Brevis: An Omnichain ZK Data Attestation Platform

Brevis Network

v1.0

Abstract.

Web2 applications often operate as walled gardens with limited data interoperability, resulting in user lock-ins and fragmented identities. Web3 decentralized applications (dApps) built on blockchains have the potential to overcome these issues. However, smart contracts, which are the core of dApps, currently lack a trust-free way to access and utilize the vast amount of data stored in the full history of multiple blockchains. To address this limitation, we propose **Brevis, a ZK omnichain data attestation platform that enables dApps to access, compute, and utilize arbitrary data across multiple blockchains in a trust-free way.**

Brevis's architecture comprises three components: zkFabric, zkQueryNet, and zkAggregatorRollup. zkFabric collects block headers from all connected blockchains and generates ZK Consensus Proofs that attest to the validity of these block headers. This component is essential for dApps to trustlessly access block headers from various blockchains. zkQueryNet is an open marketplace of ZK Query Engines that accepts data queries from dApps, processes them using the attested block headers from zkFabric, and generates ZK Query Proofs. These engines can range from highly specialized functions to generalized query languages, catering to a diverse set of application requirements. zkAggregatorRollup is a ZK rollup blockchain that acts as the aggregation and storage layer for zkFabric and zkQueryNet. It verifies the proofs from both components, stores the verified data, and commits its zk-attested state roots to all connected blockchains, allowing dApps to access the attested query results directly in their on-chain smart contract logic. This modular approach allows for flexibility in query engines and reduces costs by minimizing communication overhead and enabling cross-chain and cross-application query results sharing. Brevis has wide-ranging use cases, including data-driven DeFi, zkBridge, novel user acquisition solutions, zkDID, social account abstraction, and more.

In the initial Proof of Concept of Brevis, we built some of the fastest ZK light client circuits as parts of zkFabric for Ethereum PoS, Cosmos and BNB Chain to enable any EVM and non-EVM chains to access states of these three chains in a completely trust-free way. Using these ZK light client circuits, we implemented a user-facing asset zkBridge between Ethereum Goeril and BNB Chain to enable trust-free cross-chain bridging.

Archived copy · cryp2nova.com

Contents

1 Introduction **4**

2 Background **8**

2.1 Web 2.0: data in walled gardens 8

2.2 Web 3.0: great promises, disappointing reality 9

2.3 Computation Migration Using Zero-Knowledge Succinct Proof 10

3 Overview of Brevis by An Example **11**

4 Brevis zkFabric **13**

4.1 Existing Interoperability Solutions 13

4.2 zkFabric: Interoperability via ZKP-based Light Client 14

4.3 Example Light Client Implementations 16

4.3.1 Ethereum PoS 16

4.3.2 Cosmos Tendermint 18

4.3.3 BNB Chain 19

5 Brevis zkQueryNet **20**

5.1 Agent Smart Contract and Application Design Pattern 21

5.2 ZK Query Engine Network 23

5.3 Keeper Network 24

6 Brevis zkAggregatorRollup **25**

6.1 System Overview 26

6.1.1 System Components 26

6.1.2 Data Store 27

6.1.3 Interfaces with Other Modules 28

6.1.4 Rollup Fee Model 29

6.1.5 Basic Workflow 29

6.2 Rollup Sequencer 30

6.2.1 Insert Block Headers 30

6.2.2 Insert Query Results 31

6.2.3 Fetch Attested Data 32

6.3 Rollup Prover Network 33

Archived copy · cryp2nova.com

6.3.1 Single Rollup Block Proof 33
6.3.2 Multi-Rollup-Block Aggregation Proof 35

7 Use Cases **35**

7.1 ZK Multi-blockchain Interoperability 35
7.2 Data-driven DeFi 36
7.3 User Acquisition with Trust-free Revenue Sharing 36
7.4 zkDID 37
7.4.1 Prevent Sybil Attacks 37
7.4.2 User Life Cycle Management 38
7.5 Account Abstraction 39

8 Conclusion **39**

CRYP2NOVA

Archived copy · cryp2nova.com

1. Introduction

Web2 applications, despite being built on open protocols like TCP/IP, HTTP, and HTML, often operate as walled gardens. User data is stored in centralized databases controlled by the application providers, which limits the ability of users to control and share their data across different applications and services. This leads to issues such as financial and social data lock-ins, fragmented identities, and limited interoperability and composability.

Web3 decentralized applications (dApps) have the potential to break these information silos by being built on top of blockchains, which have append-only and publicly accessible data storage. With the growing adoption of dApps and multi-blockchain expansion, L1 blockchains and L2 rollups have accumulated a wealth of raw data, such as asset transfers, contract function calls, in-contract events, and blockchain state roots, which makes it possible to extract valuable information such as ownership of assets, user activity profiles, social graphs, financial connections, market pricing trends, trading volumes and more. Intuitively, blockchains application should be able to easily access and utilize these omnichain data insights in a completely trust-free way, because after all, these dApps are "native residents" inside of blockchains.

However, in reality, Web3 dApps simply have no way to access the vast majority of the data stored in the blockchains in a trust-free way. This is because smart contracts deployed on a single blockchain can only access data: (1) via explicitly defined interfaces of other smart contracts; (2) on the same blockchain; (3) in the most current state instead of with a complete historical view. Solutions based on off-chain Oracles [3] introduce unnecessary external trust assumptions, and direct on-chain computation by expanding state roots is prohibitively expensive. This issue is aggravated by the recent explosive growth of the multi-blockchain ecosystem. Therefore, it is impossible today to build a dApp whose internal business logic relies on information extracted and computed from arbitrary data residing on a combination of blockchains.

To lift such limitations and further unleash the potential of Web3 applications, we propose Brevis, a ZK omnichain data attestation platform that enables dApps to access, compute and utilize arbitrary data across the full history of multiple blockchains by leveraging the power of Zero-Knowledge (ZK) succinct proof.

As shown in Figure 1, Brevis's architecture consists of three main components: zkFabric, zkQueryNet, and zkAggregatorRollup. zkFabric collects block headers from all

Archived copy · cryp2nova.com

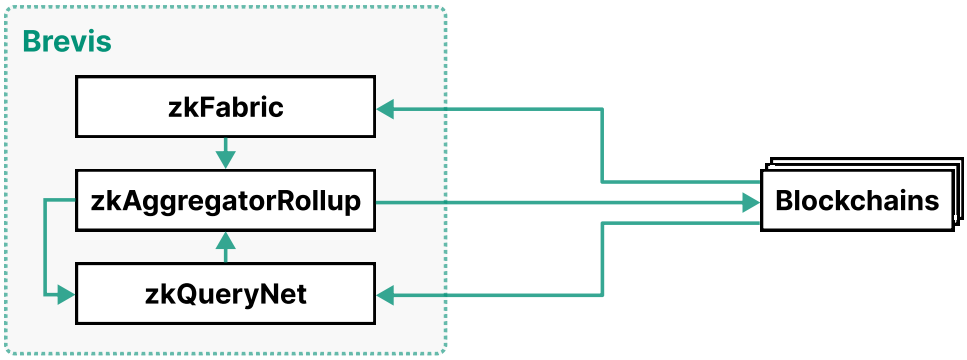


Figure 1. Brevis High Level Architecture

connected blockchains and generates ZK Consensus Proofs that attest to the validity of these block headers. These block headers are further zk-verified and stored in zkAggregatorRollup. zkQueryNet accepts data queries from dApps and generates ZK Query Proofs based on the attested block headers stored in zkAggregatorRollup. These query results are also zk-verified and stored in the zkAggregatorRollup. Essentially, zkAggregatorRollup is a ZK rollup that acts as the aggregation and storage layer for zkFabric and zkQueryNet. By committing its zk-attested state roots to all Brevis connected blockchains, zkAggregatorRollup allows dApps to access attested query results and utilize them directly in its on-chain smart contract logic in a trust-free way.

With this high-level overview, we now walk through each component.

First, to utilize arbitrary data across multiple blockchains, dApps need to have a trust-free way to access block headers from blockchains other than its native chain. This is because block headers contain state roots that can be used to access data and states in blockchains. To fulfill this need, **zkFabric** is introduced to generate ZK Consensus Proofs for block headers of all the supported chains. A Consensus Proof is generated by a light client circuit proving that the block header in question is generated according to the consensus rule of the corresponding blockchain. zkFabric itself is a decentralized system consisting of a network of block header relayers and provers as discussed in more detail in Section 4.

To actually extract valuable information from the zk-attested block headers, **zkQueryNet** is built as an open marketplace of ZK Query Engines that directly interface with dApp developers and smart contracts. A dApp developer can choose a ZK Query Engine that fits her need and write queries via high-level APIs in smart contracts.

Archived copy · cryp2nova.com

At run time, a smart contract can call the Agent Smart Contract of zkQueryNet targeting a specific ZK Query Engine. This query will be picked up by the Provers of this ZK Query Engine. Using the already zk-attested block headers provided by zkFabric, the ZK Query Engine can then compute the query results and generate a ZK Query Proof attesting that computation is done correctly.

Different ZK Query Engines can have very different APIs ranging from generalized query languages to highly specific function calls with a fixed number of parameters. For example, on one side of the spectrum, a specific ZK Query Engine may only expose a function that accepts two block numbers and two chain IDs and will return the Time Weighted Average Price for ETH/USDC pair during the specified time period on Uniswap on the specified two blockchains. On the other side, a generalized ZK Query Engine can present developers with a blockchain indexing abstraction using high-level database queries such as SQL or GraphQL, which is very much like what they can do in off-chain data solutions such as Dune Analytics [10] and Graph [11].

Brevis will provide a set of ZK Query Engines that address many immediate use cases with reasonable flexibility and high performance. As zkQueryNet is an open marketplace, we expect dApp developers and other third parties to provide other ZK Query Engines to better serve a diverse ecosystem of applications.

Finally, **zkAggregatorRollup** is a ZK rollup blockchain powered by a lightweight ZK virtual machine that aggregates different proofs and their inputs from zkQueryNet and zkFabric. Specifically, zkAggregatorRollup VM runtime has the following functionalities: (1) recursively verify proofs generated by zkQueryNet and zkFabric; (2) store the zk-verified block headers from zkFabric; (3) store queries and the zk-verified results. When plugging in a ZK Query Engine or adding a new type of consensus, zkAggregatorRollup will be extended to support the verification of the corresponding ZK proofs. Moreover, unlike many ZK rollup chains where the proofs of state root progression are only committed to one single blockchain, zkAggregatorRollup's state root proofs are committed to all the blockchains supported by Brevis.

