supporting architecture, should they inherit censorship resistance and decentralization properties of smart contracts, or are the centralized features acceptable?

Statements like “vProg maintainer is responsible for providing the proof” denote to me a centralized “layer-2” mentality, whereas personally, I myself am in the decentralization camp, as I believe this to be the core ethos defined by Bitcoin and carried by Kaspa. There is nothing wrong with centralized control, but I personally believe that when it comes to the supporting infrastructure, it must inherit decentralization properties. vProgs allow for both, and this document views related topics primarily via my personal decentralization prism.

Perhaps the most challenging element of this research and architecture is the novelty of concepts and the lack of formal terminology in many areas. The reader should be aware of that. The interpretation of many terms is very subjective. As I recently found out in the Telegram vProgs chat, even the term "optional" can have multiple context-specific interpretations.

Another challenging element in this complex domain is understanding of motivations behind some of the technical decisions made in the draft vProgs architecture. There occurs close coupling of different concepts that include L1, L2, smart contracts, composability, data availability, security, etc, which at times makes it difficult to understand some of the decisions made or properties resulting from these decisions.

Ultimately, a lot of these aspects will be discussed and worked out as the development moves forward. The goal of this document is to outline various topics deferred by the vProgs Yellow Paper, describe related concepts, and provide a holistic view of various components and subsystems involved in the platform architecture and operation.

2 Key Concepts

While the content of this document is highly technical and it is expected that the reader has a good understanding of this domain, I feel that, for the benefit of the reader, it would be beneficial to review the following core ZK concepts (viewed from the perspective of zkVM design):

- *recursive proof composition* (aka proof recursion, nested proofs, proof aggregation, proof of proof, proof wrapping, verifier embedding, modular verifier circuits ... 🤯)
- *proof adaptors* (aka cross-framework proof verification)
- *conditional proofs* (aka proof assumptions)
- *program trust*

2.1 Proof recursion and aggregation

A ZK program is a regular program for the execution of which one can generate a proof. This proof attests to the fact that program A was given a set of arguments (inputs) and has produced some data (outputs). This proof also attests that the program itself (its binary) or the framework has not been altered in any way.

Proof recursion is a process where a proof generated from the execution of a program A is supplied as an input to a program B. Program B then verifies the proof of program A as a part of its execution. This “encapsulates” the proof A within the proof B. Proof B now attests to the validity of A and B. Subsequently, nesting B within C attests to the validity of A, B, C, etc.

Proof aggregation is the same concept, but generally refers to the verification of multiple proofs in a single pass. Here, proof D can receive proofs A, B, and C as inputs and attest to the validity of A, B, C, and D.

These techniques can be effectively used for proof processing, parallelization, and data compression. If A, B, C are independent proofs, you can prove them in parallel and then aggregate them into proof D. If proofs are sequential (A->B->C->D), proof D can publish arguments of A and prove the final result, collapsing (compressing) A->B->C->D into A->D while attesting to validity of the original proof chain.

2.2 Proof adaptors

A proof adaptor is a proof recursion where proof verification is done in a different ZK framework from the proof being verified. Due to the fact that each ZK framework has different proving implementations and methodologies, this can be used to “jump” between frameworks. For example, an SP1 proof verified in RISC-0 zkVM yields a RISC-0 proof. The verifier of such proof does not need to know about the existence of SP1 zkVM. Since proof verifications are computationally cheap, such approaches can be used to simplify verifier integrations, isolating other proving systems behind adaptors.

2.3 Conditional proofs

If C needs to verify proof B and B needs to verify proof A, proof generation can not be parallelized due to interdependency.

However, B can verify the result (outputs) of A.

Proof generation is an extremely computationally expensive process, where a single proof generation can easily take more than a minute to generate in a modern CPU (GPUs reduce this to seconds). However, a VM (zkVM) that runs the program can also run the program quickly and get the output of the program execution.

The process becomes:

1) quickly run A, B, C, D, collecting arguments Ain, Aout, Bin, Bout, etc.
2) in parallel generate partial proofs A(in,out), B(in,out), etc
3) aggregate partial proofs into a final proof E that verifies proofs A, B, C, D while also comparing that Aout == Bin, Bout == Cin, etc.

Various techniques, such as cryptographic commitments (argument hashes), can be used to reduce the exchanged data footprint.

2.4 Program trust

You can only trust a program that you have created yourself (or otherwise have access to the source code, etc). If someone gives you a program, you can run it, generate proofs of its execution, but you don't know what it does internally, and as such, proofs can only attest to what these programs do. A ZK proof is meaningless unless you have direct trust (you know exactly what the program does) or you delegate trust (you trust the person, service, system) that what it does is correct.

In an ecosystem that allows users to publish their own programs, there are 3 types of trust:
- trust in the network (i.e., Kaspa and the vProg layer)
- trust in the user program's compliance with network rules (aka protocol compliance)
- trust in the published program itself (reputation trust).

The network and the execution environment can and must ensure that the published program has protocol compliance. I.e., It is the responsibility of the network to ensure that the program receives and produces data correctly.

Reputation trust, however, is similar to any smart contract or publicly accessible service. You can only trust it if the entity publishing it has a reputation or the source code is available. In the ZK ecosystem, just like regular smart contracts, CODE is LAW. The great thing about these systems is the fact that they are deterministic and compiling them will yield the same bytecode or the same program hash, allowing one to verify the authenticity of the published source code and the fact that this code is, in fact, being used to publish a program.

3 Infrastructure

3.1 ZK ecosystem

For the purpose of adoption, from the standpoint of ZK ecosystem neutrality, the architecture should not assume use of any specific ZK VM technology and be technologically agnostic, providing equal opportunity for integration with all ZK platforms, including future versions of ZK platforms and platforms that do not currently exist.

This approach brings a number of penalties in contrast to a single framework “locked in” technology execution path, but I argue that holistically, technological neutrality brings long-term scalability benefits and gives the user freedom of choice.

This subject should be the highest priority consideration as design decisions made in relation to this impact the project as a whole. Once locked into a framework ecosystem, it becomes extremely difficult to break free.

3.2 The nested VM problem

As discussed at length on the vProgs telegram channel, there exist two forms of execution - single-layer VM and dual-layer VM (aka nested VM).

Dual-layer VM provides for a much more flexible execution environment, but results in x5-x50 cycle overhead and x5-x30 larger proof traces. Monolithic systems (running with a single ZK framework) can take advantage of batching transaction execution and amortizing costs related to dual-layer ZK context creation across multiple transactions; however, the main drawback of this is:

1) Being locked within a single ZK framework
2) Batching multiple transactions makes the processing pipeline not reorg-friendly. Reorg friendliness can be established only when transactions are processed individually (see Execution Cache and Reorgs section for more details).

