Real-time Programmability over BlockDAGs

Litepaper v1

Igra Labs

Abstract

We present **igra**, a decentralized EVM-compatible programmability layer designed to work atop the Kaspa network. **igra** leverages Kaspa's security and throughput to the fullest extent, providing high volumes and internet-fast finality with proof-of-work security guarantees. To maintain decentralization, **igra** proposes a tandem of economic incentives with novel data-availability solutions tailored for Kaspa's unique consensus mechanism.

igra is a *based* rollup system (see [Dra23]): users interact with the programmability layer by posting transactions to the *base* layer. The based architecture sequences user transactions in the order determined in the base layer consensus, providing two incredible benefits: **igra** inherits Kaspa's exceptional security, scalability, and performance, while sharing its revenue with the base layer rather than being parasitic.

The **igra** design furnishes novel resolutions for trust bottlenecks. To enable many independent sources of truth, we introduce *accepted transaction archival nodes* (ATANs): a modular design for data storage that minimizes the subset of data that requires a trusted Kaspa node for validation. The ATAN design is highly modular and of independent interest to any network facing data availability requirements. We propose that **igra** achieves node decentralization by an ATAN-based network with economic incentives for node operators (in the form of a stakeable coin).

Finally, we propose three bridging models, one interim, and the other two based on the future roadmap of the base layer: ZK based bridging if ZK opcodes are ever implemented, and MPC based bridging in case they are not.

1 Introduction

Kaspa (Aramaic: כַּסְפָּא, silver) is the fastest and most capacious proof-of-work sequencer in the world. By design choice, the only on-chain utility provided by Kaspa is an implementation of the UTXO model with arbitrary payloads (with plans to add ZK opcodes and subnetwork support), delegating more complex functionalities to higher layers.

igra (Aramaic: אִיגָּרָא, roof) is a general-purpose programmability layer that can platform arbitrary smart contracts. **igra** employs a *based rollup* architecture [Dra23], where *sequencing* is delegated to the *base layer*.

igra is a programmability layer sequenced on Kaspa, offering users the fastest and most performant decentralized execution stack to date, which relies on proof-of-work secure sequencing.

A staple of the based architecture is that users interact with the igra network by posting *Kaspa* transactions. This design has many benefits, but the three most significant are:

- It allows igra execution and assets to inherit security properties of the *base* layer, such as censorship and MEV resistance, and strong sequencing guarantees.

- Since igra activity is posted in base layer transactions, igra activity directly contributes to base layer revenue. This is unlike non-based rollups (both optimistic and ZK-based), that have been described (especially over Ethereum) as ”parasitic” (see [HM25] for an overview).

- Conditioned on mild extensions to the base layer functionality (which are already on the Kaspa roadmap), a based architecture can support *canonical bridging*: a decentralized, trustless apparatus for transferring assets between the layers.

Kaspa’s high throughput and fast confirmations make it an ideal basis for a based rollup. However, to maintain decentralization in a high-throughput setting, Kaspa’s design is *pruned*. Kaspa full nodes are not required to store the entire ledger history. While users can opt to run *archival* nodes, they are not necessary for the network, and are very costly to maintain. A pruned design delegates maintaining data availability to the higher layer.

To provide data availability while avoiding archival nodes, igra developed a new form of archive called an *accepted transaction archival node* (ATAN). ATANs provide a clear logical separation between trusted and untrusted data, simplifying the syncing process and providing much-needed flexibility in data retention policies. For these reasons, we believe ATANs are of independent interest to any service that involves data availability[1]. We provide a self-contained account of ATANs (not specialized to the scope of the igra network needs) in Appendix A.

1.1 Base-Layer Sequencing

igra is a *based* architecture, which means that *all user activity* on igra is conducted by posting transactions to the *base layer*: to transact on igra, one posts a *Kaspa* transaction.

The based architecture[2] is inherently different than currently deployed rollups. Delegating the consensus process to the base layer has two significant benefits. First, not having to provide a sequencing consensus mechanism makes the design considerably simpler.

Second, the higher layer inherits many of the good properties of the base layer:

- **Decentralization:** Sequencing determines which transactions are processed and in what order. Non-based rollups cannot rely on the base layer and must provide their own apparatus to determine sequencing. For that, they must either provide their own decentralized consensus network or centralize sequencing. The former is very difficult to achieve, while the latter requires users to *trust* the sequencer not to abuse their authority for censorship, value extraction, or other forms of tampering. Based rollups utilize the base layer as a sequencer, inheriting its decentralization, Sybil resistance, confirmation confidence, and other properties.

- **Participation:** A common criticism towards non-based rollups is that they are *parasitic*. The higher layers accrue profits from all the financial activity therein, paying a pittance to the base layer for being its postage service (see [HM25] for a review). Based rollup activity requires users to post base layer transactions, repaying the network (at least) proportionally to the amount of activity on the higher layer.

[1] We point out that the ATAN design was made possible through collaboration between the igra devs and Kaspa core, which resulted in an improvement proposal that was included in the Crescendo hard fork [ZM25]

[2] often attributed to Justin Drake [Dra23]

- **MEV Resistance:** The centralized nature of non-based sequencers allows the sequencer to manipulate transaction inclusion and order arbitrarily, maximizing value extraction. Base-layer sequencing enjoys the MEV resistance of the base layer. MEV is a problem for base chains as well, but not to the same extent as higher layers with centralized sequencing. Moreover, any MEV countermeasures implemented on the base layer automatically apply to based rollups. Kaspa's fast block rate and multi-leader consensus arguably already provide it with more MEV resistance than slower chains, since miners can not control which transactions in their block will be processed. Future plans exist to implement a bidding system for transaction inclusion that will make MEV much harder, if not impossible. Based rollups seamlessly benefit from such solutions as they are deployed. A similar statement can be made about the related problem of implementing decentralized oracles.

1.2 Trust

In the context of cryptography, *trust* means whose word you have to take, and what are the consequences of having this trust breached.

A completely trustless system is impossible. Even in Bitcoin, there is no trustless way to establish that, say, there wasn't a year-deep reorg yesterday. However, the trust we *do* have to give is rather weak: it only takes a single observer to create a verifiable proof that a reorg *did* happen. This kind of trust assumption is often called ”one-out-of-n” trust, or *social consensus*, and the full history of *any* cryptocurrency can never be validated without it.

The textbook example of a *strong* trust requirement is when the sequencer is controlled by a small set of entities (as a consequence of curating a permissioned whitelist of node operators, withholding node software, or unreasonably high hardware requirements for sequencers). Even if some sort of voting procedure is used to determine the sequencers, the best you can hope for (and even that, under assumptions many find unreasonable) is that you have to trust that at least *one third* of the attesters do not censor transactions.

When trying to make a network *decentralized*, the goal is to reduce the required trust as much as possible. This is achieved by applying trustless cryptography as much as possible and ensuring that any other property relies, in the worst case, on BFT-like assumptions[3] (namely, $2n/3$-out-of-n consensus, where there are no critical obstructions for the size of n).

1.3 Bridging

A *Bridge* is a mechanism that allows transforming an asset on one chain to an asset on another chain. Bridging happens in two directions: *Entry* transactions *lock* base layer assets (namely, Kaspa coins), and *mint* a wrapped version on the high layer. *Exit* transaction *burn* wrapped assets on the high layer, and *release* the locked coins on the base layer.

The key difference between the two bridging directions is that the validity of entry transactions only relies on the state of the base layer, whereas the validity of exit transactions depends on the state of the higher layer (namely, on whether wrapped assets were actually burned).

For this reason, implementing entry transactions is rather straightforward, while (trustless) exit transactions are much more involved.

The key tool for exit transactions is ZK proof systems. They allow us to provide verifiable proof that the state of the higher layer indeed satisfies the required conditions.

[3] Strictly speaking, this is the *best possible* trust assumption for voting mechanisms that do not involve proof-of-work (or some other external, physical resource). You can somewhat improve it in a weaker sense of ”trust” that still allows a $> n/3$ collusion to act adversarially, but imposes an economic risk. The most common example is social slashing in proof-of-stake. It allows you to reduce the trust assumption from $(2n/3 + 1)$-out-of-n to k-out-of-n for some $k > n/3$, where k is determined by the number of attesters you are convinced would rather avoid breaching your trust than face slashing risks.

The holy grail of bridging is a *canonical bridge*. Namely, a bridge that completely relies on the base layer cryptography and security, and makes no additional trust assumptions. Unfortunately, a perfect canonical bridge is impossible without ZK[4]. Since base layer ZK is not yet available, most of the current effort is to reduce the bridging trust assumptions as much as possible. In particular, we expect any bridging solution to be:

- *Isolated*, in the sense that any trust assumption made for bridging is required *only* for bridging, and does not affect the trustlessness of any other activity.
- *Seamless*, in the sense that switching in the future to another bridging solution *does not require* any user action.
- *Contingent*, in the sense that there are guarantees that prevent base layer assets from being locked forever in case of some calamity (such as the entire Igra Labs team and collaborators turning Amish overnight).