With zkAggregatorRollup's state roots available on all connected chains, smart contracts can access the query results and block headers via data inclusion proofs. The primary benefit of using zkAggregatorRollup as an aggregation point is to reduce the block header communication/verification overhead from $O(N^2)$ to $O(N)$ (where N is the number of blockchains supported by Brevis), and to share query results

Archived copy · cryp2nova.com

on-demand efficiently across all connected blockchains.

To sum up, Brevis has the following key advantages.

- **Trust-free**: Brevis does not rely on any off-chain party to attest data and computation integrity; instead, it relies solely on ZK succinct proofs. Therefore, applications using Brevis do not need to post any additional trust assumptions other than those of underlying blockchains and cryptographic protocols.
- **Omnichain**: Brevis integrates with multiple blockchains running on different consensus and therefore enables an omnichain data access and computation.
- **Modularized**: Brevis uses a highly modularized design in its zkQueryNet and therefore can cater to a wide range of application requirements through different flavors and implementations of ZK Query Engine.
- **Low-cost**: Brevis's zkAggregatorRollup essentially works as a batching and aggregation layer for block headers and query results. Therefore, zkAggregatorRollup significantly reduces the on-chain costs by removing the otherwise N-to-N communication overhead and by enabling cross-chain and cross-application query results sharing.

It is important to note that the key difference between Brevis and off-chain data indexing solutions such as Dune Analytics [10] and Graph [11] is that Brevis can generate zk-attested query results that can be directly utilized by the business logic of on-chain smart contracts in a trust-free way while off-chain solutions' data results can only be used in the web2-based data analytic context. Brevis has a wide range of use cases such as data-driven DeFi, zkBridge, novel user acquisition solutions, zkDID, social account abstraction, and many more.

In the initial Proof of Concept for Brevis, we built some of the fastest ZK light client circuits as parts of zkFabric with gnark for Ethereum PoS, Cosmos, and BNB Chain to enable any EVM and non-EVM chains to access the states of these three chains in a completely trust-free way. Some of the key circuit performance benchmark numbers are summarized in Table 1 (using a Linux Server with 20 cores@2.3GHz and 384GB of memory without any GPU acceleration). Using these ZK light client circuits, we implemented a user-facing asset zkBridge between Ethereum Goeril and BNB Chain.

In the rest of the white paper, we will discuss the background, motivation, and intuition for Brevis in Section 2 and provide a system overview via an example in Section 3. Then we will dive into the design and implementation of zkFabric, zkQueryNet, and

Archived copy · cryp2nova.com

	#Constraints	Prove+Witness Gen. Time
Batch of 8 Ed25519 Sig Verifications (over BN254)	7.4m	15s
BLS12-381 Sig Verification (over BN254)	8.7m	25s
SSZ Sync Committee Update	88m	72s

Table 1. Benchmark results for the key circuits used in the light clients of Ethereum PoS, Cosmos Tendermint and BNB Chain. The circuit is implemented in gnark. The machine is a Linux server with 384GB memory and 20 cores@2.3GHz.

zkAggregatorRollup in Sections 4, 5, and 6, respectively. Finally, we will discuss some use cases in more detail in Section 7.

2. Background

2.1. Web 2.0: data in walled gardens

Web 2.0 applications have transformed the internet landscape, empowering users to generate content and interact in ways previously unimagined. However, despite the open and layered protocols underlying Web 2.0, many applications operate as walled gardens where user data is stored in centralized databases owned and managed by the platform providers. This presents a multitude of challenges for both developers and users, such as limited data interoperability, fragmented identities, and financial and social data lock-ins, leading to various consequences such as stunted innovation, vendor lock-in, privacy concerns, limited transparency, and fragmented user experiences.

Take an example of financial data lock-ins, where users are unable to transfer their financial data to other services or platforms. A user may have their credit card information stored on an e-commerce platform but be unable to transfer this data to another platform, such as a budgeting app. This creates a barrier to entry for new applications and services, as users may be hesitant to switch to a new platform if it means losing access to their financial data.

This lack of data interoperability does not only exist in financial applications. For example, a user may have a wealth of data about their workouts, such as their heart rate, distance run, and calories burned. However, this data is often siloed within the fitness app and not easily accessible to other applications or services. This limits the potential uses of the data, such as sharing it with a nutrition app to get personalized recommendations for meals based on the user's workout history.

Archived copy · cryp2nova.com

2.2. Web 3.0: great promises, disappointing reality

Web 3.0 holds great promise to break down the barriers of data isolation, as decentralized applications (dApps) are built on top of public blockchains. Unlike Web 2.0 databases, blockchains, which can be seen as append-only data structures, store all past transactions and historical states in the public domain.

One might argue that Web 2.0 databases can also store historical data, but it is critical to note that this historical data is stored via explicit business logic built into the application itself. If there is no such business logic, historical user actions will be forever lost. In contrast, for dApps built on blockchains, every transaction is permanently stored in the blockchain, regardless of whether the app itself has the business logic to record that history. Moreover, as blockchains progress, state snapshots of each block are contained in the block header in succinct digest forms such as Merkle Roots.

Many products and projects are already leveraging this unique property. Products like Dune Analytics [10] and Graph [11] provide **off-chain data indexing** or data analytics for blockchain applications across time. They can also be used to provide stateful data, such as user transaction history, for dApps' frontend UIs. These applications access blockchain data in **an off-chain way**, recording, indexing, and computing data via blockchain nodes' RPC endpoints.

With full blockchain data access being this simple for **off-chain programs**, one would assume that smart contracts, which are **on-chain programs** acting as the core component of decentralized dApps, should also be able to easily access this complete set of blockchain data. After all, smart contracts are "native residents" of blockchains. **However, this is not the case.**

The scope of a smart contract's data access is highly limited and, in fact, resembles the pattern of Web 2.0 applications today. To understand this counterintuitive fact, we need to discuss the important concept of virtual machine runtime and smart contract life cycle. When a smart contract is deployed on a blockchain, its life cycle starts in the context of the virtual machine runtime. The virtual machine's system clock moves forward monotonically as new blocks are built. Therefore, the smart contracts running in this virtual machine will **only have visibility of blockchain states in the present time**. Additionally, an on-chain smart contract can only access data that is **either internal or explicitly defined as a function**

Archived copy · cryp2nova.com

call in another smart contract. Finally, because a blockchain VM can only access states and execute programs within the same blockchain, smart contracts have **no way to access or utilize data from any other blockchains in a trust-free way**.

To make a Sci-Fi analogy, we can think of on-chain smart contracts as three-dimensional beings, such as humans living on Earth. A human always lives in the present time and cannot travel back in the past. Furthermore, a person cannot get information about another person without having the other person agree to share the information. Finally, a person cannot see the events happening in another parallel universe. Comparatively, off-chain programs are like higher-dimensional beings that have visibility back to the past, into every fact, and to other alternative universes. However, they are separated from the on-chain world and can not drive the core logic of dApps today.

One might argue that there are several possible paths to solve these challenges. For the same blockchain, it is theoretically possible for a smart contract to parse data query semantics, access state roots directly, and compute the necessary query results by expanding the state tree on-chain. However, in practice, this is prohibitively expensive due to the extremely high on-chain computation costs. Similarly, it is theoretically possible to build a trust-free light client that validates if block headers from a different blockchain are generated via the proper consensus rules. However, in practice, this is impossible due to the same cost concerns.

So, how can we allow smart contracts to access and compute data from any blockchains over an arbitrary period of time?

2.3. Computation Migration Using Zero-Knowledge Succinct Proof

Zero-knowledge succinct proof technology (ZKP) is a rapidly evolving area of cryptography research with the potential to revolutionize our interactions with digital information. With ZKP, one party can prove to another party that they have correctly generated a computation result based on an input value, without revealing any information about that value. Moreover, the other party only needs to run a computationally inexpensive program, called a verifier, to be convinced that the computation is done correctly.

Apart from the privacy benefits implied in the "zero knowledge" part of ZKP, this technology can also be used to **migrate computation from a high unit cost**

Archived copy · cryp2nova.com

location to a low unit cost location.

The idea is that zero-knowledge succinct proofs allow the prover to perform a computationally expensive operation not only to complete the computation but also to generate a cryptographic proof, while the verifier can perform a computationally cheap operation to verify the proof. This effectively enables the location of computation to be shifted from the verifier to the prover. Although the total computation cost increases when using ZKP due to the added proof generation computation, the tradeoff is still worthwhile if the prover's unit computational costs are sufficiently lower than those of the verifier.

ZKP's computation migration property is the driving force behind many ZK rollups (e.g., [7, 19, 22, 23, 30]). In this paper, we leverage this property to design Brevis, which enables access and computation of arbitrary blockchain data across the entire history.

3. Overview of Brevis by An Example

Before we delve into the details of each component, we present a concrete application example to provide an overview of Brevis. Note that this overview abstracts some key details for simplicity and leaves a more detailed design in the following sections.

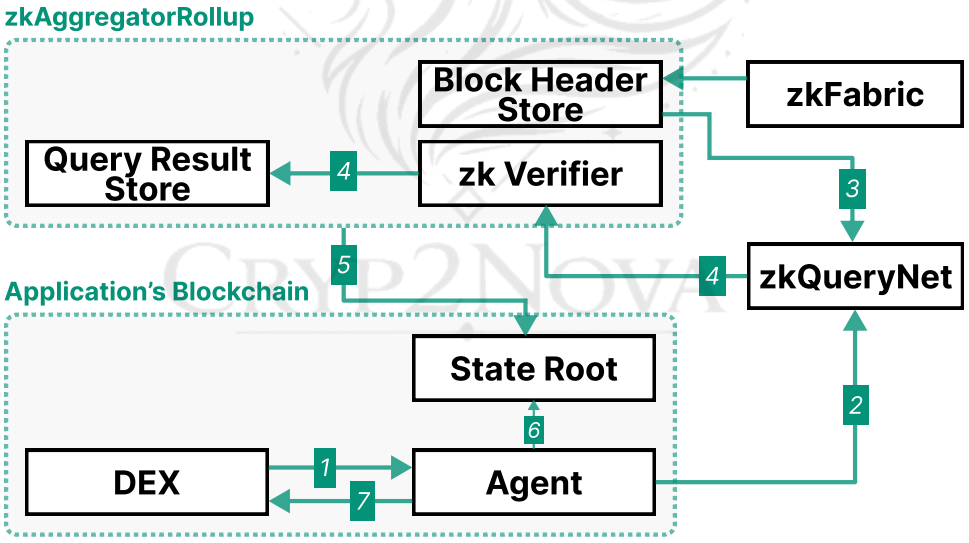


Figure 2. An Example Walk Through

Multi-blockchain decentralized exchanges (DEXes), such as PancakeSwap, often need to dynamically adjust farming rewards for incentivized pools based on trading pair quality scores such as average daily volume, 14-day volume, price volatility,

Archived copy · cryp2nova.com

number of active traders, and liquidity providers across all supported blockchains.