3.2.1 Dual-layer VM properties

- ⚠ x5-x50 slower execution times
- ⚠ x5-x30 larger traces
- ⚠ Requires ELF to be loaded into zkVM as a private input (for inner VM interpretation) - highly penalizing during transaction processing.
- ⚠ Flexible native RISC-V or other ISA interpretations, but programs themselves have no proofs (everything is a single VM proof - does not fit within the goals of ZK composability - not standalone ZK programs)
- ⚠ Faster CPI composability (syncomp), but for each CPI, each program ELF has to be loaded into the ZK context as ZK input
- ⚠ Doesn't allow composability between different zkVM platforms
- ✓ Provable termination condition and gas accounting (albeit provable gas accounting by itself typically increases cycle consumption by ~x5)

3.2.2 Single-layer VM properties

- ✓ Significantly faster (x10 slower than bare metal (native CPU execution) - very acceptable)
- ⚠ ZK program is "naked" and runs directly under zkVM - requires external validation, handling of panic, and metering overrun conditions.
- ✓ Single-layer zkVM allows for dual-layer nested zkVM execution as well.
- ⚠ Most zkVM frameworks (RISC-0, SP1, JOLT) are incapable of proving actual compute consumption (GAS metering); however, gas can be statically limited, proving the termination due to the limit overflow.
- ⚠ In the EPA model (described in 3.5), the Single-layer VM needs to be proven by a trusted program (EPA) - this requires the proof to be loaded into another ZK

program, resulting in potential costs similar to ELF loading in the dual-layer VM model (but still significantly lower overhead due to the single-layer processing).

Single-layer VM execution pattern provides for a significantly faster execution environment, but requires external provable validation via ZK execution composition.

It is important to note that a single-layer VM model does not disallow a dual-layer VM model. In fact, by externalizing validation processing, it is possible to make dual-layer VMs simpler.

3.3 Hard-fork considerations and framework integration

One of the main problems with ZK verification and embedding ZK verification into L1 consensus is that each ZK framework addition or *a major version upgrade* requires an L1 hard fork.

Due to the decentralized nature of PoW networks like Kaspa, L1 hard forks are extremely challenging as they require the entire ecosystem to coordinate and upgrade all running node software.

There are multiple ways to augment this problem:

- Addition of a very limited high-performance RISC-V or WASM VM in the L1 scripting system (very complex, but provides other benefits)
- Using the EPA (intermediary proofs) to isolate different ZK frameworks (L1 must support EPA program hash upgrades).

Both variants can be viewed as “deployable firmware” upgrades.

Both variants simplify and automate network upgrade ability, but simultaneously carry undesirable centralized control properties.

To mitigate centralization properties, while the data itself can be deployed “on-chain”, miners can vote by explicitly abstaining, allowing, or disallowing a new version (via command-line arguments) without the need to upgrade the node software version.

The interesting part of this approach is that if the voting minority is “no”, instead of ending up in a stale/split/forked network, they will be forced to upgrade.

3.4 RISC-V or WASM VM as an L1 script opcode

L1 can integrate a custom miniature / lightweight high-performance RISC-V or WASM VM that can be accessible from consensus (scripting engine) with the sole purpose of executing ZK proof verification code compiled for RISC-V or WASM ISA (CPU target). A new version of such firmware can later be published by developers and proposed to the ecosystem for activation via PoW voting.

The general concept revolves around the introduction of a new OPcode (OP_EXEC) in the L1 scripting engine that allows scripts to execute a dynamically-registered set of functions, earlier published as standalone ELFs or WASM binaries.

The counterargument for venturing on such a complex endeavour is the fact that zkVM frameworks can be abstracted by the execution proof aggregator (EPA). In the EPA model, EPA is the only ZK program that needs to be aware of different proof verification schemes and can act as a proof adaptor.

The vProg executor (L2) needs to be aware of different zkVM frameworks because it needs to execute programs. However, that is an “L2 problem”, and that complexity should be isolated from L1.

3.5 Execution Proof Aggregator (EPA)

Execution Proof Aggregator (EPA) serves the purpose of the program execution conformity validation.

It is important to outline that, unlike vProgs, the EPA is a trusted ZK program (aka "system program") - it is an extension of L1.

Since EPA is a trusted program, a valid EPA proof guarantees that the user program execution is valid for a given transaction, including all CPI/syncomp invocations of that program. The only undetermined component of an EPA proof becomes the account state input of the previous account transform, meaning that the EPA generates partial (conditional) proofs compatible with the vProg CDAG vertex specification.

It is important to note that due to CPI/syncomp handling, EPA generates a single proof representing a transaction proof. Unlike the draft vProg design described in the Yellow Paper, this unifies multiple vProgs for synchronous composition and significantly simplifies the interaction with L1.

In the EPA model, while L1 needs to know and coordinate vProg program hashes, L1 needs to verify only the EPA. EPA can function as a proof adaptor, capable of verifying proofs from different ZK frameworks. This, however, requires the EPA to be upgradable.

As described above, the EPA can be upgradable via PoW voting.

NOTE: EPA is one of the largest departures from the vProgs Yellow Paper design. Without the EPA, it is impossible to facilitate various functionality described in this document, including, but not limited to, transfers of KAS, the requirement of vProgs to publish all account state transitions as L1 transactions during CPI/syncomp, ZK prover payout mechanics, etc. The EPA does not take away from the vProg Yellow Paper design - it is an intermediate trusted ZK circuit that still facilitates all the same functionality; it simply acts on behalf of the L1.

NOTE: The EPA design originates from the Sparkle framework. The key difference from vProgs is that EPA can carry multiple program (vProg) inputs and outputs, as it can aggregate CPI / syncomp. To adapt this to vProg design, the EPA proof handling needs to support processing against multiple vProg account mutations. This can be done by associating a copy (a reference) of the EPA proof with multiple vProgs (already done implicitly by the invoking transaction) and a mapping of which account belongs to which vProg.

NOTE: vProg CDAG "Stitcher" is also a trusted ZK program that has to be mated directly to L1. In fact, the Stitcher and the EPA can be a single program following a single deployment upgrade path.

4 Resource Utilization

4.1 GAS metering

Gas metering is an extremely complex problem in ZK.

In most zkVM platforms, gas metering is handled by the zkVM itself, meaning that it is not provable. Cairo is currently the only zkVM platform that implements provable gas metering.

Gas metering is easily implemented within a dual-layer zkVM; however, due to per-opcode accounting, the metering itself can result in ~x5 overhead in cycles and the execution trace size.