1.4 Relation to vProgs

The igra architecture is closely related to the notion of vProgs, a *base*-layer facility under development in a collaboration between Kaspa core developers and Igra labs.

vProgs provide a hardwired framework for *verifying* state transitions in higher layers. Crucially, vProgs are *not* on-chain smart contracts: on-chain smart contracts store the state and logic of the smart contract on the chain, whereas vProgs and other rollup strategies delegate the state maintenance and contract execution (and in the case of ZK rollups, proof generation) to external means and only use the base chain to *validate* the current state. In terms of trust, vProgs *do not* provide an advantage over any other base architecture that validates states and state transitions on-chain.

At the time of writing, the most established benefit of vProgs is that they provide better *granularity*. vProgs are structured to allow a much more efficient parallelization of ZK proof computation and verification by providing a powerful ”segmenting and restitching” on-chain paradigm. They are also helpful for easily deploying tiny functionalities that do not merit their own network. (Current research leads suggest that vProgs might also be beneficial in terms of interoperability, but the discussion around this aspect and its feasibility is far from concluded.)

The current design of igra provides the same functionality and requires the same trust (for processing, not bridging) as vProgs. More abstractly, we can think of igra (and any based L2, for that matter) as a ”vProg of vProgs ”: a main vProg that deploys the igra network itself (committing to its initial state and state transition rules), which implements a container for vProgs that represent contracts deployed *on* the igra network.

One of the long-term plans of igra is to continue the collaboration until vProgs are mature enough to pivot to an explicit ”vProg of vProgs ” design. If successful, this will not only provide unparalleled efficiency, but also allow interesting features, such as *detaching* a contract from igra and converting it into an on-chain vProg and vice versa.

1.5 Paper Structure

In this document, we present the igra architecture and design, followed by a discussion of the trust model and economic model.

Remark. At the time of writing this document, Kaspa's latest update is the Crescendo hard fork [Sut25a]. Crescendo introduces some changes crucial for a based programmability layer, but still does not introduce ZK verification opcodes. Moreover, the design of the base layer verification

[4] In fact, it is arguably impossible even *with* ZK, as ZK tends to introduce new (mild) trust assumptions (at the very least, trusting provers to actually produce proofs). However, ZK does enable reducing trust to the lowest weakest assumptions possible.

system (including which proving systems are to be supported) is still under discussion. (This is colloquially described as the ”post-Crescendo pre-ZK era”.)

Consequently, we describe an implementation of the igra network in the current version of Kaspa, accompanied by the planned changes once ZK (and other) capabilities are introduced. We tried to make the post-ZK suggestions as concrete as possible, given that the final architecture cannot be fully determined at this time, as it will be affected by the final specification of the Kaspa ZK hard fork.

In Section 3 we describe the various components of the igra node, and the protocols that govern their data flow. We proceed to discuss each component separately.

In Section 4 we discuss Kaspa-to-igra bridging, and propose three bridging approaches:

- *Community bridging* is a *trusted* but *interim* solution, designed to get things rolling while the base layer design converges. While this type of bridging is ultimately trusted, we made great efforts to provide many contingencies for users. Community bridging is discussed in Section 4.1

- ZK *bridging* uses zero-knowledge to notify the base layer that value was transferred from the higher layer. This bridging requires ZK opcodes (and some other base layer nuts and bolts), so its design is currently tentative. ZK bridging is discussed in Section 4.2.

- MPC *bridging* is an *attester based* bridging design, proposed for the scenario where ZK opcodes are *never* introduced to the base layer. While this design will probably not be used for Kaspa-to-igra bridging, we chose to present it in detail as we plan to use a similar design for other purposes, such as bridging Bitcoin to igra, or DAO governance (see Section 5). MPC bridging is discussed in Section 4.3.

In Section 5 we discuss the $Igra coin purpose and utility.

Finally, we conclude in Section 6 with a discussion of the trust model induced by our design.

Appendix A contains a detailed discussion of ATANs in the abstract, not specialized to igra's needs. In Appendix B there is a more detailed account of the block creation flow. A future version will include a similar appendix that outlines the reorg handling logic.

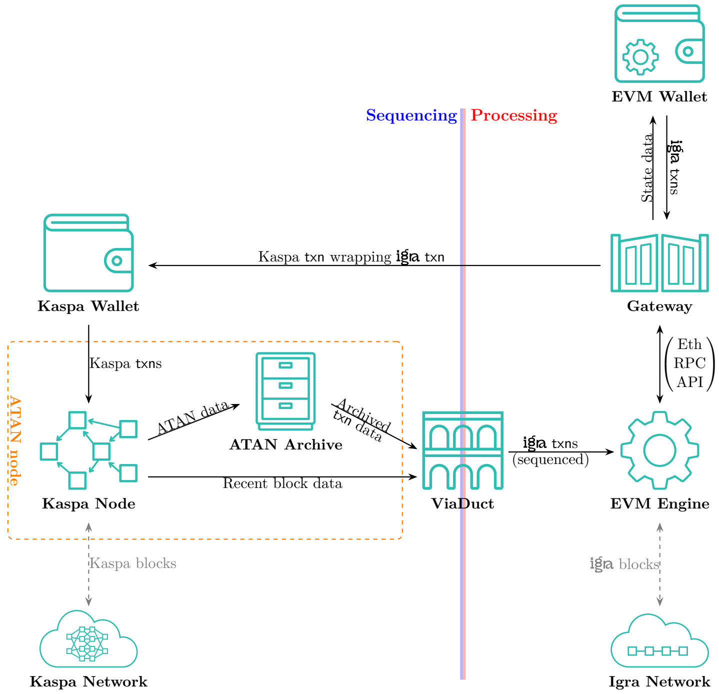

Figure 1: igra node acrhihtecture

The various components of an igra node and data flow among them. The ViaDuct component acts as the bridge from sequencing to processing. It reads off the sequencing from the base layer, unwraps and validates the igra transactions, and passes them to the EVM engine. The based architecture removes the need for such bridging in the other direction: since wrapped transactions can be directly posted to the base layer, we only need the bridge to wrap them without any further processing.
(A full node does not include an EVM wallet or a Kaspa node, but the communication with both is an essential part of the data flow, so they are included in the diagram for completeness.)

2 Background

(Required background about Kaspa, smart contracts, rollups, and so on).

To be completed in a later version.

3 Architecture

The igra node architecture is loosely divided into four main components[5]: the igra Gate, EVM Engine, ViaDuct, and ATANs. In this section, we want to introduce these components and their

[5] Of course, other components (such as a Kaspa node) are required for the network to be functional. These could be packaged with the node, or be used as external peers.

mutual and external interfaces. The high view picture of the node architecture is summarized in Fig. 1.

The component-wise division follows the same principles as most other permissionless higher layers. But with a slight twist: In most networks, there is a single component that's in charge of connecting the consensus and processing layers. igra's based architecture imposes a strong *asymmetry* (that's also very apparent on Fig. 1). Going from the processing layer to the sequencing layer is almost seamless, as it only requires wrapping the transaction appropriately and delegating sequencing to the base chain. Reading and processing the result of the sequencing into a form digestible by the EVM engine is a much more involved task, for which a designated component called the *ViaDuct* is responsible (see Section 3.3). Moreover, the ViaDuct requires help with data retention, and that job is further delegated to a component called an ATAN (see Section 3.2).

ATAN nodes provide a functional decoupling of chain integrity from data availability, which we believe is of independent interest for Kaspa builders, so we provide a more general overview in Appendix A (see also the blog post [Wyb25]).

3.1 Gateway

The igra gateway's job is to provide an API to the igra layer that is compliant with familiar industry standards. The goal is to allow existing user interfaces, such as MetaMask and other EVM wallets, to communicate with igra agnostically.

In particular, the Gateway has to deal with two types of communication: igra transactions, and EVM state queries.

The former are handled by wrapping the transaction into a Kaspa transaction, where the wrapping must comply with the format expected by the ViaDuct: txid_mask (see Section 3.2.1).

The latter are handled by directly relaying the query to the EVM engine and relaying back the response.

3.2 Accepted Transactions Archival Nodes (ATANs)

The standard full node Kaspa is *pruned*: it only stores a small suffix of the data (on the order of several days). This is sufficient for transactional layers, but other applications (smart contracts included) require more data retention. Kaspa nodes can run in an *archival* mode, where *all* data is stored. Archives solve the data availability problem, but at a very steep price in terms of storage requirements.