Currently, such adjustments must be made through governance proposals, which entail significant human overhead and can only be done when the setting deviates severely from optimal. This lag behind market trends often leads to a decline in user engagement, revenue loss, and waste of treasury funds due to sub-optimal reward configurations.

Brevis can address these challenges by enabling DEXes to adjust the liquidity farming schedule programmatically and trust-freely based on omnichain market trends. For simplicity, let us assume that volume is all we care about. Let us also assume that there is a ZK Query Engine Q in zkQueryNet that provides a set of highly optimized circuits for DEX volume data with an API that looks like the following:

```
uint_64 get_trading_volume(uint_64 chain_id,
                           uint_64 start_block,
                           uint_64 end_block,
                           address pair)
```

Figure 2 shows the step-by-step process of how this use case works. To use this data compute API, DEX smart contract first needs to make a function call to the Agent Smart Contract (denoted as A) of zkQueryNet with the above query parameters specified. Note that this function is asynchronous and only immediately returns a *query_id* so the DEX's smart contract needs to remember this ID and has a handler to process the return value later.

This function call is then picked up by Q's prover, denoted by P_q. Using the block headers for *chain_id* (which has already been verified and stored in zkAggregatorRollup), P_q will generate a ZK proof π proving that on *chain_id* during the time period from *start_block* to *end_block* the DEX's *trading_pair* indeed has *volume*.

Q's proof verifier in zkAggregatorRollup verifies π, and the result *volume*, as well as the corresponding query parameters, are stored in zkAggregatorRollup. This state is later included in a state root S of zkAggregatorRollup and committed to the chain on which the DEX is deployed.

Now, A can retrieve and verify the query result by using a state inclusion proof against S. This retrieved query result will then be returned back to the DEX's smart contract's handler function. The handler function will be able to match the stored

Archived copy · cryp2nova.com

query_id with the returned query result and adjust the liquidity farming schedule based on the volume data accordingly.

With this example as a high-level walkthrough, we will now dive into the detailed design of each component in the following sections.

4. Brevis zkFabric

In Brevis, zkFabric serves as the trust-free interoperability layer that enables any blockchain to access attested states of every other blockchain. zkFabric's **zero-knowledge proof (ZKP) based light clients** makes it fully trust-free without relying on any external validation entity, because its security is solely derived from the underlying blockchains and mathematically sound proofs.

In the following sections, we will briefly survey existing interoperability solutions and their limitations in Section 4.1, provide an overview of the architecture and technology used in zkFabric in Section 4.2, and discuss implementation details of building zkFabric in Section 4.3 with a few consensus protocols as examples.

4.1. Existing Interoperability Solutions

Blockchain interoperability has emerged as a prominent area of research in recent years, leading to the development of several classes of architectures in production. In this section, we briefly discuss various multi-blockchain interoperability solutions and their limitations. Please note that the list in each class is not meant to be exhaustive.

- **External Validation Bridge** achieves interoperability among blockchains by using an external entity sitting in between different blockchains. This external entity undertakes the role of validating and relaying messages from one chain to another. Therefore, the security of cross-chain communications solely depends on the external entity's designs, such as multi-signature validation (e.g., Wormhole [28], Ronin Bridge [21]), multi-party computation threshold signature (e.g., Multichain [15], Thorchain [26]), relay and oracle co-validation (e.g., LayerZero [layerzero]), and Proof-of-Stake (PoS) based middle chains (e.g., Axelar [1], Celer's default security model [2]). External validation bridges are often decentralized but not trust-free because the external entity can still modify the cross-chain

Archived copy · cryp2nova.com

messages when acting maliciously.

- **Optimistic-like Bridge** is a variation of the external validation model's design by incorporating a two-phase commit with a delay and optionally source-chain slashing functionality (e.g. Nomad [17] and Celer's optimistic model [2]). Specifically, an optimistic bridge system requires the external validation layer to deposit collateral and relies on a watchdog service to continuously monitor the bridge state and penalize malicious actors by slashing their collateral upon detection. However, an optimistic bridge introduces a considerable delay due to the challenging period between commitment and confirmation. Moreover, its security still depends on the deposited collateral: if the value in transit exceeds the collateral, there are still incentives for the external validation layer to cheat. Note that Optimistic-like bridges are very different from Rollup bridges, because malicious messages can still be delivered even if a challenge is successful.
- **Light-Client Bridge** implements the entire light client logic in a smart contract on the destination chain for the source chain. Compared to the aforementioned models, the light-bridge bridge model eliminates the need for a trusted third party and is trust-free. Examples include Cosmos IBC [9] and NEAR-Ethereum Rainbow bridge [16]. However, directly implementing generalized light clients as a smart contract is prohibitively expensive. For example, verifying NEAR light client involves many Ed25519 signature verifications but a single Ed25519 signature would require 500k gas to verify in EVM. Thus, light-client bridges such as NEAR-Ethereum Rainbow bridge use an optimistic model that skips the Ed25519 signature verification and relies on a watchdog to monitor (similar to the previously-mentioned Optimistic bridge), resulting long delay (4 hours) and a weakened security model.

4.2. zkFabric: Interoperability via ZKP-based Light Client

zkFabric achieves trust-free interoperability, the same as the light-client bridge model, and overcomes its implementation challenges and long delay issues by implementing light-client protocols in polynomial arithmetic circuits. Using these circuits, zkFabric can generate ZK Consensus Proofs which prove block headers are indeed generated through the corresponding chains consensus protocols. These Consensus Proofs can then be verified efficiently and recursively through zkAggregatorRollup, on all con-

Archived copy · cryp2nova.com

nected blockchains.

Figure 3 shows the architecture of zkFabric which consists of two components:

- **Block Header Relay Network**: This network synchronizes block headers from various blockchains, forwarding them to the zkFabric prover network to generate validity proofs. Subsequently, it submits the block headers and associated proofs to zkAggregatorRollup (refer to Section 6 for further details on zkAggregatorRollup).
- **zkFabric Prover Network** implements circuits for each blockchain's light-client protocol, this network generates validity proofs for block headers. Provers can leverage accelerators, such as GPUs, FPGAs, and ASICs, to minimize proving time and cost.

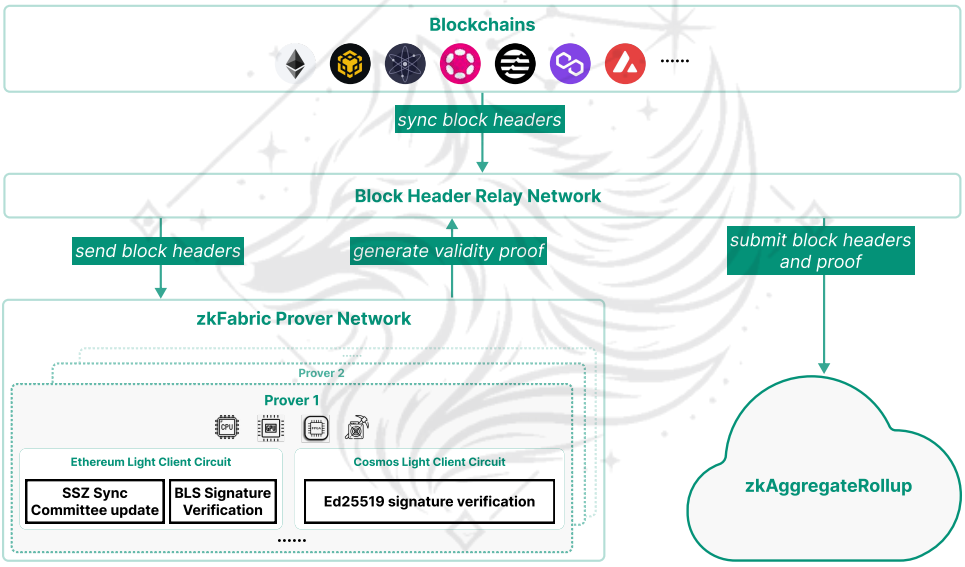


Figure 3. Brevis zkFabric Architecture

The zkFabric workflow comprises the following steps:

(1) The block header relayer network periodically synchronizes block headers from various blockchains. To avoid fork choices, the network only selects a block header after a safe block confirmation delay has passed. Additionally, external triggers (e.g., on-chain events or user requests) can initiate block header synchronization.

(2) The block header relayer network submits block headers to the zkFabric prover network.

(3) The prover network generates a validity proof π for the block headers, utilizing

Archived copy · cryp2nova.com

a chain-specific ZKP circuit that implements the light-client protocol. Section 4.3 will provide some example implementations.

(4) The prover network returns the validity proof π to the relayer network.

(5) Upon receiving the proof, the relayer network sends it and the corresponding block headers to zkAggregatorRollup using the `InsertBlockHeaders` interface. Section 6 will specify details about this interface.

Trust Assumption. zkFabric relies on the security assumptions of blockchains and underlying cryptographic protocols, without requiring additional assumptions on any external parties. However, ensuring zkFabric's liveness necessitates at least one honest relayer to synchronize the correct fork. Therefore, we propose employing a decentralized relayer network, rather than a single relayer, to optimize zkFabric's liveness. This relayer network can utilize existing constructs such as the State Guardian Network in Celer Network. However, it is critical to note that in this case, the State Guardian Network has no way to generate malicious cross-chain messages.

Prover Assignment. The prover network, a decentralized network of ZKP provers, requires the selection of a prover for each proof generation task and the payments to these provers. This is out of the scope of this paper and separate economics specifications will be published on this and other related subjects in the future.

4.3. Example Light Client Implementations

In this subsection, we outline our current implementations of light client protocols for various blockchains, including Ethereum PoS, Cosmos Tendermint, and BNB Chain, as examples and proof of concepts.