While gas metering is unprovable in most systems, the program termination due to gas limit overrun can be proven. Due to the deterministic nature of vProg execution, a termination of a program due to gas limit can be observed during regular (unproven) execution. This, in turn, can trigger an execution of this program inside of a dual-layer VM proving its failure.

A number of networks (including Solana) use a fixed gas (compute budget) fee per transaction. They allocate a specific limit of compute per transaction (like all networks) but charge a single fixed fee for execution regardless of the compute budget consumption.

4.2 Rent (storage)

State (storage) rent has different properties from gas.

The primary goal of gas is to account for program processing, or in the case of vProgs, for proof generation compute costs. Gas is used to establish the amount of fees that need to be collected for the execution of a program, while the price of gas (or gas limit) is used to throttle program execution. Throttling indirectly via gas price or directly via gas limits serves multiple goals that include equal opportunity for program execution, transaction pressure control as well as incentivizing the network operation or proof generators.

In contrast, rent is a per-byte cost of state storage. UTXOs in UTXO systems are very similar to account rent as their existence implies retention of some monetary value, indirectly protecting the network (UTXO state) health.

Rent functions in a similar manner. It continuously discourages a dormant or abandoned state. Keeps the state size bounded and incentivizes cleanup.

Note that vProg CDAG state vertices are similar in their nature to UTXOs, and their existence within L1 also requires storage.

Similar to gas, rent should be considered as a secondary dimension that is retained within the vProg account (and can be recuperated at the behest of the vProg). The presence of a minimal account balance proportional to the account size can function as a rent deposit.

5 Ledger Functionality

5.1 KAS transfers

vProg specification currently does not define mechanisms for transferring funds (Kaspa) into the vProg ZK context, nor withdrawing it from the vProg ZK context. There is also no specification on how vProgs can control KAS or transfer KAS to provers.

A number of patterns can be used to achieve transfers between L1 and L2. These patterns include:

- Retention of KAS within UTXOs (aka wKAS (wrapped KAS)) and subsequent control of this KAS using proofs.
- Consumption of transaction UTXO value and subsequent re-creation of value using ZK proofs (this methodology can be viewed as unsafe by some, see below).

It is highly desirable to associate KAS balances with vProg accounts as this allows vProgs to transfer balances among vProg invocations completely "off-chain", without any participation of L1.

Furthermore, by coupling account states together with KAS balances with the L1 CDAG processing infrastructure, it becomes possible to perform provable KAS accounting of L2 transfers within the consensus context of L1.

Coupled with the EPA (supervisor) concept, CDAG state commitments can have associated account balances extracted from vProg execution proofs. Simply adding the "amount" field to Kaspa CDAG state vertices, updated via proofs, facilitates that.

A proof generated by the vProg can publish a specific/dedicated output that denotes gas and fee processing requirements. Such output can also be considered within CDAG accounting.

Due to security considerations and possible future discovery of ZK vulnerabilities, a combination of UTXO retention and CDAG accounting would be desirable. EPA is an immutable provable observer; however, like all ZK systems, it can become susceptible to proof forgery. UTXO balance retention can mitigate this vulnerability if such a vulnerability is discovered in the future.

5.2 Token transfers

A token is a simple program that maintains its balances in separate user-associated accounts. Invocation of such a program directly or via syncomp (CPI) facilitates token transfers between accounts.

Such a design allows asynchronous transfers between two independent parties and only requires synchronous processing when parties are related. (i.e. A to B can occur in parallel with C to D, but A to B and C to B must be sequential).

A token program can exist in different forms. Each token can be a separate vProg deployment, or each token can have its own ID, where issuance and transferring of such a token is managed by a single vProg.

Regardless of the patterns used, there will be different vProgs implementing token management functionality. As this is a constantly evolving ecosystem, in the future, developers working in this area will deploy vProgs implementing various features, such as different types of privacy, etc.

6 vProg Management

6.1 vProg registration and upgrades

There exist two models for program upgrades:

- non-upgradable (Ethereum)
- upgradable with an option of upgrade ability revocation

The main question that arises on this subject is: if a program is upgraded, what happens to existing account data (account ownership)?

It is beneficial (from the standpoint of system architecture) for the vProg account space to be monomorphic, i.e., the vProg executable data (ELF or WASM binary) and vProg account data to be represented by a single account data structure.

Such a data structure can have the authority pubkey field that, in the case of a program account, denotes the upgrade authority of the program.

Within a CDAG, a vProg can be denoted as an account state vertex that follows account read semantics for each program execution. Such an account can contain metadata of the program containing multiple program hashes representing multiple versions of the program.

A special “program manager” ZK program (or a similar consensus-driven mechanism within L1 reacting to a special transaction opcode) can be the only component authorized to gain write access to the program metadata account for the purposes of 1) adding new program hash (new version) 2) removing a program hash (deprecating a version or purging the program metadata entirely) 3) revoking or changing the upgrade authority

6.2 vProg loader

Given that vProg design allows for complex modifications of consensus, a number of pathways are open for publishing vProgs on the network:

- vProgs can be published via a program loader (a special system-level ZK program) that receives transactions containing chunks of an ELF, then assembles these chunks into a single final ELF. Such a program can then be the sole program on the network that can change an account type from a regular data account into an executable account.
- L1 can support the *publishing of the program hash* via a custom opcode execution. Such an approach does not provide for a method by which an ELF would be made available to all vProg nodes on the network and requires a separate pipeline to publish the actual program ELF.
- L1 can support uploading of the ELF without ZK proofs.

The first approach is **prohibitively expensive** as the entire process needs to be proven, whereas if the program publishing and registry is supported by consensus (2nd+3rd approach), it would be **significantly faster**. L1 only cares about a program hash registration; hence, how the vProg program binary is handled by the vProg node is a prerogative of the vProg node. A custom payload-based protocol can be developed to deliver ELF chunks to the vProg layer (L2) coupled with subsequent program hash registration.

It is important to note that it is desirable to allow L1 to function as a completely isolated sequencer where the content of the ELF as well as its execution are done completely “off the network”. I.e., where L1 functions as a completely blind sequencer, allowing for 3rd parties to use it as a sequencer without exposing any information about what is being sequenced, leaving it to be the responsibility of the vProg creator to prove vProg execution.

6.3 vProg data exchange protocol

Note that, unlike the vProg Yellow Paper draft v0.0.1, this specification requires account data to use public inputs and outputs. In this design, it is the EPA program that accepts program data as private inputs, coupled with public input account commitments. In turn, EPA produces new account commitments. If EPA verification passes, the output data of the vProg can be considered as valid.