Pre-Crescendo, attempts to construct higher layers (with limited functionality) over Kaspa ”solved” data availability issues by expecting the layer two indexers to run archival nodes, rendering the networks extremely centralized. This is not only due to the exorbitant storage requirements, but also since syncing a new node trustlessly requires obtaining *and fully processing* a rapidly increasing run stack.

The storage requirements could potentially be dramatically reduced once ZK verification is possible on the base layer. But at the time of writing, the design of ZK opcodes over Kaspa is still a matter of discussion (and after this discussion is concluded, the discussion on how to best use the ZK capabilities to remove data availability requirements can *begin*).

The key observation is that a small change to how transactions are committed to blocks in Kaspa can enable this optimization. This small change was proposed by igra as KIP-15 [ZM25] and implemented in the Crescendo hard fork. (See Appendix A.1 for a detailed discussion).

This enables a new design – which we call an *Accepted Transactions Archival Node* (*ATAN*) – that *decouples verifying the chain consistency from processing transactions.*

ATANs are designed to provide an interim solution by minimizing storage requirements as much as possible. The result is a node configuration whose storage increases linearly, but at a much more manageable rate than a full-blown archival node.

More abstractly, ATANs decouple the validation of the authenticity of the current igra state from providing services that require data availability (such as validating transactions with old dependencies). This could lead to future designs that use ATANs for data availability, while using the sync logic in lighter modes of node operation.

Unlike an archival node, an ATAN only needs to store the *headers* of the *selected-chain* blocks, and the transaction IDs (which are hashes), and only the bodies of *relevant* transactions. (In fact, the power of hindsight tells us that with another minor modification, we could even get rid of the headers[6]. Hopefully this modification will be introduced in the next hard-fork.)

This change has two important consequences:

- The amount of overall stored data reduces by orders of magnitude.
- There is a clear logical separation between the (small) *sync data*, and the much larger *state data*. The sync data can be obtained from any other ATAN and verified against any trusted Kaspa nodes. This data suffices to trustlessly verify the state data, allowing nodes to gather it from multiple sources in any arbitrary order (for example, by demand). For a more extended discussion of this aspect, see the blog post [Wyb25].

The above description is actually just one possible configuration for an ATAN, designed for the igra network requirements. ATANs are more flexible and can be tailored to many applications with different data availability requirements. We provide a more detailed discussion in Appendix A.

3.2.1 The txid_mask Optimization

There is a particularly nice "trick" igra uses in its implementation to easily facilitate ATANs that (almost) only store relevant transactions.

The trick is to require the first ℓ bits of the transaction ID to match a given string.

Recall that a transaction ID is a hash, so each of its bits distributes pseudo-uniformly, making all possible initial strings equally likely.

We can use that by choosing an arbitrary *bitmask* txid_mask of some length ℓ, and proclaiming that a valid transaction txn must satisfy txid_mask = txn.ID $[0 : \ell]$. This requires having a field in the transaction structure (most naturally, its payload) for a nonce. When creating the transaction (in our case, when the igra gateway wraps the L2 transaction), nonces must be churned until finding one that produces an ID starting with txid_mask.

By setting this rule, you can set an ATAN tasked with storing all valid transactions to filter out transactions with an invalid prefix. The ATAN will only store a fraction of about $2^{-\ell}$ of irrelevant transactions: those that came up with the correct prefix by sheer chance.

This filtering process can be tuned for arbitrary accuracy, while providing an extremely simple and efficient filtering protocl that does not even require inspecting the transaction content, just computing its ID.

This benefit comes at almost no cost or inconvenience for the user. Having to compute tens of thousands of hashes to produce a single transaction might seem excessive, but it is actually inconsequential. IDs are usually computed with highly efficient hashes, typically not the same hashes uses for mining. In Kaspa, block and transaction IDs are computed with Blake3, known for its efficiency, especially on generic CPUs. With standard hardware, computing a *single* ECDSA signature is roughly as heavy as computing *one hundred thousand* blake3 hashes.

[6] To be more specific, igra nodes need to store the DAA scores of headers in an authenticable way. Storing the entire header chain provides a way to authenticate them, but we could avoid storing the headers by heading the DAA scores to the AIDMR (see Appendix A.2). Recall that Kaspa headers are rather chunky due to how pruning works (see Section 2), so this is significant.

3.3 The ViaDuct

The ViaDuct is responsible for scanning the base layer for two things: relevant transactions and reorgs, and communicating this information to IgReth (the Igra execution engine, see Section 3.4) accordingly. It is instructive to think of the ViaDuct as ”emulating” the environment that EVM engines were designed for, pretending to retransmit transactions via gossiping, whereas they were actually extracted from the base layer.

The ViaDuct scans the network for appropriately wrapped igra transactions, sifts them by performing ”shallow” checks that do not require access to the state of the igra network (validating the txid_mask (recall Section 3.2.1), payload form, and so on), and reports survivors to IgReth via its RPC interface.

Besides that, the ViaDuct is responsible for delimiting the set of blocks that fit into a single igra block. It is currently parameterized so that igra block represents approximately ten Kaspa blocks, or about one second. When the ViaDuct decides to start a new window, it provides IgReth with header data and instructs it to construct a new block. A more detailed account of the block construction procedure is provided in Appendix B.

Finally, the ViaDuct is responsible for handling reorgs. Whenever a new window is about to start, the ViaDuct checks whether all base layer blocks that were included in the recently closed window are still on the selected chain. If they do not, it initiates a reorg procedure on the igra network. At the time of writing, the reorg procedure is still being optimized, a detailed account of reorg handling will be added in a future version.

A key subtlety is that the ViaDuct does not consider the most recent selected tip. Instead, there is a slight delay. The reason is that more recent blocks have a higher chance of being reorged (recall that shallow reorgs constantly occur on Kaspa even during healthy operation). Looking too close to the tip induces a large reorg overhead, while looking too far induces confirmation delay. Hence, it is imperative that the reorg handling will be as efficient as possible. At the time of writing, the delay is in the vicinity of 3-4 seconds, but it is far from being fully optimized. Future versions of this paper will include a detailed account of the optimized reorg handling procedure.

3.4 IgReth Execution Engine

The igra execution engine is based on *Reth*, the highly performant Rust implementation of Ethereum’s execution engine.

Reth follows the Ethereum API engine, which standardizes the communication between the consensus and execution layer. In our case, it is the layer connecting the execution engine to the ViaDuct.

Since Reth is both complicated and sensitive, a key design decision was to cater to Reth’s original design as much as possible, and work around its expected inputs. In particular, as we have seen in Section 3.3, the ViaDuct was designed to emulate the natural environment of Reth as closely as possible.

However, some changes were inevitable. The most major difference is that IgReth, unlike Reth, *clears the mempool* after each igra block (or about ten Kaspa blocks). This means that a transaction that appears in the window but not in the resulting block must be resubmitted.

To progress the state, IgReth and the ViaDuct work together to construct the next igra block. The exact block construction flow is described in Appendix B.

The final responsibility of IgReth is to determine the gas price. Currently, IgReth follows Reth almost to the letter by implementing the standard EIP-1559 [BCD+21], except also enforcing a small lower bound to prevent denial-of-service attacks[7].

[7] If gas is basically free, cheap transactions could drain the gas limit of the entire block.

4 Bridging

In our context, *bridging* is any infrastructure that enables the following two operations:

- **Entry**: *Lock* base layer coin ($Kas) to *mint* an equivalent amount of wrapped coin ($iKas) into an address of your choice.

- **Exit**: *Burn* wrapped coin to *release* an equivalent amount of base layer coin.

Note that base layer coins are locked and released, whereas wrapped coins are minted and destroyed on demand. That's natural since, on one hand, the base layer should not adapt its emission to higher layers, but on the other, wrapped coins that are not backed by locked base coins should not exist.

The ”easy” way to provide bridging is with a centralized service (similar to how Chainge provided wrapped Kaspa), that is responsible for issuing and burning wrapped coins, and is *trusted* to have enough solvency to allow anyone to exit on demand.

Providing *trustless* bridging is much more challenging, as it requires coordinating two networks without delegating excessive power to any single entity. The key issue is *who controls the locked coins*. A central entity that manages locking and releasing induces the strongest trusted assumption on the most sensitive guarantee: that user funds are not stolen. On the other hand, if we let users retain control of the locked coin, then they must be *trusted* not to spend it before burning the corresponding wrapped coin.

In this section, we consider three bridging paradigms for igra, and the trade-offs they induce:

- Community bridging: a simple *trusted* setup where the trust is delegated to community elected representatives. This is a *temporary* solution that's only meant to bootstrap onboarding so that the network can start operating while a more long-term solution is decided.