4.3.1. Ethereum PoS

Circuit. The ZK circuit for the Ethereum PoS light client primarily comprises two components:

- *SSZ Sync Committee Commitment.* This circuit computes the SSZ commitment for the 512 validators in the sync committee, requiring 1025 SHA-256 computations over a 64-byte public key. It is worth noting that SHA-256 is not ZK-friendly due to bitwise operations, leading to a large number of constraints in our implementation. Fortunately, since the Ethereum sync committee updates

Archived copy · cryp2nova.com

	#Constraints	Prove+Witness Gen. Time
BLS12-381 Sig Verification (over BN254)	8.7m	25s
SSZ Sync Committee Update	88m	72s

Table 2. Benchmark results for Ethereum PoS Light-client Circuit implementation in gnark. The machine is a Linux server with 384GB memory and 20 cores@2.3GHz.

every 27 hours, this circuit is invoked only occasionally. However, the per-block aggregate BLS signature verification, which also necessitates calculating the sync committee commitment, is impractical at such a large number of constraints for a per-block frequency. To address this issue, we employ a technique by Succinct Labs [24], allowing the circuit to compute two commitments: one using the original SHA-256 hash function and the other utilizing a ZK-friendly hash function, such as Poseidon or Rescue hash. An on-chain mapping from the original SSZ commitment to the new ZK-friendly commitment is then stored for later use in per-block BLS signature verification.

- *BLS Signature Verification.* This circuit first calculates the ZK-friendly commitment and compares it to the on-chain commitment mapping. Subsequently, it verifies the BLS12-381 signature, involving hash-to-curve calculations and BLS12-381 pairing over the BN254 scalar field (since the initial implementation requires light client verification in EVM, where BN254 pairing is supported as a pre-compile).

We implement the circuits in gnark [6] with Groth16 instantiated with BN254. The gnark framework has proven to be highly optimized in performance, as demonstrated in industrial benchmarks [32]. **Furthermore, gnark provides an efficient implementation for range checks which we leverage to significantly reduce the number of constraints in non-native field operations for BLS signature verification.** A similar optimization for the bitwise operations in SSZ Sync Committee Commitment circuit is in progress.

Table 2 presents the benchmark results for the aforementioned circuits on a Linux server with 384GB memory and 20 cores@2.3GHz.

Light Client. Our system utilizes the latest beacon chain light client APIs, which provide both sync committee updates and finalized blocks. Proofs are submitted to on-chain light client contracts for sync committee and block signature verification. Sync

Archived copy · cryp2nova.com

	#Constraints	Prove+Witness Gen. Time
Batch of 8 Ed25519 Sig Verifications (Over BN254)	7.4m	15s

Table 3. Benchmark results for verifying a batch of 8 Ed25519 signatures (over BN254 scalar field) implementation in gnark. The machine is a Linux server with 384GB memory and 20 cores@2.3GHz.

committee updates must be submitted every 27 hours to establish a chain of trust. Blocks, on the other hand, can be updated on-demand as long as they are finalized by the current committee. To support use cases like generic message passing, we also generate a Merkle proof for the execution state root in the beacon state and submit it to the light client to persist the latest execution state root on-chain. Consequently, other applications can easily call the Ethereum API `eth_getProof` to obtain Merkle proofs for any state of interest, and the on-chain contract ensures that the state root matches the persisted root.

4.3.2. Cosmos Tendermint

Ed25519 Signature Verification Circuit. One of the most computationally expensive aspects of Cosmos Tendermint consensus is the verification of a batch of Ed25519 signatures (for example, in Ethereum, verifying a single Ed25519 signature on-chain costs approximately 500k gas). To leverage the computation migration property of ZKP, we incorporate the Ed25519 signature verification into a ZK circuit.

However, Ed25519 signature verification necessitates elliptic curve operations on Curve25519, whose scalar field is larger than BN254. As a result, the Ed25519 signature verification circuit requires non-native field arithmetic operations (it is theoretically impossible to avoid such non-native arithmetic operations if we need to verify Ed25519 signatures in EVM [33]). We implement the Ed25519 signature verification (over BN254 scalar field) circuit in gnark, where each Curve25519 element is divided into 5 limbs with 51 bits each. Additionally, we employ the efficient range check gadgets in gnark to significantly reduce the number of constraints in non-native field operations. Table 3 displays the benchmark results for our implementation on a Linux server with 384GB memory and 20 cores@2.3GHz. We anticipate further improvements in proving time for a batch of signatures through parallelization and hardware acceleration.

Light Client. We have integrated the Ed25519 verification circuit into an on-chain

Archived copy · cryp2nova.com

Tendermint light client contract based on Tendermint-sol [25]. The light client first verifies a protobuf-serialized Tendermint block header by performing basic checks that are not particularly computationally intensive. It then delegates the validation of validator signatures to the Ed25519 verification contract. In our current simplified implementation, the circuit validates signatures from 2/3 of the validators in a single proof with a constant estimated proof verification cost of 300k gas. This cost saving is significant compared to the estimated 500k gas per signature if we were to use an Ed25519 implementation in EVM, where validating 10 signatures would have cost 5M gas instead.

Once a Tendermint block header is verified, the light client stores the Merkle root of the Tendermint state tree (referred to as $appHash$ in Tendermint terms), which can then be used to verify transactions using the Merkle proof verification spec standardized by ICS-23 [13]. A particularly useful application of ICS-23 is verifying Cosmos IBC transactions in a trust-minimized manner. We will follow up with an example of the BNB chain.

4.3.3. BNB Chain

Tendermint and PoA Light Client for BNB Chain. The BNB chain adopts a two-layer architecture, comprising the base layer, referred to as the BNB Beacon Chain (BBC), and the execution layer, called the BNB Smart Chain (BSC). The BBC utilizes Tendermint consensus, while the BSC employs a Proof-of-Authority (PoA) consensus called Parlia, which is largely based on Clique [5]. Parlia features a set of authorized validators that produce BSC blocks in a round-robin fashion. Each BSC block is signed off by one of the validators in the set. As a specialized Tendermint application, the BBC manages the election and updates of the BSC validator set. Every 24 hours, a new set of BSC validators is elected and synchronized to the BSC via an IBC-like cross-chain mechanism that predates the official IBC specification. The BSC then verifies the validator set update via its own Tendermint light client, which is a system-level precompile contract implemented in Go, rendering gas costs inconsequential.

To verify a transaction on BSC on another EVM-based chain, one must implement an on-chain Proof-of-Authority light client. Since each PoA header is signed via a single secp256k1 key for which the EVM has a precompile, verifying a PoA header is not gas-

Archived copy · cryp2nova.com

intensive. However, as PoA is not Byzantine fault-tolerant on its own, we took an extra step to verify the validity of the PoA validator set using our Tendermint light client, assisted by zk-based Ed25519 verification. The BBC Tendermint header containing the "BSC validator set update" IBC package is synchronized to the Tendermint light client contract and verified, with the updated BSC validator set stored on-chain in the PoA light client. Subsequently, each BSC block header can be verified trivially by checking that it was signed by one of the validators.

5. Brevis zkQueryNet

Brevis zkQueryNet offers a comprehensive network of Zero-Knowledge Query Engines (ZQEs) that can be directly integrated into smart contracts. For application developers, zkQueryNet provides a standardized and user-friendly Agent Smart Contract API and an actor-based design pattern, enabling them to build decentralized applications (dApps) that rely on arbitrary data query computation results from any combination of blockchains. For ZQE providers, zkQueryNet offers a unified Keeper service and interfaces for their Provers and Query Services to be seamlessly integrated into Brevis.

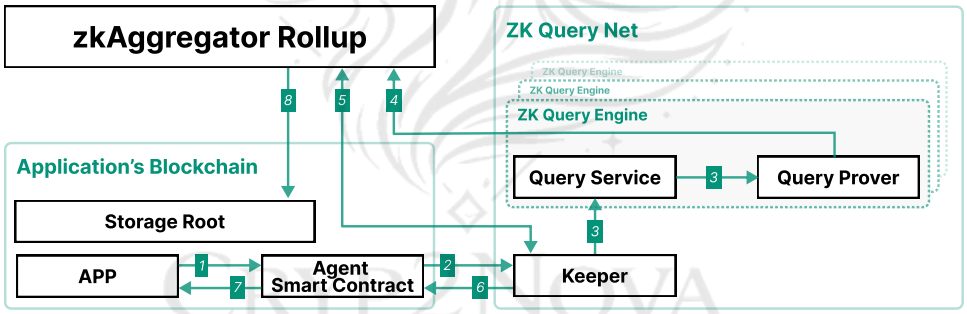


Figure 4. A Walk Through of zkQueryNet

As illustrated in Figure 4, we provide a step-by-step walkthrough of how a smart contract interacts with zkQueryNet:

(1) When a smart contract needs to obtain certain data, it issues a query by calling a zkQueryNet's Agent smart contract using a standard interface.

(2) This query is picked up by zkQueryNet Keeper, which routes the query to the corresponding ZQE.

(3) Upon receiving the query, the ZQE's Query Service parses the query and gen-

Archived copy · cryp2nova.com

erates the data computation results. Next, the ZQE's Query Prover generates a proof based on the query input, relevant block headers, and results.

(4) The Query Prover then posts the query results, along with proofs, to zkAggregatorRollup by calling the `InsertQueryResults` interface. Once verified, the query and the results are stored in zkAggregatorRollup. The details about the `InsertQueryResults` interface will be specified in Section 6.

(5) After the zkAggregatorRollup's block containing the verified query is committed to the application's local blockchain, the Keeper calls the `FetchAttestedData` interface to zkAggregatorRollup and obtains an inclusion proof for the query result, which can be verified by the Agent Smart Contract.

(6) The Agent Smart Contract then calls the handler function of the application to deliver the query results.

(7) The application smart contract executes the corresponding business logic based on the returned data.

5.1. Agent Smart Contract and Application Design Pattern

A shared Agent Smart Contract is deployed on each blockchain to act as an interface between the applications' smart contracts and other components of the zkQueryNet system. The Agent Smart Contract is shared among all ZQEs and provides the following interfaces:

```
uint256 query(uint256 engine_id,
              uint256 deadline,
              bool cache_result,
              bytes[] query_input);
bool deliverResult(uint256 query_id,
                   bytes [] query_result,
                   bytes [] proof);
bool registerEngine(uint256 engine_id);
bool removeEngine(uint256 engine_id);
```

The `query` function serves as the entry point of Brevis. Any smart contract that wants to utilize the data query and computation capabilities of Brevis must first call `query`. In the function call, a smart contract must specify the ZQE it intends to use via

Archived copy · cryp2nova.com

`engine_id`. Additionally, a `deadline` must be specified to inform other components of zkQueryNet about the optimal scheduling and to prevent stale data from being used. If the query result needs to be reused by other smart contracts on the same chain, `cache_result` should be set as `true`. Finally, `query_input` is the serialized Query API call based on the chosen ZQE. This function will first emit an event containing all the aforementioned parameters.