6.3.1 ZK Inputs

- Transaction instruction inputs (public, arbitrary raw data)
- Incoming account data inputs (public, raw data)
- Execution context (txid, stable RNG seed, gas info (TBD))

6.3.2 ZK Outputs

- Transformed writable accounts (raw account data)
- Events (arbitrary binary payloads prefixed with event index)
- Processing metadata (gas indicators - TBD)

6.3.3 Non-ZK outputs

- vProg execution logs (trace of all CPI/syncomp data)

NOTE: Account states can be provided via private input and obtained via ZK IO channels (see 6.8 Account Data Exchange).

6.4 vProg validation protocol (EPA)

Once proven, vProg execution (including CPI (syncomp)) is supplied to the EPA program for validation.

ZK Inputs
- vProg proof (containing raw data inputs and outputs listed above)
- vProg program hash
- Incoming account data commitments
- Prover payout address

ZK Outputs
- Outgoing account data commitments
- vProg proof commitment (TBD)
- Event commitments
- Processing metadata commitments

vProg execution is considered valid only if EPA generates a proof attesting to its compliance.

6.5 Events

Events are protocol-driven, arbitrary binary payloads constructed by vProg and published as outputs, allowing clients to observe vProg activity.

Events provide a purpose-driven compact data exchange mechanism where each program can publish its actions without the need for integrators to monitor and deserialize account state data.

Events are used in EVMs and require additional considerations, such as compute (gas) and bandwidth use.

An EPA proof output contains account state commitments. It can also carry the event data commitment, making events provable to an external observer.

Event exchange protocol can be a binary payload comprised of multiple event records prefixed with a vProg-defined event ID. An RPC subscription system can provide clients with the ability to register for events and filter such event deliveries by event IDs.

6.6 Interoperability

vProg can receive custom data via its transaction execution arguments. Such data can include PKI authority signature (oracle-like data exchange) as well as a ZK proof verifiable by the vProg.

vProg itself can publish events that will be considered valid once proven, and once they are sufficiently protected by the PoW security (i.e., accumulated a sufficient number of confirmations).

6.7 Random number generation (RNG)

Programs such as games require random number access. Due to this, best efforts should be made to study the subject in greater depth in an effort to supply vProg executions with unpredictable, yet deterministic seed for the random number generation.

A XORed composition of various components can potentially satisfy stability and uniqueness requirements. Such a composition can use the following components to generate a seed:

- Transaction hash shuffling where the shuffling order depends on some semi-stable random value (such as a selected block hash N DAA blocks ago)
- XORed combination of block hashes N DAA blocks ago (deterministic sampling of BlockDAG block hashes in the past).

Transactions intended for vProgs can carry various configuration options specifying which additional environmental information (such as RNG) should be supplied to the vProg, as long as this information is stable in relation to the BlockDAG formation and execution cache pipelines.

It is important to note that an integrated RNG is fine for the majority of applications; however, anything high-value must use VRF or a combination of RNG with a commit/reveal scheme.

6.8 Account Data Exchange

Special consideration needs to be made in relation to the vProg programs' publishing account data.

A ZK program is a deterministic program; this means that it runs in the same way in unproven and proven modes.

vProg must have the ability to mutate accounts. To mutate accounts, it needs to be able to publish raw account data.

There are two ways raw account data can be published. The first approach is using public outputs of the proof itself. This unfortunately injects account data directly into the proof itself, potentially making the proof very large.

The second approach is to publish data using auxiliary unproven channels known as *Logs*, *I/O,* or *hints (hereafter referred to as IO channels)*. Such channels allow ZK frameworks to write unproven data during their execution.

A commitment to this data can then be published via the public output (this is known as a *private output*). This technique is frequently used to reduce resulting proof sizes, as the commitment can be used to verify the program data created via the above-mentioned side channel.

There is, however, a possible contention issue here as vProg developers need to be able to use *IO channels* to output various useful debug or application log data.

There are two ways to mitigate this:

- Requiring developers to use structured data output. Helper functions can be provided to help the executor running the zkVM distinguish between types of output (by prefixing the data type). Such types can be logs, events, or account data (the output must be binary).
- The EPA model solves this by enveloping (recursing) the vProg proof, allowing vProgs to use public inputs and public outputs freely, isolating the IO channel subsystem.

7 State Management

7.1 State domain separation

(ref vProg YP 5.1 - 5.3)

This subject requires additional research and discussion.

It is important to note that L1 CDAG allows each vProg to execute independently (isolated) from one another, linking vProg executions only during composability.

This means that a set of vProgs A can operate completely independently from another set of vProgs B as long as these vProgs do not have any type of composability. This means that state retention A and B can be completely isolated from one another.

Section 5.3 of the vProgs Yellow Paper describes whitelisting of vProgs, a concept that allows a vProg to declare a set of vProgs it is willing to interoperate with. Such a whitelist can be introduced as and referred to as a “vProg composability intent” included within the "vProg metadata" structure published alongside the vProg itself.

vProg descriptors, once processed recursively, can be used to create a vProg dependency tree. This dependency graph can be used to instruct the vProg node (L2) to perform state retention *only for vProgs identified within this dependency tree*.

The problem arises when a vProg wants to interoperate with any other vProg. Such vProgs can be, for example, token programs. Token programs are meant to be composable with any program that wants to transfer tokens.

The problem becomes the fact that the token program can not declare its intent - it is meant to be interoperable with any program. (In general, composability intents can be a complicated subject due to the fact that programs can know who they will use, but can not know who will use them.)

There are multiple ways of handling this challenge:

- Await for a proof of transaction that refers to the missing data.
- Use a gossip protocol to obtain missing data via the p2p network.

7.2 State compression and transforms

Programs usually store their data as uncompressed serialized data structures (serde, borsh, etc.). In the ZK domain, it is not possible to obtain traditional data compression benefits, as any form of data compression requires more cycles to compress or decompress than raw data processing. (Some LUT embedding approaches can be used for custom data encoding, but have a very limited scope.)

While cryptography and blockchain-related data (such as keys, hashes) are not susceptible to compression due to the inherent entropy of the data, program data structures are. Hence, the external account storage can be managed as follows:

1) Account states can be stored as RLE encoded data (RLE is very efficient)
2) State transforms can be stored as “binary diffs”, reducing both the size of the state and making state transforms more efficient.

Diffs can provide significant bandwidth economy benefits during p2p sync as well as in the client delivery context. Any external observer can subscribe to an account transform feed, and once the local state is established, it can receive only diffs representing state changes.

7.3 State retention

One of the main challenges in integrating with Kaspa is the reorg handling mechanism.