- ZK bridging: contingent on appropriate ZK support on the base layer, this approach allows delegating the bridging to the base layer, inheriting its trust assumptions.

- MPC bridging: delegating the bridging to a *permissionless validator network*, using proof-of-stake based BFT to approve fund release. The permissionless validator network is on the roadmap for another purpose: bridging Bitcoin to igra.

4.1 Community Bridging

Community bridging is a *temporary* and *trusted* solution, meant to bootstrap the network in its early days.

At its core, community bridging is a specialization of *multisig* bridging. Recall that an n-out-of-k multisig wallet is a UTXO with n owners, such that *any* subset of k out of the n owners can spend the UTXO.

Multisig bridging follows this template:

- **Setup**: n and k are chosen, as well as the n signatories. In a public ceremony, the signatories create the wallet. All observers can use the resulting signature key to compute a *locking script* lock. We discuss the locking script in more detail in Section 4.1.1.

- **Entry**:
  - **Lock**: The user submits a transaction with a P2SH output whose address is H(lock).
  - **Mint**: Nodes mint wrapped coins accordingly.

- **Exit**:

  - **Burn**: The user burns wrapped coins on chain.
  - **Unlock**: The signatories pay the user on the base layer from the locked coins.

Note that this template does not assume that lock is a simple multisig wallet. In fact, igra uses a more complicated script to provide more contingencies to users.

Also note that the trust bottleneck is in the unlocking phase. Indeed, the other bridging paradigms follow similar templates, where the key difference is in the unlocking stage. The fact that unlocking is the trust bottleneck is a direct consequence of the setting. As layer two designers, we can set protocol rules that track minting and burning and govern locking. However, unlocking requires facilities for making the base layer correspond to the higher layer.

To reduce conflicts of interest, Igra Labs proposes that the community will vote in the signatories. This will create a healthy separation between providing igra's technology and curating its liquidity.

4.1.1 The Locking Script

The locking script relies on three tenets:

- A small set of *bridge operators*,
- A large set of *waivers*, and
- A *time lock*.

The time lock represents a date by which the community believes a better bridge will be available. Bridge operators and waverers are *both* community-elected signatories. The roles are as the names suggest: bridge operators are the only ones allowed to spend the UTXO, but they can only do so if the time lock has either expired or been waived.

Practically, if sufficiently many operators and sufficiently many waivers agree (or if the operators agree and the time lock expired), they can spend locked funds however they want for whatever reason they want[8]. However, we *trust* them to do the following:

- Pay $Kas to users who burned $iKas.
- Transfer the locked funds to a new bridge (represented by a new locking script) once one is available.
- Transfer the locked funds to a new locking script with an updated lock, if the lock expired before a better bridging method becomes available.
- Release money back to the users who locked it in the doomsday scenario where the igra network ever goes out of commission.

4.2 ZK bridging

Recall that the central trust bottleneck for bridging is unlocking funds on the base layer. And the key issue is that there is no way to report to the base layer that the state of the higher layer is consistent with releasing these funds.

Zero knowledge (ZK) provides a way to condition the release on the state of the higher layer. Instead of the base layer inspecting the higher layer, it validates a *proof* that the state of the higher layer is as claimed.

[8] Bitcoin and Kaspa do support posing conditions on the *receiver* of the transaction. Once the locking script is satisfied, all valid outputs are allowed. Placing conditions on who could receive the coin is sometimes called *covenants*.

This ability makes it possible (though far from trivial) to coordinate higher-level burning with base-layer unlocks.

At the time of writing, the ZK utilities on Kaspa are still under consideration, in an ongoing discussion between Kaspa core and Igra Labs developers. The latest snapshot of the discussion is available in [Sut25b].

4.3 MPC bridging

MPC stands for *multi-party computation*, a reformative paradigm in cryptography whose base question is: how can many participants who do not trust each other compute a function together, such that none of them learn about each other's input[9].

Our architecture connects MPC to consensus via a framework called FROST[10] [KG20]. The FROST framework provides us with a way to create something that is, in a way, an *adaptive* multisig.

It is similar to multisigs in the sense that there is a UTXO on the base layer that can only be spent if enough people cooperate. It is different in the sense that the list of eligible voters (and the weights of their vote), can be updated *without* changing the locking script. As long as no funds are unlocked, the UTXO remains as is, no matter how many times the validator set was changed. The only situation where the UTXO is spent is for the purpose of unlocking money, and in this case the change is spent to a UTXO with the *same* locking script.

In a bit more detail, the key cryptographic primitive here is a *threshold signature*. To initialize a threshold signature, three pieces of data are required: a list of signatories, a *weight* for each signature, and a *threshold*. A signed message is considered valid if and only if *the accumulated weight* of *distinct* signatories who signed it exceeds the threshold. A key feature of threshold signatures is that they are very succinct. The signature size does not grow with the global number of signatories, or with the number of signatures on the particular message. Yet, the scheme guarantees that a valid signature has enough *distinct* weights to pass the threshold (in particular, it is smart enough not to double-count signatures, even if the same signatory signed many times).

The FROST framework provides the following:

- A *trustless* framework to generate a threshold signature. (Here ”trustless” means that the setup does not generate any ”toxic waste” whose holder can override its security).
- An MPC protocol to generate threshold signatures.
- An MPC protocol to update the list of signatories.

Over this framework, proof-of-stake protocol can be mounted whose rounds look roughly like this:

- Anyone who wishes to vote in this round stakes coin.
- They run the FROST MPC protocol to provide each signatory with a signature key whose weight is proportional to the size of its stake, where the weights are normalized such that they sum to 1. The threshold is set at $2/3$.
- Each validator proposes a transaction, a validator that thinks no money should be unlocked this round signs on $\perp$.

[9] The origin of this distinguished line of research is often attributed in the lore to the classical *Yao's Millionaire problem* (Andrew Yao, 1980): how can n mutually distrusting millionaires determine who is the richest such that no one learns anything about how rich the other millionaires are. (This question was asked back in the days when being a millionaire was enough to be rich.)

[10] This is the same framework used by Zcash for threshold signatures for shielded transactions [Zca25].

- The validators run a threshold-signature-based proof-of-stake protocol, and run it to obtain a valid signature.

- If the signed message is not $\perp$ but a valid transaction, it is posted to the blockchain.

In our case, the staked coin will be the Igra coin. The usual minting rewards and slashing mechanisms are used to incentivize good behavior.

5 $Igra Coin Utility

igra will launch with two tokens: $iKas and $Igra.

The $iKas coin has a very strict utility as the medium of bridging between the base and higher layer. This utility dictates a permanent correspondence between existing $iKas and locked $Kas. In particular, $iKas cannot be minted for any purpose other than backing locked $Kas, so it is only suitable for utilities monetized as $Kas (such as gas).

The $Igra token utilizes and facilitates all other economic needs of the igra network. It provides network participants with the means to *attest* for various decisions, and fund utilities that are in the interest of the igra network.

The most straightforward utility of $Igra is to incentivize users to run nodes and become attesters. Attesters are responsible for ensuring the consistency of the igra state with the igra transactions on the base layer.

Once ZK is introduced, another equally fundamental responsibility emerges: providing ZK proofs. Computing ZK proofs in a timely manner is an expensive feat that often requires custom hardware. While one may argue that network users will run nodes out of altruism, and the general betterment of the coin (as they indeed do in many coins, including Bitcoin and Kaspa), it is less reasonable to assume the same for the much more technically demanding and expensive task of producing proofs[11]. This is crucial to decentralization, as centralized proof generation can be easily used to censor and even coerce the network. The $Igra coin could compensate provers, be it from the community or from an external prover network, for their proofs.

Finally, $Igra is used for *attestation*. $Igra holders can *stake* coin to attest for the current state of the network. Like it usually is in proof-of-stake, a super majority of attesters must be honest to ensure correct attestations. However, in the situation where the honest attesters are a *plurality* of at least one-third, slashing will apply to punish the rogue attesters. Staking allows and encourages participation, reducing the risk that a supermajority (or even a plurality) of validators are controlled by the same entity (Igra Labs or otherwise). This is the exact same trust and economic assumptions that exist in any proof-of-stake coin.

Finally, $Igra holders receive privileged access and reduced fees for network services. This creates incentives for holding $Igra that are not reserved for attesters, or even node operators, but apply to all users and holders.

The igra network will launch with a DAO for $Igra holders to vote on various protocol parameters, such as window sizes, the txid_mask length, staking rewards, and bridge operator thresholds.

6 Trust

To truly understand how potentially decentralized any network can be, one must understand its trust bottlenecks and assess their impact.