As a query often involves data access, computation, and ZK proof generation processes from the ZQE, it is unreasonable to expect the `query` function to return results immediately. Instead, the function returns a unique `query_id` deterministically generated based on the `query` function's calldata. Consequently, application developers must employ an asynchronous design pattern when interacting with the Agent Smart Contract, recording the `query_id` and any metadata in the application smart contracts. On the application side, a handler function with the following signature is required to handle the returned query results later:

```
bool handleQueryResult(uint256 query_id,
                       byte[] query_result)
```

When the query result is zk-verified and stored in zkAggregatorRollup, and the corresponding rollup block is committed to the application's blockchain, the Keeper will call the `FetchAttestedData` interface to zkAggregatorRollup using `query_id` and obtain the `query_result` as well as its `proof`, which can attest to the inclusion of the `query_result` in the committed rollup Verkle tree. The Keeper will then call the `deliverResult` function, which first checks if the deadline has passed. If not, the function further validates the inclusion of the `query_result` using the `proof`.

After the validation is completed within `deliverResult`, `handleQueryResult` is called with the `query_id` and `query_result`. This handler function is expected to be part of the asynchronous query flow, matching the result with the pending query and executing the corresponding business logic.

With the Agent Smart Contract concealing all the common "logistics", developers are provided with a clean and user-friendly interface to integrate their dApps with Brevis. It should be noted that the current version of the Brevis whitepaper does not cover economics and incentive mechanisms. Once a complete economics mechanism is in place, we expect the Agent Smart Contract's interface to be expanded with the functionality to commit to pricing and deadlines, along with a staking and slashing

Archived copy · cryp2nova.com

mechanism for addressing missed Service Level Agreements. We will cover the economic incentive mechanisms in a different paper.

5.2. ZK Query Engine Network

The ZK Query Engine Network forms the core of the zkQueryNet system, where queries are transformed into zk-attested results. Anticipating a diverse range of future use cases for Brevis, we have designed the ZK Query Engine Network as an open marketplace, allowing any ZK Query Engine to be easily integrated. A ZK Query Engine in Brevis comprises two parts: Query Service and Query Prover.

Query Service is responsible for defining Query APIs and generating the query results. In terms of query results generation, it operates similarly to off-chain data analytic platforms, where data is gathered via off-chain blockchain RPCs or from other data platforms. Query APIs are the distinguishing factors for different ZQEs at the application usage level. The design of Query APIs spans a wide spectrum, ranging from highly specific to highly generalized. Below, we provide some illustrative examples:

- Highly specific: `uint256 get_retention_uniswap(uint256 start, uint256 end)` retrieves the 7-day retention of active users on Uniswap for a cohort between block number `start` and `end`.
- More general: `uint256 get_active_traders_by_filter(string dex_name, uint256 start, uint256 end, byte[] filter)` returns the Verkle root of all users from the DEX with `dex_name`, between block number `start` and `end`, matching a specific filter with simple syntax. It is also possible to present a more generalized indexing abstraction akin to relational databases or graph databases and use standard query languages such as SQL or GraphQL or some Domain Specific Languages (DSLs) and light-weight VM to process and compute on the requested data.
- Highly generalized: Pushing to the other end of the spectrum, it is possible to use a fully general zk-enabled virtual machine as the processing layer. Developers can build with highest level of flexibility at the tradeoff for significant performance overhead.

Although highly generalized Query APIs are possible, we envision that most pro-

Archived copy · cryp2nova.com

duction APIs will lean toward the more specific end of the spectrum using DSLs and specific APIs. This is because there is a trade-off between the level of generalization and the performance of Query Provers. Consequently, high-frequency use cases will naturally favor highly specific designs to reduce access costs. Nevertheless, some generic solutions will be available for less-frequent use cases or those requiring a high degree of flexibility. For each ZQE, an on-chain serializer smart contract should be available to generate the serialized Query API call for the `query` function in the Agent Smart Contract.

Once the Query Service generates a result for the query, the Query Prover creates a ZK proof **attesting** that the API call parameters, combined with all relevant block headers already attested by zkFabric, indeed produce the query result. This is where the majority of the work occurs. Different ZQEs will implement their own query stack in a ZK-verifiable manner, utilizing low-level circuit frameworks like Circom [4], Halo2 [12], or gnark [6], or high-level VM/compiler frameworks such as RISC0 [20], zkWASM [31], and zkLLVM [29].

Following proof generation, the Query Prover will submit `query` and `query_result` along with the proof to zkAggregatorRollup by calling the `InsertQueryResults` interface (refer to Section 6 for details about this interface). Upon successful verification, `query` and `query_result` will be stored in zkAggregatorRollup's query result store. At this stage, the zk-attested query result is integrated into the on-chain world. Please refer to Section 6 for details on how applications can access these query results from zkAggregatorRollup's state roots.

5.3. Keeper Network

The Keeper Network is a trust-free component in zkQueryNet that connects ZQEs to the rest of the system. First, it is utilized to relay query events emitted from the Agent Smart Contract to the Query Service and Query Prover. Once the query result is verified and stored in zkAggregatorRollup, it retrieves the query results and their inclusion proof from the rollup and delivers them back to the application through the Agent Smart Contract.

It is crucial to emphasize that the Keeper and other off-chain components in zkQueryNet do not introduce any trust assumptions into Brevis. This is because they are either used to generate ZK proofs that are verifiable on a chain

Archived copy · cryp2nova.com

or simply act as message relayers that cannot alter the content of the message. A query made by a dApp solely relies on the security of the blockchains that the query referred to. With the Keeper Network being permissionless and trust-free, it is possible to implement customized Keepers to obtain query results directly and implement customized callbacks to smart contracts.

6. Brevis zkAggregatorRollup

On top of zkFabric and zkQueryNet, we introduce zkAggregatorRollup, an aggregation layer that verifies and stores the block headers from zkFabric and the query results from zkQueryNet. In other words, it serves as a "rollup" of information from zkFabric and zkQueryNet, and commits (compressed) information to different blockchains that Brevis connects to. Additionally, it acts as a store for attested block headers and query results. We denote Brevis supported and connected blockchains collectively as Base Chains in this white paper.

More specifically, zkAggregatorRollup fulfills the following functionalities:

(1) Verify the validity of the block headers generated from zkFabric and the query results generated from zkQueryNet by checking their validity proofs.
(2) Store the verified block headers and query results in a data structure, which can be later retrieved and verified from any Base Chain the rollup is connected to.
(3) Commit the rollup state root to Base Chains and prove the validity of the root update by generating aggregate proofs for computations performed in steps 1 and 2.

Although it is possible to achieve the above functionalities by making zkAggregatorRollup a Turing-complete virtual machine, we opt to implement the rollup as a lightweight state machine that (1) maintains a storage/state commitment data structure, and (2) recursively verifies ZKP from zkFabric and zkQueryNet.

Moreover, it is worth discussing the necessity of introducing this aggregation layer (instead of verifying, committing, and storing the block headers and the query results directly on each connected blockchain). The primary benefit offered by zkAggregatorRollup is the significant reduction in on-chain verification costs, in the following aspects:

Archived copy · cryp2nova.com

- Without this aggregation layer, each chain would need to store the block headers for every other chain, resulting in $O(N^2)$ "connections" for N blockchains. By introducing zkAggregatorRollup, each blockchain only needs to store the state root for the rollup, reducing the overall connections to $O(N)$.
- The zkAggregatorRollup aggregates multiple proofs for block headers/query results and submits a single proof for verification on each connected blockchain. Consequently, the on-chain verification costs for block headers and query results are efficiently amortized.

In the remainder of this section, we present the system design for zkAggregatorRollup and outline several challenges encountered while constructing the rollup.

6.1. System Overview

6.1.1. System Components

The current architecture of zkAggregatorRollup comprises four system components (see Figure 5):

- **Rollup Sequencer** receives block headers from zkFabric and query results from zkQueryNet, verifies their validity, inserts block headers and query results into storage, and commits them to each Base Chain.
- **Rollup Prover Network** generates validity proofs to demonstrate that the block headers and query results are verified and that the rollup storage is accurately updated.
- **Base Chain Rollup Contract**, deployed on all Base Chains, verifies the validity proofs of each rollup block, and stores the corresponding state roots.
- **Data Availability Layer** provides data availability for zkAggregatorRollup. The DA layer can leverage existing solutions such as Celestia and Eigenlayer.

It is important to note that Brevis zkAggregatorRollup differs from traditional L2 rollup chains in that zkAggregatorRollup commits to multiple Base Chains, making it an "omnichain" rollup. The reason that zkAggregatorRollup can perform this multi-blockchain commitment is that, unlike commonly referred L2 rollups, zkAggregatorRollup does not involve $BaseChain \rightarrow Rollup$ deposits nor $Rollup \rightarrow BaseChain$ withdrawals and is a "rollup-transaction-only" rollup, where the "rollup transactions"

Archived copy · cryp2nova.com

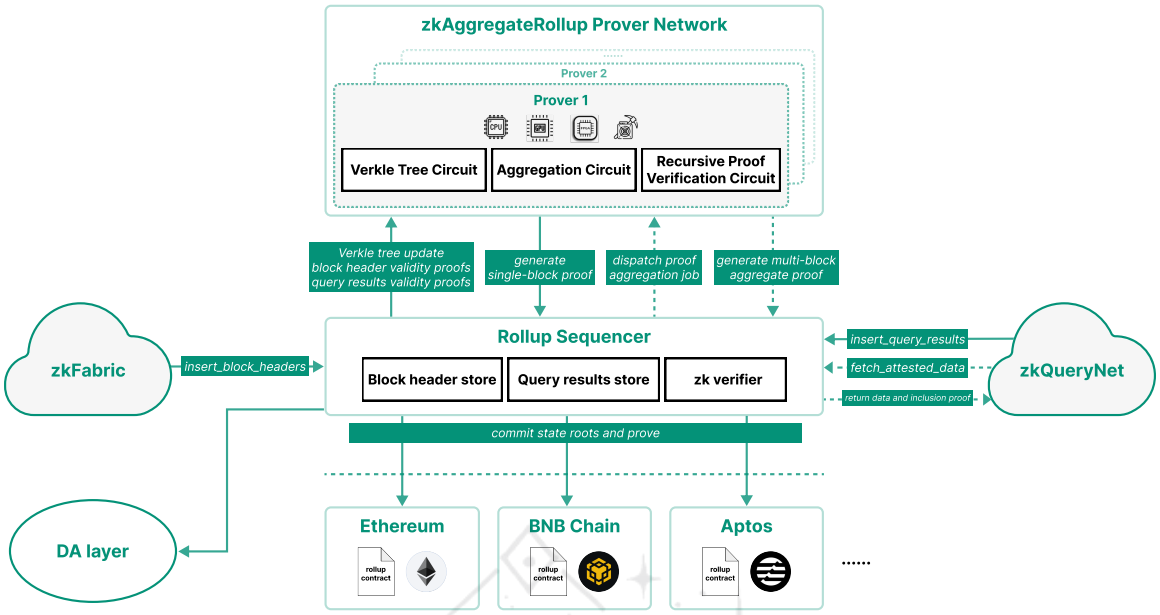


Figure 5. Brevis zkAggregatorRollup Architecture

are state tree updates and recursive proof verification for block headers and query results.