Kaspa is a very high-performance system capable of handling thousands of TPS. This results in a significant amount of data in the form of transaction and block history. Likewise, this can result in a substantial amount of data processed by vProgs.

Kaspa PoW BlockDAG mechanism has the following properties:

- The Kaspa BlockDAG undergoes frequent reorgs on the new block arrival (at the tip of the DAG).
- The BlockDAG typically settles within ~10 seconds, but depending on network conditions, reorgs can occur deeper, but rarely do.
- There exists a DAA window (approx 44 min). The network will typically reject blocks submitted that are older than the DAA window (unless a specific flag is set during submission)
- Blocks with a cumulatively lesser proof-of-work (than the current tip of the BlockDAG) will be merged as long as they are within the merge depth bound, and their transactions will be executed after they are merged, following the global ordering induced by GhostDAG, not in the order they were originally mined.
- A “finality point” (a system-defined constant, approx 12 hours in the past as of the Crescendo update) acts as a permanent checkpoint. A reorg is no longer possible past that level. Nodes will actively reject any attempt to reorg below that point.
- A “pruning point” is a data deletion period (approx 30 hrs) past which the data is considered irrelevant. It is a default data retention period.

The probability of reorg decreases significantly within a matter of seconds. However, network partitioning due to global outages can split the network, resulting in reorgs that can be a few hours deep (up to the 12-hour finality point).

A state management subsystem needs to understand and respect these mechanics. This means that state history must be present and available up to the finality point of the Kaspa node so that, in case of a reorg, the vProg node can revert and reconstruct the state based on the new BlockDAG layout.

There exist several mechanisms that can help optimize this type of data management:

- Periodic full-state snapshots (checkpoints, aka keyframes)
- Diff-based state transform retention (in a temporal space close to the tips)
- Program replay (full program re-execution)

State distribution can be arranged as a series of periodic state snapshots, backed with state change diffs in periods closer to the tip. If diffs are not available, the state can be re-created via transaction-driven program execution replay.

Transaction data is already available on the local Kaspa L1 node; hence, transaction replay (access to the transaction data) is feasible if needed. Once the state crosses the finality point boundary, the vProg node state can be permanently erased.

The state retention must be closely integrated with a prover subsystem (covered in section 9 Prover Subsystem) and handle asynchronous prover job creation and cancellation in close cooperation with the execution cache (described next).

7.4 Execution cache and reorgs

A universal FIFO memory-bound (or TTL-bound) cache can be introduced to help accelerate reorg processing.

A reorg can re-arrange transaction execution order; however, this does not mean that some transactions should be invalidated. In fact, if transactions are composed against different accounts (executed in parallel), their order of execution in relation to one another is irrelevant; hence, transaction reordering has no effect.

The following components represent a unique ordered transaction execution:

- Transaction id
- Account commitments (account state hashes) for accounts used in this transaction

A cryptographic commitment (hash) of these components can be used as a cache key, where the cache value is a set of resulting post execution account states.

For each transaction execution, such a key can be created, and the cache checked for the existing entry. If such an entry exists, account states present in it can be applied directly to the current account state. If such an entry does not exist, vProg can be executed, creating a new cache entry.

Entry cleanup can occur based on the entry TTL or a FIFO memory bound.

Such cache can also effectively tie into the proof generation feed.

7.5 Client delivery protocols

7.5.1 Trim Protocol (erase + re-execute)

A client interested in observing state changes (whether state changes denote account data changes or events), can use a "trim" approach for data ingestion.

Each message delivered to the client must be tagged with some sequence marker (such as DAA score or some other sequence value that can serve as an anchor to reorgs).

If a reorg occurs, a reorg message (aka "trim") must be sent to the client indicating that any data since this marker must be discarded, following that, the node must re-transmit the newly updated data since this marker.

The trim message can include a full account state, whereas all transactions following that can be transmitted as diffs.

7.5.2 VCC protocol (forward changes)

VCC (Virtual Chain Changed) is a protocol that delivers state changes for any subset of changed data (used by Kaspa to deliver UTXO set mutations).

This approach can also be used, but it is different from the above-described trim protocol because VCC is assumed to indicate current or the latest change without engaging in a full transactional replay.

When it comes to a full transactional replay (for example, DEX activity), external observers may want to observe each program execution. For example, an external algorithmic trading system may want to observe a temporary DEX token price spike. The only way to reliably do that is by making sure that the client receives the results of all transaction executions.

8 CDAG implementation

8.1 Internal processing

L1 CDAG can be implemented in two ways:

- A completely independent data structure that is adjacent to L1 consensus state processing, with the ability to interoperate with the UTXO set.

- CDAG can also be implemented on top of the existing UTXO set and UTXO state management.

The first variant (standalone subsystem) requires the design of a separate subsystem, data structures, synchronization mechanisms, and interoperability mechanisms. (Following additional analysis and review, this appears to be the best and cleanest approach as it does not affect existing SDK data structures).

The second variant requires customization of the UTXO data structure, introducing a new variant of the UTXO. This variant allows vProgs to capitalize on the existing UTXO processing infrastructure.

NOTE: The topic of the UTXO reuse described here may be confusing due to various overlapping concepts. What I am referring to is converting the UTXO data structures to an `enum UTXO { OriginalUtxo { … }, CdagVertex { … } }`*. The complexity of this approach does seem to be quite high, but nevertheless, I am leaving the material below for reference.*

At its core, a CDAG is a graph of state transform dependencies. This graph has very similar properties to UTXO-driven transaction processing.

This subject requires additional research, but the benefit of re-using the UTXO subsystem (if feasible) would allow the implementation to capitalize on the existing UTXO processing infrastructure (including the existing high-efficiency multi-threaded transaction processing, UTXO set state synchronization, etc). CDAG vertices have different properties, but it would be worthwhile to understand whether the existing infrastructure can be reused (and if not, perhaps some of it can be mirrored when creating a dedicated implementation).

NOTE: If such an approach is considered, special attention needs to be paid to the possible performance impact on the existing infrastructure and the impact on the 3rd party integrators. Any changes to the UTXO subsystem may require 3rd party systems (explorers, wallets, SDKs) to update their processing pipelines. If such an approach is considered, it might be relevant to consider filtering out vProg-related (CDAG) data.

On the other hand, as described above, if CDAG vertices carry account amounts alongside commitments, they become very similar to UTXOs - effectively, they become "L2 UTXOs".

A vProg transaction that specifies N accounts that are readable (R) and writable (W) can be expressed as a transaction that requires a set of R[]+W[] inputs and produces a set of W[]

outputs. The sequencing of such processing and the UTXO CDAG vertex processing can be derived from a mergeset providing a deterministic set of UTXO mutations.