[11] Some may argue that the ZK proof could be paid for externally by the user. But even if we allow this problematic constellation, it still leaves open questions such as who will produce the (much more involved) proofs of state snapshots, or any other feature that requires ZK proofs.

To analyze igra, we divide its flow into *three* parts: Synchronizing a new node, processing incoming traffic, and bridging between igra and the base layer.

This division allows us to inspect the trust bottleneck of each service. Since these services are all essential, the *strongest* trust assumption required among the three is ”the” trust assumption required for the continued operation of igra.

However, this division serves a purpose beyond simplifying the analysis. It is beneficial to know that, for example, processing and syncing require less trust than bridging. First, it tells us that to reduce the trust required by the entire network, it suffices to reduce the trust required for bridging. Furthermore, it is encouraging news for builders of a project whose value proposition relies on layer two processing for more than bridging[12].

6.1 Trust Assumptions

Trust is about relying on someone to act in a certain way. That’s hard to formalize, because we need to capture both the expected behavior and the repercussions of breaching our trust.

The strongest form of trust is known as a *full trust*: an entity (or small number of entities) that *must* act in a particular way, or there are no guarantees whatsoever.

A weaker form of trust is k*-out-of-*n trust, where out of a group of n entities, it suffices that k behave honestly for the protocol to work *regardless* of how the remaining entities behave, even if they are colluding.

We can qualify this further, into *permissioned* and *permissionless* k-out-of-n, where in the former the n entities are chosen in advance[13], while in the latter this list can be updated without requiring any additional trust.

For example, we can say that a typical proof-of-stake system consensus protocol provides a permissionless $2n/3$-out-of-n consensus (though other components of the network, outside consensus, might have weaker/stronger trust assumptions).

More interestingly, obtaining a ”source-of-truth” for syncing a Kaspa/Bitcoin node only requires permissionless 1-out-of-n trust: the protocol rules provide a cryptographic litmus to determine which of the responses is ”true”, so to sync properly it suffices to have *one* truthful source[14].

However, while it seems that Kaspa and Bitcoin require the same trust assumptions, there is another crucial subtlety: how does n evolve with time? If becoming a source of truth quickly becomes more expensive as time goes by, the 1-out-of-n required trust is actually undermined by a decreasing (or not sufficiently increasing) n.

This is a discussed concern for Bitcoin: despite the very slow data accumulation, the ability to have a *sync-then-pruned* node that only verifies the chain once (and even that in pieces, never holding more than a fraction of the chain at once) and keeps maintaining only the last few days from there. Such a node cannot be a source of truth, but there is great incentive (in terms of hardware, storage, and energy costs) to prefer it over a full node. This dynamic reduces the number of sources of truth, strengthening the trust assumption.

This dynamic is even more pronounced in a high-throughput network processing thousands of complex contracts daily. Naively requiring each source of truth to store the entire ledger, to be fully downloaded and verified by each new synced node, strongly limits the growth of n. So while it is still permissionless 1-out-of-n, if n never exceeds 3, and increasing it is allowed but

[12] Consider, for example, a token used for online auctions. The fundamental guarantee is that the bids cannot be tampered with, which is unrelated to bridging. Of course, there should also be a sufficient guarantee that bidders can enter and exit their funds. However, the latter is not as fundamental: even if it turns out that the bridge was compromised at some point, it does not undermine the validity of the bids that took place during that time.

[13] By who? That might be a bottleneck too!

[14] Note that this is just the trust model for *syncing*, you still need an $n/2$-out-of-n honest behavior on the consensus layer

too expensive for most people, we are still at a very strong bottleneck. Hence, when discussing permissionless k-out-of-n trust, we should not only consider how k grows with n, but also how n grows with time.

Finally, when discussing a trust bottleneck, we must also examine the repercussions of breaching our trust assumptions. For example, in bridging, a trust breach might mean stealing user funds or unlocking coins whose wrapped counterparts weren't burned. Whereas in proof-of-stake, violating $2n/3$-out-of-k trust but not the $n/3$-out-of-k trust typically implies possible liveness issues (where economic mechanisms like slashing greatly increase the costs of such attacks) but no full-blown cheap censorship attack. Arguably, the latter makes a stronger assumption, but for less dire consequences.

6.2 Base Layer Trust Delegation

Throughout this document, we have assumed that we have access to a *trusted* Kaspa node. This should *not* be interpreted as a single entity that we have to trust, but rather as *delegation*: we assume the ability to find a node that we can trust temporarily (or sync one ourselves), but that we can detect and recover from any deviations.

The *trusted* here implies that we accept the *trust model* of the underlying network. To truly resolve the trust bottlenecks of igra (or any layer two, for that matter), one should also include the trust bottlenecks of the base layer. And to understand its trust model we need to understand the trust model of all underlying cryptography, and there we need to understand the trust model of the underlying hardware and so on and so forth. Assuming trust allows us to layer this process, effectively saying that our trust model is everything we specified, plus all assumptions required to trust external mechanisms (such as a base layer, a ZK proof system, etc.).

6.3 Synchronization

As discussed in Section 3.2, a key hurdle for designing a layer two over Kaspa is its pruning mechanism (recall Section 2).

In non-pruned networks, assuming access to a trusted base layer node implies unrestricted access to its entire history. Since we are working over Kaspa, this is no longer the case.

The straightforward solution is to make layer two nodes double as base layer archives, at least from the time the higher layer was launched. Archival nodes store a very significant amount of potentially unnecessary data, including headers, rejected transactions, and other related information. As stated above, experience shows that in this constellation, most operators will opt for a light node that uses a trusted recent snapshot of the state and works from there, without *ever* validating the state.

This puts us in the *terrible* position where there are very few sources of truth, and validating them becomes very impractical[15]. Hence, we cannot call this a k-out-of-n trust model, since we cannot assume that the verification process is accessible.

On the base layer, the consequence of a trust breach in syncing – Recall that a ”breach of trust” here is that not a single truthful node is available to the syncing node – is that the syncing node (and all nodes that refer to it) could be manipulated into believing a forged history. This is slightly sweetened in the higher-layer case, as the inconsistency with the base layer will be revealed during validation. However, such a breach of trust still prevents new nodes from syncing, potentially allowing existing nodes to censor transactions.

[15] Some argue that this is the current state in Ethereum, and one would be extremely pressed to find the hardware required to validate its state since genesis. Ethereum acknowledge that it has been at least five years since their chain was validated from scratch, and are transitioning into an *history expiry* paradigm [Gar25].

6.3.1 ATAN Based Nodes

In a pruned network, where the data availability falls to the higher layer, the weakest possible trust model is 1-out-of-n. Any improvement upon this model requires further assumptions on the base layer (or delegating trust to yet another network). Information-theoretic considerations show that without any form of succinct proof of validity, we must store a linear amount of data and perform a linear amount of processing to sync. To use a succinct form of validity, we need the base layer to validate it. This usually comes in the form of ZK proofs.

In the pre-ZK era, the only measure of decentralization is the growth of n. Namely, the effort required to create and maintain a new source of truth, and the strength of incentives to do so.

The ”worst possible scenario” starting point is when sources of truth must be fully archival nodes, the syncing process requires the entire ledger data, and there are no benefits for syncing new sources of truth.

ATANs (recall Section 3.2) address the first two concerns. Recall that ATANs decouple trust and data validation. ATANs only require trust to share a small subset of the data called *ATAN data*. Once obtained, additional data from *any source* can be *trustlessly* verified. That means that the assumptions we need for *correctness* of the data are much weaker than assumptions for *availability* of the data. Which is the best we can hope for, given that requiring the data is not going away as long as there are no ZK opcodes on the base layer. This has two consequences:

- Becoming a source of truth is much easier. All you need is up-to-date ATAN data, which is much smaller (and much easier to validate) than the entire ledger data.

- Providing data could be delegated to other *untrusted* sources, such as simple data-storage services, or deferring data retention to the user[16].

The $Igra token addresses the third concern by providing a tangible economic reward for operating a node (see Section 5).

6.3.2 ZK Based Trustless Snapshots

Like other parts that have to do with the explicit ZK based design, a full discussion must be delegated for when it is available (or at least fully specified) on the base layer.

The design will follow the foundations laid in the discussions between the Igra and Kaspa devs, a snapshot thereof available in the Kaspa Research forum (in particular, see [Sut24]).

6.4 Processing

Processing is modeled by assuming a user has access to a sufficiently recent *trusted* snapshot of the igra state (where *sufficiently recent* means more recent than the latest pruning point, obtaining such a state is the synchronization process we discussed in the previous section). We then ask what kind of trust is required for the user to validate the correctness of a more recent state.