6.1.2. Data Store

As mentioned above, zkAggregatorRollup maintains data stores for attested block headers (called "**block header store**") and query results (called "**query result store**"). In our case, the data is immutable after being attested and inserted, so each data store is an append-only key-value store with a state commitment scheme that allows for efficient data inclusion proof in Base Chain smart contracts. Additionally, it is often the case that a batch of block headers (or query results) needs to be inserted, fetched, or verified, so the data store should also support efficient batch insertions, range queries, and inclusion proofs for a batch of data. Finally, key-value insertion and state update in the data store should be zk-friendly so that they can be proved in a ZK circuit.

While there are many choices for the data store (e.g., Merkle Patricia Tree [14], Sparse Merkle Tree [38], IAVL+[8]), we use the Verkle Tree[27] to store attested block headers and query results. The Verkle tree combines polynomial commitment with the Merkle tree, making wider branching factors more efficient and reducing tree depth. As a result, it significantly reduces the data inclusion proof size, especially when the

Archived copy · cryp2nova.com

tree contains a large number of elements. Moreover, the Verkle tree also supports efficient inclusion proof for a batch of data: by employing random linear combinations of polynomials, a batch of data inclusion can be proved with a single polynomial opening proof.

In zkAggregatorRollup, there are two Verkle trees: $\mathcal{T}_b$ for storage of block headers and $\mathcal{T}_q$ for storage of query results. For the block header Verkle tree $\mathcal{T}_b$, the key is the "(chain ID, block number)" pair, enabling efficient range queries (e.g., retrieve headers of block 1234 to block 1256 of chain ID 1) using the prefix. For the query results Verkle tree $\mathcal{T}_q$, the key is the query ID.

Data Availability. Various ways can be used to provide data availability. For example, we can use any existing DA layer solutions (e.g., Celestia, Eigenlayer) or run a full node for zkAggregatorRollup to access the data. There is no trust assumption about the data availability provider since data validity will be verified via inclusion proof on Base Chains.

For brevity, this paper does not delve into the detailed data structure of Verkle trees nor discuss the data serialization scheme.

6.1.3. Interfaces with Other Modules

zkAggregatorRollup interacts with zkFabric and zkQueryNet in the following ways:

- `InsertBlockHeaders` **(zkFabric → zkAggregatorRollup)**: zkFabric may invoke this interface to submit a batch of block headers along with their proofs for storage in zkAggregatorRollup.
- `InsertQueryResults` **(zkQueryNet → zkAggregatorRollup)**: zkQueryNet may invoke this interface to submit a batch of queries and corresponding query results along with their proofs.
- `FetchAttestedData` **(zkQueryNet → zkAggregatorRollup)**: Via this interface, zkQueryNet can look up block header contents or query results from zkAggregatorRollup, along with their Verkle tree inclusion proof, which can be efficiently verified against the rollup state root directly in the Base Chain Rollup Contract.

Archived copy · cryp2nova.com

6.1.4. Rollup Fee Model

In zkAggregatorRollup, a fee is paid when zkQueryNet sends the `FetchAttestedData` request to the rollup. The paid fee is used to cover the prover cost, $Rollup \rightarrow BaseChain$ commit gas cost, and sequencer cost. To amortize the cost, the $Rollup \rightarrow BaseChain$ zk commit only occurs once the sequencer has accumulated enough `FetchAttestedData` requests. Naturally, there will be a timeout to ensure that the batch waiting time is upper-bounded. In the future, there will be finer-grained fee models where each user may select the maximum delay and the maximum fee they are willing to pay.

6.1.5. Basic Workflow

The overall workflow of zkAggregatorRollup is as follows:

(1) The sequencer receives a `InsertBlockHeaders` or `InsertQueryResults` transaction.
(2) The sequencer verifies if the submitted block headers or query results are valid by checking the enclosed validity proof.
(3) If valid, insert the block headers or query results into the Verkle tree.
(4) The sequencer sends the Verkle tree updates and the block headers/query results to a rollup prover to generate proof attesting that the Verkle tree is correctly updated and the validity of the block headers/query results has been verified.
(5) The rollup prover sends back the proof to the sequencer.
(6) As more `InsertBlockHeaders` or `InsertQueryResults` transactions are submitted, the sequencer accumulates more proofs.
(7) The sequencer dispatches the accumulated proofs to a rollup prover for aggregation as soon as the sequencer receives enough `FetchAttestedData` transactions from zkQueryNet.
(8) The rollup prover aggregates the proofs and sends back the aggregate proof to the sequencer.
(9) The sequencer commits the accumulated Verkle tree root updates to the Base Chain contract and submits the aggregate proof to prove the validity of these updates.

In the following two subsections, we describe the detailed protocols for the rollup

Archived copy · cryp2nova.com

sequencer and the rollup prover network.

6.2. Rollup Sequencer

In zkAggregatorRollup, the sequencer verifies the validity proof for the block headers and query results and maintains Verkle trees for storage of attested block headers and query results. Initially, the sequencer is a centralized server and will become a decentralized sequencer network in the future.

Whenever a `InsertBlockHeaders` or `InsertQueryResults` transaction is processed by the sequencer, the Verkle tree is updated and a new **rollup block** is generated. A rollup block i is **finalized** on Base Chain if the Verkle tree root at block number i has been committed to Base Chain rollup contract and the corresponding validity proof has been verified in the Base Chain rollup contract. Note that a `FetchAttestedData` transaction won't update the Verkle tree state so a new rollup block is not generated (but the sequencer still needs to take some actions, to be introduced later).

Below we describe in detail how the sequencer handles these transactions.

6.2.1. Insert Block Headers

Let's assume the latest rollup block number is i. The Verkle tree at block number i is represented by $\mathcal{T}_i^b$ with its root denoted by root_i^b.

Suppose that the sequencer receives a `InsertBlockHeaders` transaction from a zkFabric relayer. The following information should be submitted with the transaction:

- A list of block headers to be stored: $\mathcal{B} = \{\mathsf{B}_n^c\}_{c,n}$ each element B_n^c represents the block header of chain c at block number n.
- Aggregate proof $\pi(\mathcal{B})$ that attests the correctness of the block headers.
- Hash of the block headers $H(\mathcal{B})$. The hash along with the aggregate proof $\pi(\mathcal{B})$ will be later submitted to the rollup prover as part of the public inputs to the recursive verification circuit; otherwise, the recursive verification circuit will need to take the full list of block headers $\mathcal{B}$, which increases the verification circuit size.

Then the workflow for the sequencer to handle the `InsertBlockHeaders` transaction is as follows:

(1) The sequencer verifies the validity proof $\pi(\mathcal{B})$ for the submitted block headers

Archived copy · cryp2nova.com

$\mathcal{B}$.

(2) If valid, it generates a new rollup block $i+1$ and updates the Verkle tree by inserting the verified block headers, i.e., inserts key-value pair $(c, n) \mapsto H(B_n^c)$ for each block $B_n^c \in \mathcal{B}$. Note that only the hash values of the block headers are stored in the Verkle tree leaves; the full block header contents are stored in a separate KV DB and be accessed via $H(B_n^c) \mapsto B_n^c$.

(3) The sequencer sends the Verkle tree update $\mathcal{T}_i^b \to \mathcal{T}_{i+1}^b$ along with the block headers $\mathcal{B}$, their hash value $H(\mathcal{B})$ and the validity proof $\pi(\mathcal{B})$ to one of the rollup provers to generate a validity proof $\pi_{i+1} = \pi\big(\pi(\mathcal{B}), \mathcal{T}_i^b \to \mathcal{T}_{i+1}^b\big)$ attesting the fact that the validity proof $\pi(\mathcal{B})$ was correctly verified and that the Verkle tree update $\mathcal{T}_i^b \to \mathcal{T}_{i+1}^b$ was correct. The proof π_{i+1} is then sent back from the rollup prover to the sequencer. Note that to speed up the proof generation process, the proofs for different rollup blocks can be generated in parallel on different rollup provers.

The new Verkle tree root_{i+1}^b won't be immediately committed to Base Chain. Instead, the *Rollup* $\to$ *BaseChain* commit will be triggered by `FetchAttestedData` transaction from zkQueryNet, which will be specified in Section 6.2.3.

Protocol 1 formalizes the above flow for inserting new block headers.

Protocol 1 Sequencer: `InsertBlockHeaders`

Input: A list of block headers $\mathcal{B} = \{\mathsf{B}_n^c\}_{c,n}$
Input: Aggregate proof $\pi(\mathcal{B})$ that attests the correctness of the block headers
Input: Hash of the block headers $H(\mathcal{B})$
Input: The latest rollup block number i

1: **if** VERIFY($\pi(\mathcal{B})$)=True **then**
2: Increment the rollup number to $i+1$
3: **for** each block $B_n^c \in \mathcal{B}$ **do**
4: Insert key-value pair $(c, n) \mapsto H(B_n^c)$ into Verkle tree
5: Insert key-value pair $H(B_n^c) \mapsto B_n^c$ to KV DB
6: **end for**
7: Send Verkle tree update $\mathcal{T}_i^b \to \mathcal{T}_{i+1}^b$, block headers $\mathcal{B}$, hash value $H(\mathcal{B})$ and proof $\pi(\mathcal{B})$ to a rollup prover
8: **end if**

6.2.2. Insert Query Results

The flow for handling the `InsertQueryResults` transaction is described in Protocol 2, which is similar to `InsertBlockHeaders`.