However, unlike the UTXO subsystem, CDAG needs to be able to execute forward processing without proofs. I.e., a local node processing vProg transactions needs to be able to track the CDAG data set forward, without proofs. This could be the main differentiating factor, making such an approach not viable. It is also possible that the vProg node should perform its own unproven pre-processing, where UTXOs are only processed and collapsed upon receipt of proofs. (The subject of unproven processing is not covered in the vProgs Yellow Paper and should be discussed separately to see which subsystems and data structures can accommodate this).

The argument against this could also be potential performance loss due to storage or lock contention, or other side effects. The goal of raising this topic for discussion is to understand if any potential benefits can be obtained from reusing the existing infrastructure (database, p2p sync).

8.2 Resource starvation (unproven vProg node operation)

Given the deterministic nature of ZK program execution, a vProg node can operate locally without ZK proofs. Like a regular indexer (similar to KRC indexers), such a node will have a viable and provable state derived from L1 consensus. (The only problem with such processing is the inability for the indexer to prove its state to any external observer - the very problem solved by ZK proofs).

Proofs are used to inform L1 consensus and 3rd-party observers of the fact that a specific set of operations (vProg executions) has occurred.

However, there is nothing preventing a user from running a local node without any proofs, processing transactions, and trusting the deterministically consensus-derived state.

A standalone DEX node can observe user activity, update user balances, and even issue vProg-based withdrawal transactions without any use of ZK proofs.

Logistical benefits of such a setup are huge, as this allows complete decoupling of centralized service-oriented infrastructure (by creating multiple high-performance redundant node instances) from the ZK infrastructure.

However, such a standalone execution processing model becomes subject to the “resource starvation problem”.

If proofs are not generated fast enough, CDAG and vProg state maintenance can fall behind, potentially resulting in a large amount of data storage that can ultimately lead to the system running out of free storage space.

Given these considerations, it makes sense to consider making efforts toward decentralized proving (described below) as the only viable solution to reliable and censorship-resistant ZK proving.

8.3 Proof availability and prioritization

Partial (conditional) proofs (proofs created by vProg or the EPA) can be proven in parallel and delivered to L1 out of order.

Proof suppression is a condition that can occur during high mempool pressure if the network enters a prolonged period of high-frequency transaction issuance.

If proofs are submitted via a standard mempool, they become subject to the fee market.

CDAG must be capable of receiving and accumulating partial (conditional) proofs out of order and retain them (or otherwise mark a subset of CDAG as proven) until a full sequence of proofs comes in, even if a stitching proof becomes available (the entire proof processing mechanism must be async-aware - even if all proofs are pre-processed and posted to L1 as a batch, their delivery can be out of order).

Depending on the CDAG implementation, there can arise conditions where, due to proof withholding or due to mempool pressure, a proof-proving transaction thousands of transactions ago may not be available. Such conditions can result in OOM (Out of Memory) or “out of storage” conditions. I am just raising these topics as the vProg Yellow Paper does not account for the asynchronous nature of the data delivery (which makes sense for simplicity purposes).

To mitigate proof availability concerns, the Kaspa node can employ the following techniques:

- Exempt transactions containing verifiable proof from fees.
- Always treat proof transactions in the mempool as high-priority.
- Utilize a secondary mempool-like data-availability channel for proof delivery (highly desirable).

Unlike regular transactions, proofs have properties similar to Blocks in that they help the network reduce the footprint of the CDAG and provide confirmations to the user of L2 transaction execution.

Unlike regular transactions, proof validity can be verified at the RPC and p2p layers, preventing data relay flood attacks without fees (this type of validation is similar to transaction integrity validation).

One can view this problem as a "building entrance problem". If you have a skyscraper housing many people with a single door, people entering and exiting (especially during peak hours) will be constantly bumping into one another. If you create two doors, one marked "entrance" and another "exit", the foot traffic will be much more fluent. The same principles apply to network data flows.

8.4 Stitching

(ref vProg YP 4.2)

The addition of the EPA changes the validation pattern of vProg execution. If the EPA model is adopted, the stitching algorithm can potentially be simplified by offloading some of the pre-processing into the EPA program (and thus parallelizing it).

In general, if any of the features described in this document are adopted within the vProg infrastructure, the pseudocode in vProg YP 4.2 must be reviewed and possibly altered, accounting for subsystems like the EPA.

9 Proofs

9.1 SNARKs vs STARKs

SNARK proofs are a few hundred bytes in size, while STARK proofs are 30-200 kilobytes. SNARK proofs require various types of setup ceremonies; STARK proofs do not.

SNARKs are frequently used to recursively compress STARKs. However, SNARK proofs are expected to be non-quantum resistant (non-PQ-resistant).

Generally speaking, any long-term forward-looking platform should consider bandwidth and compute performance to be scalable over time. Given such an approach, vProgs can adopt STARK proofs. Such proofs require an increase in the block mass limit. (NOTE: like storage mass, transient mass, etc., there can also exist a "proof mass").

NOTE: SNARK proofs also capitalize on the fact that they prove data commitments. This means that the data itself needs to be exchanged with whoever needs to validate the proof integrity. In a vProg CDAG context, this can be a problem as the L1 node would need to receive inputs and outputs of the underlying compressed STARK proof.

vProgs must receive and transform account data. While they can receive account data via a private input, they need to publish it via the public output. EPA model further compresses this by creating a proof containing only input and output commitments. Without the EPA, SNARK-compressed STARK proof of the vProg execution needs to transmit full account data to L1 for validation. With EPA, the footprint of the data can be significantly reduced, but generally speaking, due to the other above-mentioned considerations, SNARKs should be avoided.

Another factor to consider is that STARK-specific VPUs (hardware-accelerated provers) are under development and may eventually become feasible and accessible to consumers. It is unlikely such hardware will support SNARK proof generation.

9.2 Proof compression

Since STARK proofs are relatively large, in the near future, they can impede the network processing performance. At a cost of some latency, the following approaches can be taken to optimize proof sizes:

- EPA can be generalized to carry proofs of multiple independent transactions.
- L1 can accept proofs from an “EPA aggregator” program proving multiple disjoined transactions. Incentives for creating such aggregations can be derived from the underlying vProg payments, allowing a prover creating such an aggregation to tap into the portion of the prover fees.

The problem with such methodology is that this can result in overlapping proofs. This can be solved by forcing proofs to be window-aggregated (per mergeset) or per a group of mergesets (this can be further combined with the PAP concept described in Section 9.5).

10 Prover Subsystem

The prover subsystem is a very complex topic that needs to have a separate discussion. This section attempts to outline some of the key concepts related to this subject.