In the pre-ZK era, the validating user has three options, with different trade-offs:

- Trust a particular igra node to be truthful about the current state. This is a simple solution, but relies on the strongest (1-out-of-1) trust assumption. While it is suitable in some contexts (where simplicity is of the essence, and not detecting a lie wouldn't have dire consequences), it cannot be the only option.

- Sync a node. This reduces the trust assumptions to the same ones required for syncing a node, which is the best possible. However, this is an expensive operation.

[16] Current igra ATANs are programmed to exchange the entire data, but that's a matter of comfort, not of technical necessity, and the ATAN model allows great flexibility in design.

- Attestations. Recall that $Igra is staked for attesting for the current igra state (see Section 5). Trusting these attestations requires the same trust as any proof-of-stake. This is not as secure as syncing a node yourself, but provides a *huge* improvement over blindly trusting a node, while not increasing computational costs noticeably.

Post-ZK, processing validation will become much simpler, as it only requires validating the transition proofs on the base layer (see [Sut24]). As usual, we defer a sharper discussion for when explicit specifications are possible.

6.5 Bridging

Each of the three types of bridging outlined in Section 4 requires different trust assumptions.

Community bridging imposes the strongest trust assumption: k-out-of-n for predetermined signatories. The consequences of a breach of trust depend on whether the trust was breached by operators and waivers, whether the lock already expired, and so forth (recall Section 4.1.1). Mapping all the cases is superfluous, so we just state that in the worst case, a breach of trust allows *stealing all locked assets*.

This is far from ideal. Hence, we only propose community bridging as a temporary phase while the upcoming facilities on Kaspa are finalized.

In MPC-bridging, we have trust assumptions identical to proof-of-stake. What is *different* from proof-of-stake is the consequences of a trust breach. Here, in the worst case, a supermajority collusion can steal the locked assets.

However, this consequence is not written in stone and can be thwarted with appropriate base layer utilities. Kaspa has already introduced a crude form of covenants in KIP-10 [BN24], and more covenant-like opcodes (namely, opcodes that can impose conditions on the receiver of a spent UTXO) can provide protections against MPC validators stealing assets. In the case where Kaspa does not implement ZK opcodes, appropriate covenant opcodes (which are much easier to implement) could be introduced to enable MPC-bridging with better trust. That being said, with the current base layer facilities, MPC-bridging provides the best balance of assumptions to repercussions. (For comparison, if the bridging is centralized, then the trust assumption becomes the much stronger 1-out-of-1, while the repercussions remain the same.)

Finally, the exact trust assumptions of ZK-bridging depend on the final design of ZK-opcodes. However, ignoring the trust assumptions introduced by ZK (mainly, the need to trust some entities to create proofs on demand), ZK-bridging is canonical, and thus requires no additional trust assumptions.

References

[BCD+21] V. Buterin, E. Conner, R. Dudley, M. Slipper, I. Norden, and A. Bakhta. EIP-1559: Fee Market Change for ETH 1.0 Chain. https://eips.ethereum.org/EIPS/eip-1559, 2021. Ethereum Improvement Proposal.

[BN24] M. Biryukov and O. Newman. KIP-0010: New Transaction Opcodes for Enhanced Script Functionality. Kaspa Improvement Proposal, 2024. Accessed: 2025-09-28.

[Dra23] J. Drake. Based rollups—superpowers from L1 sequencing. https://ethresear.ch/t/based-rollups-superpowers-from-l1-sequencing/15016, 2023. Posted on Ethereum Research, accessed 2025-06-17.

[Gar25] M. Garnett. Partial History Expiry Announcement, Jul 2025. Ethereum Foundation Blog.

[HM25] L. Heimbach and J. Milionis. The Early Days of the Ethereum Blob Fee Market and Lessons Learnt, 2025, arXiv: 2502.12966.

[KG20] C. Komlo and I. Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures, 2020, IACR: 2020/852.

[Sut24] M. Sutton. On the Design of Based ZK Rollups over Kaspa's UTXO-Based DAG Consensus, Dec 2024. Kaspa Research.

[Sut25a] M. Sutton. The Crescendo Hardfork. https://github.com/kaspanet/kips/blob/master/kip-0014.md, 2025. Kaspa Improvement Proposal (KIP) 14. Accessed 17-06-2025.

[Sut25b] M. Sutton. L1<>L2 canonical bridge (entry/exit mechanism). Kaspa Research, Discourse post, Jan 2025. Kaspa Research discussion post.

[Wyb25] S. Wyborski. How to Decentralize L2s on Kaspa in the Post-Crescendo Pre-ZK Era (KIP-15 for dummies). https://functor.network/user/912/entry/985, 2025. Accessed 2025-06-17.

[Zca25] Zcash Foundation. The State of FROST for Zcash, May 2025. Zcash Foundation Blog.

[ZM25] M. Zak and R. Melnikov. Canonical Transaction Ordering and SelectedParent Accepted Transactions Commitment. https://github.com/kaspanet/kips/blob/master/kip-0015.md, 2025. Kaspa Improvement Proposal (KIP) 15. Accessed 17-06-2025.

A Approved Transaction Nodes – Extended Disccussion

The ATAN archive is essential for providing a decentralized execution layer before ZK opcodes are possible. However, it has other far-reaching applications and should be considered as software of independent interest. For this reason, we expend a bit on possible modes of operations that could be useful for various applications, stressing that Igra's implementation is tailored for igra's needs.

Recall Kaspa full nodes are pruned (see Section 2), which creates data availability issues. These issues are not manifest in a transactional layer (as the UTXO set contains enough data to process incoming transactions[17]) but become noticeable in smart-contracts, as the result of the execution might rely on arbitrarily old transaction data (e.g., invoking a function from a contract that was deployed a year ago).

The direct solution is to use an *archival* node, that stores the entire ledger data. However, archival nodes have prohibitive limitations: First, requiring all validators to maintain an archival node makes validating the network extremely costly, hence centralized. Second, syncing a new node requires access to all historical data (at least as early as the network to be validated was launched), creating a huge scarcity in sources of truth. In particular, if the few original nodes refuse to provide the full sync data (e.g. in designs where "standard nodes" only require a current suffix of the execution stack), they become a trust bottleneck, as the veracity of new nodes relies on the veracity of the original nodes. The motivation behind the ATAN design was to remove this trust bottleneck.

The ATAN design provides a workable trade-off. It still requires increasing storage capacity from nodes that want the ability to verify any incoming transaction. However, by decoupling chain verification from data verification, they allow node operators to choose what transaction data to store. Crucially, syncing a new ATAN does not require any transaction data at all, only the much smaller so-called *ATAN data*. Syncing a new ATAN requires nothing but the ATAN data (*not* transaction data) of a *non*-trusted existing ATAN, as well as access to a trusted standard (that is, pruned) Kaspa node (which the ATAN operator can run by themselves). Once synced, an ATAN operator can validate arbitrary transactions by themselves. Given several transaction, the ATAN data can validate not only that they were included into the chain, but in what order they were included. The process of obtaining transaction data (opposed to ATAN data) becomes completely trustless.

Leaves handling the actual transaction data at the discretion of the node operator allows a myriad of designs, all compatible with each other. Some might opt to store the entire transaction data since they were launched, others might choose only to store particular data they are interested in (e.g. only transactions pertaining to a particular service), while others might choose not to store any transaction data at all, requiring users to provide transaction data instead. ATANs also offer new, arguably more sensible designs for archival nodes. Since the transaction data is not required for verification, archival nodes could become ATAN nodes with separate, slower, and cheaper storage for old transaction data, only pulling it on demand.

The ATAN design requires a clever, gentle modification to Kaspa's header structure. This modification was proposed by igra in Kaspa Improvement Proposal 15 ([ZM25]) and implemented in the Crescendo hard-fork, deployed in May 5th 2025.

Remark. An additional small change, namely adding the block's DAA score to the sequencing commitment, will allow removing header data completely for igra network ATANs, as that is the only required information in the header that is not part of the transaction body. This change

[17] Deep down, this is a consequence of the possible dependencies between transactions. In a transactional layer, two transactions could be either independent (in which case you process both), or conflicting (in which case you discard one and process the other). The key point is that if both transactions are valid, then you are guaranteed that processing them in either order will result in the same UTXO set, which no longer holds for more complicated states.

will hopefully be introduced in Kaspa's next hard-fork.

A.1 Transaction Sequencing Commitment

In this section, we consider the following problem: given a transaction txn, how can we *know* that it was accepted and processed by the Kaspa network? Furthermore, given two processed transactions, how can we determine the order in which they were processed? In other words, what is the minimal data we have to store to be able to validate inclusion and sequencing order on the base layer?