Archived copy · cryp2nova.com

Protocol 2 Sequencer: `InsertQueryResults`

Input: A list of queries and corresponding query results $\mathcal{QR} = \{(Q_i, R_i)\}_i$ where element represents the i-th query Q_i and corresponding result R_i. Each query Q_i is associated with a unique query ID (denoted by $Q_i \cdot id$).
Input: Hash of queries and results $H(\mathcal{QR})$
Input: Aggregate proof $\pi(\mathcal{QR})$ that attests the correctness of the query results
Input: The latest rollup block number i

1: **if** VERIFY($\pi(\mathcal{QR})$)=True **then**
2: Increment the rollup number to $i + 1$
3: **for** each block $(Q_i, R_i) \in \mathcal{QR}$ **do**
4: Insert key-value pair $Q_i \cdot id \mapsto H(Q_i||R_i)$ into Verkle tree
5: Insert key-value pair $H(Q_i||R_i) \mapsto (Q_i, R_i)$ to KV DB
6: **end for**
7: Send Verkle tree update $\mathcal{T}_i^b \to \mathcal{T}_{i+1}^b$, queries/results $\mathcal{QR}$, hash value $H(\mathcal{QR})$ and proof $\pi(\mathcal{Q})$ to a rollup prover
8: **end if**

6.2.3. Fetch Attested Data

As previously mentioned, when zkQueryNet sends a `FetchAttestedData` request to the sequencer, not only the data and its inclusion proof is returned, but also the *Rollup* $\to$ *BaseChain* state root commit and finalization may be triggered.

Specifically, assume that the last finalized rollup block number on Base Chain is n_f, and the latest rollup block number is n_r, which means that rollup blocks $n_f + 1, ..., n_r$ haven't been committed and finalized on Base Chain. The corresponding rollup proofs are $\pi_{n_f+1}, ..., \pi_{n_r}$ and Verkle tree roots are $\mathsf{root}_{n_f+1}, ..., \mathsf{root}_{n_r}$.

To amortize the cost, the sequencer will commit and attest these Verkle tree roots in the Base Chain rollup contract once the number of accumulated `FetchAttestedData` requests exceeds M (of course, there will be a timeout to cap the max waiting time for each batch of requests). Suppose the current accumulated number of `FetchAttestedData` requests is m (batch counter).

The flow for a `FetchAttestedData` transaction is as follows (see Protocol 3)

(1) zkQueryNet submits a `FetchAttestedData` transaction to the sequencer with some *key*. For example, *key* could be chain ID+block number for block header retrieval or query ID for data query results.
(2) The sequencer look up the value via $key \mapsto hash(value) \mapsto value$ from the Verkle tree and the KV DB, and generates inclusion proof for the fetched data. The sequencer then sends the data value and the inclusion proof to zkQueryNet for verification on Base Chain.

Archived copy · cryp2nova.com

(3) The sequencer increments the batch counter $m \to m + 1$.

(4) If the batch counter $m \geq M$, the sequencer sends proofs $\pi_{n_{f+1}}, ..., \pi_{n_r}$ to a rollup prover for aggregation.

(5) After receiving the aggregate proof $\hat{\pi}_{n_{f+1} \to n_r}$, the sequencer commits the Verkle tree roots $\mathsf{root}_{n_f+1}, ..., \mathsf{root}_{n_r}$ to the Base Chain rollup contract and submit the attestation proof $\hat{\pi}_{n_{f+1} \to n_r}$ to finalize these state commits.

(6) The sequencer reset the batch counter $m = 0$.

Protocol 3 Sequencer: `FetchAttestedData`

Input: *key* (*key* is chain ID + block number for block header retrieval or query ID for data query results)

1: Look up the value for the key from the Verkle tree and the KV DB.
2: Generate inclusion proof for the key-value pair
3: Send the key-value pair and the inclusion proof to zkQueryNet
4: Increment the batch counter $m \to m + 1$
5: **if** $m \geq M$ **then**
6: Send accumulated rollup block proofs $\pi_{n_{f+1}}, ..., \pi_{n_r}$ to a rollup prover for aggregation.
7: After receiving the aggregate proof $\hat{\pi}_{n_{f+1} \to n_r}$, commit the Verkle tree roots $\mathsf{root}_{n_f+1}, ..., \mathsf{root}_{n_r}$ to the Base Chain rollup contract and submit the attestation proof $\hat{\pi}_{n_{f+1} \to n_r}$ to finalize these state commits.
8: Reset batch counter $m = 0$
9: **end if**

6.3. Rollup Prover Network

The rollup prover network is responsible for generating proofs for rollup blocks. Provers in the network are expected to utilize accelerators such as GPUs, FPGAs, and ASICs to reduce the proving time and proving cost.

The sequencer may dispatch a single-rollup-block proof job or a multi-rollup-block aggregation-proof job to a rollup prover. Multiple rollup provers may generate proofs in parallel. Figure 6 illustrates how the aggregate proof is generated by rollup provers.

6.3.1. Single Rollup Block Proof

Each single rollup block proof involves two circuits:

(1) A **Verkle tree circuit** that proves the correctness of Verkle tree update (in particular, the correctness of tree update after inserting a key-value pair).

(2) A **recursive verification circuit** that proves that the proof $\pi(\mathcal{B})$ of block

Archived copy · cryp2nova.com

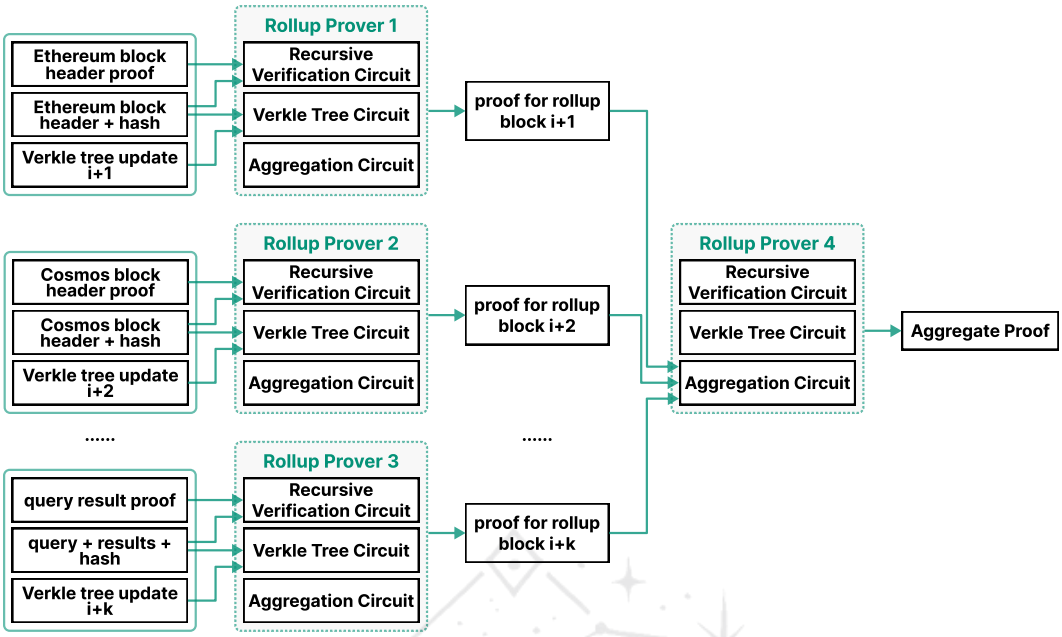


Figure 6. Proof generation by rollup provers.

headers or the query results proof $\pi(\mathcal{QR})$ has been correctly verified.

After receiving the rollup block proof assignment from the sequencer, the rollup prover first generates two proofs using the two circuits. Then it aggregates the two proofs into one proof and sends it back to the sequencer.

In terms of ZKP design, the recursive verification circuit is of particular interest, where we embrace a **composable architecture** and don't restrict ourselves to one particular ZKP development framework.

- The "inner" proof $\pi(\mathcal{B})$ or $\pi(\mathcal{QR})$ may be generated in heterogeneous systems (e.g., Plonk, Groth16, STARKs) for performance reasons. For example, our benchmark results [32] show that SHA-256 proof generation is faster in Groth16 than in Plonk due to its special binary wire values, while other operations such as elliptic curve pairing and non-native field arithmetics may be faster in Plonkish arithmetization than R1CS due to its flexibility. Thus, different blockchain consensus mechanisms may be best proved in different proving systems, not to mention the more heterogeneous query patterns. As a result, the recursive verification circuit should be able to verify different proving systems in the circuit. In fact, such a composable architecture has already been adopted in production ZK systems such as Polygon zkEVM [19] which compose STARKs with Groth16.

Archived copy · cryp2nova.com

- Moreover, Brevis zkAggregatorRollup commits to multiple Base Chains where heterogeneous cryptographic primitives are supported. For example, while Ethereum only supports BN254 pairing as a pre-compile, other blockchains support efficient pairing over other elliptic curves as pre-compiles or built-ins. This opens up the space for more efficient recursion such as the 2-chain scheme [37].
- Finally, we are also researching and developing efficient "outer" layer protocols that enable efficient and interoperable instantiations (e.g., LegoSNARKs [36], sumcheck [35]).

6.3.2. Multi-Rollup-Block Aggregation Proof

In addition, the rollup prover may also be assigned a job for aggregating the multiple rollup block proofs $\pi_i, ..., \pi_{i+k}$. Then a single aggregate proof is sent back to the sequencer and will be later submitted to Base Chain for on-chain verification.

Our current implementation is using the batching technique [18] and will evolve to more sophisticated aggregation techniques such as aPlonk [34] and Nova [39]

The assignment of proving jobs within the rollup prover network is of independent interest and won't be discussed in this paper. We will publish separate papers on the economy design for the rollup prover network.

7. Use Cases

Brevis empowers developers to create dApps that can access omnichain data across arbitrary time horizons in a way that was previously impossible. This innovation will undoubtedly establish a new paradigm, transforming how dApps are developed across all sectors. In this section, we present several immediate applications, but we anticipate the community will discover many more use cases.

7.1. ZK Multi-blockchain Interoperability

As discussed in Section 4, nearly all existing interoperability solutions fall into the "external validation model" which necessitates trust in an intermediary entity. Although on-chain light client solutions are trust-minimized in theory, they are prohibitively expensive to implement in practice. Brevis addresses this challenge by combining light client protocol and ZKP, where zkFabric generates ZK Proof of Consensus for all con-

Archived copy · cryp2nova.com

nected blockchains and stores the corresponding block headers in zkAggregatorRollup.

With these zk-attested block headers available, constructing a zkBridge on top of Brevis is as straightforward as implementing a ZQE with the following Query API:

```
byte[] get_message(uint256[] chain_ids,
                   uint256[][] message_ids);
```

This Query API enables batched delivery of multiple messages from different blockchains. An off-chain relayer can drive this Query API by monitoring source chain messages and assisting the destination chain receiver smart contract in calling the Agent Smart Contract with the appropriate `message_ids` and `chain_ids`. Existing multi-chain dApps, including cross-chain DeFi and asset bridges, can easily adapt to this ZK interoperability solution without altering their application-layer smart contracts.