Depending on imposed compute budget limitations prove traces can be very large and measured in megabytes. Bandwidth consumption and GPU requirements imposed by such traces can be significant. Use of different ZK frameworks and large trace sizes warrants the design of different subsystems that aim to generalize, optimize, and localize proof generation processing where possible.

10.1 Proof envelope protocol

Given the fact that different types of proofs are required by the network (EPA, vProg, different ZK frameworks), it makes sense to create a standardized trace exchange envelope. Such an envelope can be an arbitrary data structure that contains a header and a payload containing a trace. The header of the proof envelope can contain metadata (header) that identifies the following properties:

- proof type
- transaction id
- prover kind (Cairo, RISC-0, etc)
- prover version

The metadata can be followed by the proof-related data:

- vProg execution trace (includes ZK inputs and outputs)

The information needed to generate an EPA proof related to the vProg execution (EPA inputs and outputs) can also be supplied within the envelope. vProg proof must be generated in order to create an EPA proof; hence, due to “data proximity”, it makes sense that the prover generating a specific vProg proof immediately switches to the related EPA proof generation.

It is important to note that in the EPA model, vProg proof contains raw account inputs and outputs; hence, the “data proximity” consideration can yield a significant performance optimization.

10.2 Trace generator and distributor

There exist different types of prover configurations:

- decentralized prover networks (remote GPU clusters)
- proofing as a service networks (remote GPU clusters)

- GPU mining farms (a set of local GPU clusters)
- standalone GPUs

A vProg node operating in a prover mode can create traces for all vProg executions. Such proofs can be processed in parallel.

In a local GPU cluster setup (a mining farm), it makes sense to have a client-server daemon architecture where a vProg node produces proof traces. These traces are then picked up by the “proof coordinator” (a server) and are then distributed to local clients (GPU clusters).

A GPU cluster is typically a lightweight computer that delivers / relays data to a number of physically attached GPUs. Such computers are typically cheap and do not satisfy the MSR (Minimum System Requirements) for the L1/vProg node.

In this pattern, a custom GPU-aware proving software (a client) can receive trace packages, prove them, and submit them back to the server. The server can then relay them back to the node for CDAG processing and p2p relay.

Proving vProg Node <-> Coordinator <-> Provers

The coordinator daemon is best suited to be a standalone proof job distributor that can be configured to interface with dedicated prover instances, prover networks, etc.

Existing mining pools can run coordinators, allowing miners (clients) to run provers on existing mining farms, same way as one would run PoW mining (albeit proving networks require separate considerations, as unlike PoW, where throttling can be achieved via difficulty adjustment and probabilistic accounting, such mechanisms do not apply to proof generation).

A custom proving software is needed to ensure that different types of proofs (Cairo, RISC-0, etc) can be processed by the same software.

10.3 Decentralized proofing

What happens when no one submits a proof for a specific vProg execution?

What happens when two different provers submit the same proof - who should be compensated for the work done?

The vProg Yellow Paper externalizes these challenges.

If all provers have equal opportunity to “fetch jobs” from the network and generate proofs, a lot of concerns, such as proof withholding/censorship, disappear.

In my personal opinion, it is a mistake to view a vProg as an “L2 subnetwork where the responsibility for this subnetwork operation lies with the vProg maintainer”. (I have heard such views multiple times in the past.) If vProgs are to embrace decentralization, there must be a strict separation of concerns:

- vProgs are published by creators.
- proving should be generalized, allowing anyone to contribute their computing power to the network as a service.

In many ways, this problem is reminiscent of the “if you want to make a service, you should run your own node” problem. Software developers can create software; many can do that as a contribution, but a single node is capable of processing 500-700 clients, requiring any application that needs to interact with a large number of clients to run multiple nodes. Doing this can be not affordable to the program creator.

Same principles apply here - if I create a vProg, I want to deploy it and not care about its state management or proof generation - these services should be facilitated by the ecosystem.

10.4 Prover payout mechanism

This subject requires additional research.

One of the vProg outputs can contain the desirable payment amount intended for the prover.

The prover payout address can be included by the prover in the EPA proof as a public input. (Since the payout address is not deterministic, it can not be included in the vProg proof, but it can be included in the EPA, as EPA inputs do not need to be deterministic).

When submitted to L1 CDAG, the payout address can be used to facilitate the payment to the prover.

How this is handled on L1 is a subject for additional research. The following sections describe two hypothetical concepts that can be employed to create a decentralized proving mechanism.

10.5 Payment aggregation prover (PAP)

This subject requires additional research.

One of the concepts that can be researched is the creation of a separate ZK system program (similar to EPA) that collaborates with L1 and is capable of receiving multiple EPA proofs, providing L1 with CDAG-style commitments intended for prover payments.

PAP could maintain a temporary (periodic) state and progressively receive EPA proofs while publishing CDAG-style commitments that can be understood and conditionally accepted by L1.

PAP could handle payment distribution to provers. In particular, it can receive multiple proofs for the same transaction (created by different provers) and distribute earnings between these provers for the work done.

L1 CDAG needs to accept and react to the proof immediately to facilitate CDAG and state maintenance; however, payments to ZK provers can be delayed.

PAP can be comprised of two subsystems:

- PAP accumulator (a vProg that collects payments from all vProg proofs into its account)
- PAP distributor - a module within L1 that receives PAP proofs, validates them against the L1 state (CDAG history, related transactions, etc), and allows this proof to be processed only if it contains valid data.

A PAP proof can be externally constructed (outside of the vProg subsystem) using the PAP accumulator and submitted to L1. Since PAP is a system program, L1 can have explicit consensus rules forbidding any transactions against it unless they are validated by L1.

10.6 Proof job distribution

This subject requires additional research and game theory analysis.

The above-described PAP concept allows for decentralized permissionless proof job distribution.

If we assume that under 1,000 TPS, we yield 1,000 proof jobs per second, all these jobs can be processed in parallel. If a proof takes 1 second, 1,000 GPU clusters can process all 1,000 proofs within a second (discarding data transfer and other latency penalties). The problem in this case becomes the job distribution mechanism.

One of the approaches to facilitate job distribution in a decentralized environment can be a similar methodology to the one employed by the Kaspa mempool processing: an order-biased random job selection backed by a real-time p2p gossip protocol.

Such a mechanism will result in proof overlap, which can be mitigated by the above-described PAP program.

The benefit of this mechanism is that it ensures decentralization and censorship resistance for vProg proof generation.

11 Data Access and Account Structure

11.1 External (client) data access

In many use cases, program data (accounts) can not be constructed out of arbitrary data. If accounts contain arbitrary data, it becomes impossible to index them in a database by any type of key. Key-based access is one of the key pillars (pun intended) for information lookup.