For the first problem, we look to the headers. Recall that each Bitcoin block stores the root of a Merkle tree that stores all transactions within. This allows keeping the block header small, while also being committed to a given set of transactions. Kaspa blocks also keep such a Merkle tree, but they also keep another Merkle tree called the *Accepted ID Merkle Root* (AIDMR). The AIDMR contains, for each block, the set of approved transactions in its anticone (including its selected parent, but not including the block itself). It is calculated by computing the GHOSTDAG ordering of the merge-set, extracting the list of approved transactions, and arranging them in a Merkle tree. In particular, say we *know* that some AIDMR appears on a header of a block *on the selected chain*, and we are given a transaction txn along with a Merkle proof that it is an element of AIDMR, then we *know* that the transaction was processed on the Kaspa network. So the first question is: how can we tell that an AIDMR appeared in the selected chain?

One direct approach for that is to store all headers of the selected chain (recall Section 2), going back sufficiently so that all transactions you might want to verify are in the AIDMRs of these headers. For example, an igra node might want to have selected chain headers going as back to the igra launch. Other ATANs might be satisfied with shorter periods. For examples, nodes that only track a particular token on igra don't care about selected chain headers created before the token was deployed.

Currently, any ATAN design for an archival node must store the entire selected header chain, as this is necessary for validating any transaction.

Given any (sufficiently new) transaction txn, the selected chain allows us to verify that txn was in the network, but we need the Merkle proof for that. The ATAN design refrains from requiring the Merkle proofs by storing all transaction *hashes* as well. This is most of the storage bulk. With it, an ATAN can verify the set of transaction IDs by computing the corresponding Merkle root and checking that it matches the one in the header.

At this point in the specification, we can verify any (sufficiently new) transaction, which completes the first goal.

Now say that we have the data for two transactions, txn and txn', can we tell which one was processed first? If they are included in two *different* AIDMRs, this is easy: the earlier transaction is the one whose AIDMR was posted to an earlier header. But if they are both in the same AIDMR, we run into a problem.

The Merkle root induces an ordering on its elements, as reordering them will result in a different root. If the Merkle ordering is the same as the order in which transactions were processed, our problem is solved. But this was not the case before Crescendo. Instead, the developers decided to order transactions by increasing hash value, completely discarding the ordering process[18] *This* is what KIP-15 fixes. It changes the definition of an AIDMR to be the Merkle hash of transactions *in processing order*.

We could call this AIDMR a sequencing commitment, but it is more useful to compose sequencing commitments on each other. Hence, given a block B we define its sequencing commitment as:

[18] The motivation was that it makes it easy to post a proof of *non*-inclusion. A feature that has never found use in any application.

$$B.\mathsf{SeqCom} = \mathsf{H}(B.\mathsf{AIDMR}, B.SelecedParent.\mathsf{SeqCom}).$$

We can think of $B.\mathsf{SeqCom}$ as a huge, highly uneven Merkle tree that contains *all* accepted transactions in B's past.

A.2 ATAN data

An ATAN is parameterized with an *anchor* block, the earliest selected chain block for which we can validate transactions. We divide the type of data stored by a chain into two part: *validation data* that must be obtained trustlessly, and provides whatever information is needed to authenticate transactions (perhaps relative to some additional proof data), and *transaction data*.

The idea is that we only need trustlessness guarantees for validation data. Once this data is present and authenticated, transaction data can be authenticated locally, and so it can be obtained from non-trusted sources.

The underlying assumption is that we always have access to a trusted *Kaspa* node. That is, we can assume that we have access to all data stored by full nodes, and that the base layer authenticates this data. ATAN data is any subset of the entire *archival* data that satisfies:

- It can be validated from the full node data
- Once validated, it can authenticate *any* transaction accepted above the anchor (maybe requiring additional verification data of constant size)

There are several ways to choose the ATAN data, each with slightly different trade-offs (where we usually trade off the ATAN data size to allow verifying more properties). But all these choices rely on the so-called *posterity chain*. A sparse subset of the selected chain headers that *all* Kaspa full nodes are required to store. The headers in this chain are called *posterity headers*[19]. Posterity headers are useful as periodical "anchors" from which many cryptographic proofs can be verified retroactively. Each posterity header points to the previous posterity header[20], making the posterity chain verifiable all the way from the current tip to genesis block. Pre-Crescendo, Kaspa nodes stored a posterity header once every 24 hours. The Crescendo hard-fork reduced this to 12 hours. One guarantee that will remain even if pruning and posterity blocks are decoupled is that the *second latest* posterity block and its entire future were not pruned. That is, ordinary Kaspa nodes can validate everything that happened above the second latest posterity block. Let P be the latest posterity block. The property above is required for the *second latest* posterity block, as guarantees that the Kaspa node could validate anything over P even if (one) new posterity block was created during the sync process. Let S be a posterity block old enough to have the origin block in its closed future (so S is possibly the origin block itself).

From here, there are two approaches to complete the ATAN data:

- Header-based ATAN
  - Data: All selected headers from P to S.
  - Authentication: for any B validate that $\mathsf{H}(B) = B.SelectedParent$[21].

[19] Though they are often also referred to as *pruning headers*, as the pruning headers currently double as posterity headers (though this coupling is a convenience that might change in the future)

[20] Actually, the reality is slightly weirder: each posterity block points to the next posterity block, skipping two posterity blocks and pointing at the third one. This is a consequence of using pruning blocks for posterity, which is easy to work around. We ignore this subtlety.

[21] We use the fact that given a header, we know which of its parent is the selected parent, is at is always the first parent in the first level.

  - Transaction inclusion/order verification: immediate, but requires that transactions are accompanied by Merkle proofs.

- Hash-based ATAN
  - Data: For all B from P to S, the IDs of all transactions accepted in B
  - Authentication:
    - For all B from P to S, use the hash data to build a candidate $B.\mathsf{AIDMR}'$ for the block B (note that we typically don't know the header of B).
    - For all B from S to P, excluding S, compute

      $$B.\mathsf{SeqCom}' = \mathsf{H}(B.\mathsf{AIDMR}', B.SelectedParent.\mathsf{SeqCom}).$$

    - Verify that $P.\mathsf{SeqCom} = P.\mathsf{SeqCom}'$.
  - Transaction inclusion/order verification: immediate, since we now hold a complete, ordered, authenticated list of hashes of all processed transactions since origin.

A header-based ATAN is much more efficient in both communication/storage and computational complexity. However, it has the disadvantage of requiring Merkle proofs for validating transactions. This is problematic from a user experience perspective. Unlike the transaction data, the Merkle proof cannot be computed locally, and is only known once the transaction has been included and confirmed. Naturally, standard wallets do not provide this functionality. Furthermore, once a transaction is pruned, it is impossible to compute its Merkle root retroactively from a Kaspa node.

There are several other things that we might want to authenticate (and indeed, these are precisely the things we *need* to authenticate in our particular ATAN use case, namely smart-contracts) that require additional data. The key observation is that in this point we can *authenticate this data locally*, and do no longer need to trust the source of the data.

- Transaction *exclusion*: verify that a transaction was *never* accepted. Possible in hash-based ATAN.

- Non-censorship: verify that *all* relevant transactions are available. Possible in hash-based ATANs, but requires all transaction *data*.

- Consensus data: verify that some fields of the header of the block containing the data satisfy some conditions. Possible in header-based ATAN.

An igra ATAN requires all of the above, so it obtains both the header data and the transaction data. However, the only reason it needs consensus data is for authenticating the DAA score of each block. In the next hard-fork, the definition of SeqCom will be slightly adjusted to:

$$B.\mathsf{SeqCom} = \mathsf{H}(B.\mathsf{AIDMR}, B.SelecedParent.\mathsf{SeqCom}, B.DAAScore),$$

allowing an igra ATAN to be completely hash-based.

We conclude with a rundown of the advantages ATANs provide, in the context of the igra network:

- Decouples ATAN data from untrusted data. The ATAN data requires the minimal amount of trust: access to a trusted *Kaspa* node.

- The amount of data a node needs to retain to function is small. After obtaining and authenticating the data, the ATAN can store all igra transactions and their DAAs, and discard the remaining non-ATAN data.

- In particular, a situation where there is a relatively small number of archival nodes or ATANs that store and provide all transaction data does not increase trust assumption: *any* ATAN can provide enough data to *authenticate* the archive, so the archival node need not be trusted.

- The clear logical separation between obtaining authentication data and obtaining authenticated data furnishes flexibility that could be tailored for many node configurations (e.g. light clients, widgets, and so on).

B Block Construction Flow

This section discusses in some detail the flow of the block creation proccess between the ViaDuct (see Section 3.3) and IgReth (see Section 3.4). The overview is not a full specification, but it does go deeper into the details to illuminate the general structure of the igra network. In particular, there is no discussion of handling reorgs, though such a discussion will appear in a near-future version.