7.2. Data-driven DeFi

In addition to the liquidity farming reward auto-adjustment example discussed in Section 3, we believe data-driven DeFi will become a broad class of applications leveraging Brevis, thanks to the scalability and privacy it enables.

With trust-free access to comprehensive historical and omnichain trading flows, derivatives such as options can now incorporate novel exercise conditions, such as Exponential Time Weighted Price or Time-Volume Weighted Price. Alternative derivatives tracking user behavior, price trends, protocol or blockchain TVL changes, volatility, price correlations, and more will also be possible through various implementations of ZQEs.

On-chain active fund management solutions can generate ZK proofs demonstrating that position adjustments are based solely on specific algorithmic models derived from market data, without any unauthorized human intervention. Furthermore, due to the privacy-preserving property of Brevis, the precise model parameters can remain concealed to maintain a competitive market advantage.

7.3. User Acquisition with Trust-free Revenue Sharing

The lifecycle of user acquisition in Web 2.0 typically concludes at the point of user download or registration. At this stage, advertisers pay advertising channels, and their

Archived copy · cryp2nova.com

relationship terminates. This arrangement is suboptimal for both parties: advertisers pay the same upfront price for all acquired users, without the ability to differentiate based on their Lifetime Value; advertising channels also cannot share long-term revenue streams, even when acquiring high-value users. This lose-lose situation arises because, in the Web 2.0 world, post-acquisition user data falls within the advertiser's private domain, making it impossible for advertising channels to access post-acquisition user statistics in a **trust-free way**.

Web 3.0 has the potential to fundamentally transform this user acquisition model since most user activity data are in the public domain. However, current Web 3.0 user acquisition platforms, such as Quest3, still follow the Web 2.0 paradigm because there is no way to obtain post-acquisition user revenue data directly on-chain and in a trust-free manner. Brevis revolutionizes this by enabling advertising channels to generate revenue proofs for all users they acquire. By leveraging Brevis, the user acquisition landscape in Web 3.0 will undergo a fundamental shift: advertisers will pay only for high-value users, and advertising channels will be incentivized to match users with individual advertisers optimally.

7.4. zkDID

With the capability to prove any statistics about users' historical behaviors, Brevis will be an essential tool for constructing trust-free zkDID solutions that can be employed in various use cases.

7.4.1. Prevent Sybil Attacks

Since creating an account on the blockchain incurs practically zero cost, the issue of Sybil Attacks has plagued the blockchain community from day one. When incentivized user acquisition events or retrospective airdrops occur, it is often challenging for blockchain projects to differentiate bots from real users. Off-chain solutions frequently entail regulatory and trust issues by creating a centralized token distribution entity.

By using Brevis, applications can send queries to different ZQEs to generate ZK proofs of individual or aggregated users' identities based on hard-to-fake on-chain behaviors, such as account age, interaction with KYC'ed exchanges, fees paid to relevant protocols or dApps, long-term holding of meaningful assets, and more. Using

Archived copy · cryp2nova.com

anti-Sybil solutions built with Brevis, dApp developers can prevent Sybil Attacks in airdrop campaigns or beta rollouts to exclusive early contributors while operating in a fully trust-free manner.

7.4.2. User Life Cycle Management

While accessing external data for a dApp can help prevent Sybil Attacks, a dApp's internal data can be crucial for building a trust-free user life cycle management process. For example, a loyalty system can be constructed for new users who passed the initial Sybil Attack firewall based on their day-to-day usage and behavior within the application.

Consider a DEX, for instance; one can build a VIP trader program to reduce fees for professional traders based on volume. A qualifying trader can send a query via Brevis to generate corresponding zkDID proof, demonstrating that she has traded a volume exceeding a VIP tier threshold during an accounting period. In another example, blockchain gaming LiveOps campaigns can be designed based on arbitrary user behaviors such as 7-day retention, gameplay frequency, session time, winning ratio, and various other in-game metrics.

One might argue that such internal data accounting and analysis logic can be built into the dApp itself without relying on Brevis. While this is true, solutions built with Brevis offer several key advantages:

- **Low-cost**: Incorporating in-app accounting incurs additional on-chain costs for every transaction and all users. This cost is significantly higher compared to the as-needed proof generation for only a relatively small number of users matching specific criteria. It is important to note that proof generation and on-chain verification can actually be batched across many users to further amortize the costs.
- **Future-proof**: Smart contracts are generally immutable unless altered through governance processes. However, user life cycle management often requires diverse campaigns targeting different users based on current circumstances. Adding data accounting directly within a dApp will not be able to address dynamically changing requirements. However, Brevis-powered zkDID solutions can flexibly change user targeting of various campaigns without modifying any on-chain smart contract, making them much more future-proof.

Archived copy · cryp2nova.com

- **Portable**: Tracking statistics with in-dApp accounting means that data are still isolated within walled gardens, and there is no simple way for users to reuse these already proven identities. However, zkDIDs generated via Brevis are publicly stored in zkAggregatorRollup and are therefore portable to different applications.

7.5. Account Abstraction

Another compelling use case for Brevis lies in the realm of Account Abstraction (AA). The principle of account abstraction involves using a smart contract as an "abstraction" layer for users to access blockchain applications. Compared to employing a bare-metal crypto wallet, AA often provides enhanced security and usability.

One frequently discussed usability benefit of AA is social recovery, wherein access to the wallet can be regained even when the main controlling key is lost. Current social recovery designs rely on a static set of predefined multi-signatures. However, this set of multiple signatures must be carefully maintained and updated to ensure recoverability in the event of a lost key. By using Brevis, a smart wallet can implement social recovery capability based on recent social and financial transaction connections.

Instead of relying on a fixed set of external wallets, the social recovery function can query Brevis to trust-freely obtain the top five financial counterparties with the highest outgoing transaction value during a specific period or the top three KYC'd exchange accounts to which the account deposited funds in the last year, and use those as sources for social recovery. With thoughtful design, this approach will significantly reduce maintenance overhead and further lower the barrier of entry for new users in the blockchain ecosystem.

Numerous other use cases exist in all the aforementioned directions, and the applicable scenarios in other areas such as social and NFT gaming are equally thrilling. We encourage community developers to explore and discover more exciting use cases.

8. Conclusion

In this white paper, we propose Brevis, a ZK omnichain data attestation platform that empowers dApps to access, compute, and utilize arbitrary data across the entire history of multiple blockchains by leveraging Zero-Knowledge (ZK) succinct proofs. Brevis's

Archived copy · cryp2nova.com

architecture consists of three main components: zkFabric, zkQueryNet, and zkAggregatorRollup. Together, these components provide a trust-free, omnichain, modularized, and low-cost solution for data access and computation. The economic incentive design of Brevis is not covered in this white paper and will be discussed in future works.

Archived copy · cryp2nova.com

References

[1] *Axelar*, https://axelar.network.

[2] *Celer cbridge*, https://celer.network.

[3] *Chainlink*, https://chain.link.

[4] *Circom*, https://docs.circom.io.

[5] *Clique proof-of-authority consensus protocol*, https://eips.ethereum.org/EIPS/eip-225.

[6] *Consensys gnark*, https://docs.gnark.consensys.net.

[7] *Consensys zkevm*, https://consensys.net/zkevm.

[8] *Cosmos iavl+ tree*, https://github.com/cosmos/iavl.

[9] *Cosmos ibc*, https://ibc.cosmos.network.

[10] *Dune analytics*, https://dune.com/browse/dashboards.

[11] *The graph*, https://thegraph.com.

[12] *Halo2*, https://zcash.github.io/halo2.

[13] *Ics-23: Vector commitments*, https://github.com/cosmos/ics23.

[14] *Merkle patricia trie*, https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie.

[15] *Multichain*, https://multichain.xyz.

[16] *Near rainbow bridge*, https://near.org/bridge.

[17] *Nomad bridge*, https://app.nomad.xyz.

[18] *Plonk aggregation circuit*, https://github.com/matter-labs/recursive$_a$ggregation$_c$ircuit.

[19] *Polygon zkevm*, https://docs.hermez.io/zkEVM/zkProver/Overview/zkProver-Overview.

[20] *Risc zero*, https://www.risczero.com.

[21] *Ronin bridge*, https://bridge.roninchain.com.

[22] *Scroll zkevm*.

[23] *Starkware*, https://starkware.co.

[24] *Succinct labs proof of consensus*, https://blog.succinct.xyz/post/2022/10/29/gnosis-bridge.

[25] *Tendermint light client in solidity*, https://github.com/ChorusOne/tendermint-sol.

[26] *Thorchain*, https://thorchain.org.

[27] *Verkle tree*, https://vitalik.ca/general/2021/06/18/verkle.html.

[28] *Wormhole*, https://wormhole.com.

[29] *zkllvm*, https://github.com/NilFoundation/zkllvm.

[30] *zksync*, https://zksync.io.

[31] *zkwasm*, https://github.com/DelphinusLab/zkWasm.

[32] *Sha256 proof benchmarks: https://blog.celer.network/2023/03/01/the-pantheon-of-zero-*

Archived copy · cryp2nova.com

knowledge-proof-development-frameworks (2023).

[33] R. Akeela and W. Chen, *Yafa-108/146: Implementing ed25519-embedding cocks-pinch curves in arkworks-rs*, Cryptology ePrint Archive (2022).

[34] M. Ambrona, M. Beunardeau, A.L. Schmitt, and R.R. Toledo, *aplonk: Aggregated plonk from multi-polynomial commitment schemes*, Cryptology ePrint Archive (2022).

[35] J. Bootle, A. Chiesa, and K. Sotiraki, *Sumcheck arguments and their applications*, in *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I 41*. Springer, 2021, pp. 742–773.

[36] M. Campanelli, D. Fiore, and A. Querol, *Legosnark: Modular design and composition of succinct zero-knowledge proofs*, in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 2075–2092.

[37] Y. El Housni and A. Guillevic, *Families of SNARK-friendly 2-chains of elliptic curves*, in *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part II*. Springer, 2022, pp. 367–396.

[38] F. Haider, *Compact sparse merkle trees*, Cryptology ePrint Archive (2018).

[39] A. Kothapalli, S. Setty, and I. Tzialla, *Nova: Recursive zero-knowledge arguments from folding schemes*, in *Advances in Cryptology–CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV*. Springer, 2022, pp. 359–388.

CRYP2NOVA

Archived copy · cryp2nova.com