Main lookup mechanisms that are typically needed for account lookup are: program ownership (id of a program an account belongs to) and authority (optional - which user is associated with an account).

For example, a token program account can be accessed in a database via a lookup for a token program (ownership) and an authority (PKI public key of the account owner). This allows a wallet to instantly latch onto the user account associated with a specific token (token represented by a vProg instance).

Other (more computationally expensive) mechanisms can be data filters. For example: “give me all program accounts that contain a u64 balance above 100 at this specific data offset”.

11.2 Internal (program) data access

While external database-index account access is simple, a program also needs to be able to access (create, track, validate) accounts that belong to it in an efficient manner.

The industry standard for such data management is known as PDAs (Program Derived Accounts). Similar schemes are known as intent-derived accounts, module-derived accounts, and implicit accounts.

PDA scheme is based around using the program ID as a prefix and an arbitrary program-defined suffix as a seed to create a new identifier (typically a PKI-based pubkey).

Example:

GameTile(P,Q) -> Fn(vProg_id, P, Q) -> Account_id

where Fn can be any type of collision-resistant function, such as a public key generator or a hash function.

NOTE: In systems that use PKI-based account identifiers, PDAs use signature schemes capable of producing off-curve public keys (available in signature schemes such as ed25519). This ensures that no private key can exist for an account. Given that the Kaspa vProg system is not designed to allow users to create user-owned accounts, the off-curve methodology is not applicable. User association in vProgs can occur using the dedicated authority field.

11.3 Program account data structure

The above-described considerations include the following requirements for account management:

- unique account identifier
- database access indexed by owner and authority
- monomorphic account representation (program or data)
- different types of ZK frameworks (program kinds)
- ability for programs to transfer KAS
- need for L2 to manage account rent
- need to retain and manage arbitrary program data

We arrive at the account data structure that should contain the following fields:

- account id
- vProg id (owner)
- authority (user identifier - optional)
- account type - program (including ZK framework kind) or data

- balance - account KAS balance
- data - arbitrary data managed by the program

Within different infrastructure subsystems (such as state management), an account should also carry a hash (state commitment) composed of the above-listed fields.

11.4 Account creation

(ref vProgs YP 2.2)

vProgs must be capable of creating accounts. In the ZK program ecosystem, there is no simple mechanism for accessing external systems such as databases (and it is outright impossible in a single-layer VM that assumes execution within existing unaltered zkVM frameworks. At the same time, vProgs need to be able to manage large data sets.

Creation of accounts allows vProgs to manage large datasets by allowing vProgs to use accounts as data records (for example, each account created by a vProg can be associated with the user).

Examples: A token program can have a per-user account allowing users to transfer funds by supplying 2 user accounts. A game can have a world where each in-game tile is represented by an account storing all resources (buildings, tanks) on each tile.

Account creation and deletion can be facilitated by a vProg execution proof, provided that the proof follows account creation rules.

Account ownership should be a property of an account. A vProg needs to be able to receive an account and determine its owner, as otherwise it becomes impossible for a vProg to differentiate accounts it receives.

The vProg domain can be monolithic or modular (see 7.1 State domain separation). If a monolithic domain is used, then a vProg can simply create an account by publishing it as one of its public outputs. Due to scheduling semantics and the fact that the same account can be created by two concurrent invocations of vProgs, a non-existent account must be supplied to a vProg as a writable account. Such a mechanism prevents the vProg transactions scheduler from executing two vProgs creating an account concurrently.

If an invalid account creation is executed (account data is withheld during vProg execution) CDAG resolution should fail (proof would be rejected) as the previous account state will cause a collision.

If the vProg domain is modular (vProgs may not have direct access to each other), a global "system program" would be required. Such a system program (similar to EPA) would need to exist in L1, be accessible to all domains, and be capable of creating accounts on behalf of different vProgs (i.e., an owner of the created account would be initialized to the invoking vProg).

12 Project structure

12.1 vProg Framework

In order to have the path of least resistance for vProg adoption, the project should be implemented as an integral subsystem within the rusty-kaspa project.

This allows vProgs to capitalize on the existing project infrastructure, including the Rusty Kaspa framework as a whole and, more specifically, RPC and Rust/WASM/Python SDKs.

If the vProg subsystem is optional, following the Rusty Kaspa daemon upgrade, any node maintainer can choose to upgrade the VPS to a higher spec to accommodate vProg processing.

Likewise, following an upgrade, any integrator currently using one of the Rusty Kaspa SDKs would be able to immediately interoperate with the vProg ecosystem, significantly reducing the integration friction.

KIP-12 (Wallet Integration KIP) is already designed to accommodate interfacing with secondary networks. It was designed to be updatable to accommodate vProg-specific interfaces (such as subscriptions to vProg events, account state transforms, and fungible token standards).

For Rust-compatible frameworks (RISC-0, SP1, JOLT), it makes sense to develop a monomorphic vProg SDK that can allow for a standardized vProg instrumentation. This includes declaring entry points, receiving account data, publishing account data updates, CCI/syncomp invocations, etc. Due to different ZK-optimized syscalls, not everything can be standardized, but the above-mentioned elements, IO channels, and framework-specific account data deserialization can bring benefits and greatly simplify the developer experience. Cairo would require a separate SDK implementation.

12.2 Core system components

The following list contains most of the primary components involved in the construction of the vProg ecosystem:

- CDAG state processing
- CDAG p2p sync
- CDAG unproven state management
- CDAG maintenance
- vProg scheduler (mergeset transaction ordering based on account access semantics)
- vProg executor (executes vProg zkVMs)
- Execution caching pipeline
- vProg state retention subsystem (storage, rollback, replay)
- vProg p2p state bootstrap (aka IBD)
- vProg p2p realtime sync
- vProg SDK (vProg account data access in different ZK frameworks)
- RPC interface for state access
- RPC for account transform and event subscriptions
- Client-side state syncer primitives (help clients handle reorgs)
- WASM/Python SDK adaptors (bindings to Rust SDK)
- Wallet integration
- A binary for vProg interaction (CLI)
- vProg upload (publishing) functionality
- EPA program
- EPA proof adaptors
- System program vProg (TBD)
- Program loader program vProg (TBD)
- vProg executable data validator (ELF and WASM binary validation)
- Fungible token program vProg (token standard)
- Proving subsystem (proving job scheduler integrating GPU support from different ZK frameworks).

Acknowledgments

Many thanks to Maxim, Alex, and Natasha for their continuous support and feedback on different aspects of this research.