B.1 Technicalities

B.1.1 Block Windows

An igra block does not represent a single base layer block, but rather the transactions accumulated within a succession of blocks. A *block window* is the set of all layer one blocks processed by ViaDuct between two consecutive igra blocks. The ViaDuct controls how block windows are partitioned, as it is the component responsible for issuing build block commands to IgReth.

ViaDuct attempts to choose the window to represent a fixed amount of *time*, currently set to one second. Note that this partition must be *in consensus*, so it cannot appeal to any clocks and must rely completely on ledger data.

The "clock" ViaDuct uses is the DAA_score (recall Section 2). Since the network is regulated to produce ten blocks per second, the difference between DAA scores should provide an approximation of the difference between the *times* the blocks were created (given in tenths of a second). While this approximation has some issues, most of them are irrelevant when considering only blocks on the *selected chain*.

In particular, while two blocks may have the same DAA score, the DAA score along the selected chain is monotonically increasing. More precisely

$$B.\mathsf{DAA\_score} = B.SP.\mathsf{DAA\_score} + 1 + B.\mathsf{Blue\_Anticone\_Size} \quad .$$

Stated differently, the DAA score counts the number of blocks rolled into the selected-chain block's selected transaction Merkle root (including its selected parent, which is way the DAA score must strictly increase along the selected chain).

To ground the window to the base layer, we call the first block in the window an *anchor*. To aim for windows of, say, one second, ViaDuct can ask IgReth to construct a new block whenever the processed block DAA score is larger than the anchor block's DAA score.

B.1.2 Reorg Handling Vs. Confirmation Delay

In the previous section, we referred to the "processed block" without specifying which block that is. One might be inclined to think that it must be the selected tip, but this is not quite the case.

While we are not covering the procedure in this version, we must account for *reorg complexity*. The procedure of resolving a reorg is naturally more exacting than just processing an incoming block. Yet, in Kaspa, the block delays are much shorter than the network delay, making shallow reorgs of the selected chain block a very common occurrence.

This induces a natural trade-off. If we process blocks at a distance[22] d from the selected tip along the selected chain. Choosing a small d will incur a lot of reorg overhead. Increasing d slightly will potentially reduce this overhead without harming the finality (since any agreements will be reorged away eventually). However, after some point d starts to introduce its own delay. At an extreme example, if reorgs happen to *all* selected tips, then setting $d = 2$ will provide great performance improvement at no delay cost. On the other hand, setting $d = 100$ will avoid almost all reorgs, but introduce an unacceptable 10-second delay before processing even *starts*. Worse yet, the sweet spot d is a variable quantity that depends on network conditions.

The key is to push d to be as small as possible by making reorg handling as efficient as possible, while applying heuristics that avoid choosing d unnecessarily small. This sort of optimization requires testing under realistic condition, and is one of the goals of the caravel testnet.

B.1.3 DAA Time

Reth makes strict assumptions about timestamps. First, they are integer values given in units of one second. And second, they have to be *strictly increasing*. This assumption permeates in many subtle ways into the Reth logic, and tinkering with it might open a Pandora's box of unexpected side effects.

Hence, it is essential to design the block construction in igra such that the strictly increasing timestamp assumption is preserved.

Taking the timestamps of the layer one blocks (say, the first in the window) does not cut it. The window might be slightly shorter than a real-time second, and even when it isn't, clock drifts can distort the timestamps to appear as though it did.

So instead, the ViaDuct uses the anchor DAA score in the window as the timestamp. It is important to remember that the value of an igra block ”time”stamp, while related to the block creation time, does not represent it directly.

B.1.4 Transaction Dependencies

The dependencies of a transaction txn are the transactions that must have already been processed for processing txn to be possible. When we say a transaction is *dependent*, we particularly mean that it depends on *unavailable* transactions. A transaction is *independent* if *all* its dependencies are either in the mempool or were already included in previous blocks.

When a transaction is added to the mempool (either from the dependency queue or directly from an RPC request), IgReth *always* checks the dependency queue for transactions that have become independent, and adds them to the mempool as well. Note that this is a *recursive* process[23].

B.1.5 IgReth Queues

Reth has several queues for storing transactions, depending on their status. Of these queues, only two are relevant to IgReth: the *mempool*, which contains transactions ready to be included, and the *dependency queue*, which contains transactions that depend on transactions that have not yet been processed.

[22] The term ”distance” is intentionally vague, as specifying this measure is part of the problem. It could be topological distance, DAA difference, or many other things.

[23] Say that txn_2 is in the dependency queue and depends only on txn_1, which is also in the dependency queue and depends only on some txn_0, then if txn_0 is ever added to the mempool, IgReth will notice txn_1 became independent and add it to the mempool, wherby it will notice txn_2 is also independent and add it to the mempool as well

The job of the dependency queue is to track, for each transaction, the transactions it depends on. It provides us with an efficient way to track missing dependencies as new transactions are added. For the matter of this exposition, it allows us to say things like ”check whether a transaction has all dependencies” without worrying about how to do so efficiently.

Clearing the queues after each block is a privilege that makes the entire flow much easier. In particular, it removes complicated tasks such as tracking transaction dependencies over time, or retaining transactions that don't pay enough for gas in case the gas price goes down.

B.2 Block Creation Flow

In the block creation flow, the ViaDuct communicates with IgReth whenever it processes a new *base layer* block. The ViaDuct can respond in two ways: either append the content of the base layer block to the currently constructed igra block, or conclude the currently processed igra block and use the newly processed base layer block as the anchor for the new window.

Remark. The description here is a *functional*: we explain *what* is being computed and *why*, but not how. While some of the descriptions seem algorithmic, it *does not* mean that the actual computation is implemented the same way, only that it leads to the same result. The implementation details will be available in more technical accounts, such as specifications and code documentation. In particular, we freely assume that we can perform tasks such as tracking dependencies or switching between different orderings of transactions. In practice, the details are crucial for smooth operation (especially since we describe a process that happens several times a second), but are beyond the scope of this document.

B.2.1 Extending an Existing Window

We assume the currently processed block B is added to an already existing window. (The condition used by igra is that B's DAA score is larger than the anchor's DAA score by less than 10, but the current section is agnostic to how windows are delimited.)

In this case, the mempool and dependency queue might not be empty.

Upon determining that B is to be appended to the currently accumulating igra block, the ViaDuct does the following:

- Read igra transactions off the block and unwrap them.
- Run shallow verification of all transactions. (Here *shallow* means verification that does not require access to the igra state, and only depends on the form of the transaction. This includes validating the txid_mask, payload form, etc.)
- Report surviving transaction to IgReth in sequencing order using the sendTransaction RPC call. (There are future plans to implement a new RPC call for reporting a sequence of transactions at once.)

It is crucial to report transactions in sequencing order, as the ordering cannot be inferred from the *unwrapped* transactions.

For each incoming transaction, IgReth:

- If the transaction is dependent, add it to the dependency queue.
- Otherwise, add it to the mempool. (Recall that adding a transaction to the mempool implicitly triggers an update of the dependency queue).

B.2.2 Closing a Window, Starting a New One

The ViaDuct can decide it is time to finish the current igra block. This always happens when processing a potential anchor for the next blocks. The reason is a bit tricky and has to do with reorgs, so we will not get into it here[24].

In igra, the condition that triggers a new window is that $B.DAA\_Score$ is larger than the anchor's DAA score by at least 10.

To initiate a new window, the ViaDuct computes the header of the new igra block and reports it to IgReth with a engineNewPayload RPC command.

Upon receiving this instruction, IgReth computes the full previous block:

- Computes gas price. The computation is identical to EIP-1559, except it enforces a small fixed lower bound to prevent DDoS attacks enabled by minuscule gas prices.
- Set aside all transactions that do not pay enough gas.
- Repeatedly add transactions in *sequencing-dependency* order (that is, always add *independent* transaction that is most precedent according to the sequencing ordering) until the block is full.

When the process is finished, IgReth responds to the engineNewPayload. The ViaDuct verifies that the block B is still in the selected chain, and responds as following:

- If B is still in the selected chain, treat its contents exactly as it would any other base layer block during a window.
- If B is not in the selected chain, but $B.SP$ is, create a new block header whose anchor is the selected chain block that points to B[25].
- If $B.SP$ is not on the selected chain, initiate a reorg handling procedure.

In the first two options, IgReth will clear all its queues in preparation for the next block.

[24] but trying to figure it out for yourself is a great exercise!

[25] There is actually a nuance here: we additionally require that the selected chain block that points at B (assuming B is not a tip) has a DAA score larger by at most ten from the DAA score of the anchor block of the current window, otherwise we also initiate reorg handling.