Burrata: Bridging Web$_2$ to Web$_3$

The Burrata Team

Abstract

Smart contracts are a basic building block of Web$_3$, enabling users to transact on data in a decentralized manner. Smart contracts, however, are limited by the blockchains they run on and cannot access external state.

Burrata enables smart contracts to operate on data from familiar Web$_2$ services by relying on clients to bring this data to Web$_3$ with trust. Burrata focuses on bridging this data with efficiency and privacy across multiple chains: a typical Burrata data transfer on Ethereum costs about 20,000 gas ($^1/_3$ the cost of an ERC-20 transfer), occurs in a single transaction and supports selective disclosure of personal data.

1 Introduction

Web$_3$ is a promising new platform for supporting an ecosystem of decentralized applications based on blockchains. The basic building block of these applications are smart contracts, which enable users to transact on data without a central gatekeeper: public-key cryptography is used to prove ownership of accounts, and permissionless consensus engines execute smart contract code which define operations on data. Smart contracts are used to build tokens and NFTs, decentralized exchanges (DEXes) such as OpenSea [28] and Uniswap [3], DeFi applications such as 1inch [1] and Aave [2] — and have the potential for many more applications.

A limitation of Web$_3$ is that smart contracts cannot directly access external state from familiar Web$_2$ services: the consensus engines which run blockchains rely on state machine replication (SMR) [22], which requires that each replica observe the same state transitions. External state, however, may vary when accessed by each replica. For example, fetching a web page may yield a different result depending on the replica's geographical location or when the replica reads the data. As a result, most blockchains limit the data smart contracts can access to data internal to the blockchain, greatly limiting the applications which can be built in Web$_3$. For instance, a smart contract cannot directly check if a user is from a particular country, if they have sufficient funds in their bank account, or even the current weather.

To address this gap, the Web$_3$ community relies on oracles to bring external data into Web$_3$ [5, 7]. Oracles bring data into Web$_3$ by listening to events on blockchains, servicing the event on oracle servers in Web$_2$, and then posting the result back on the blockchain. While oracles form the foundation for many Web$_3$ services today, such as DEXes and cross-chain bridges, oracles have a number of idiosyncrasies which limit their uses. For instance, a typical oracle exchange follows an asynchoronous request-reply pattern, which requires two transactions and delay between request and completion. These actions are published on the blockchain, which lack privacy and opens oracles to front-running attacks. In addition, the data is collected by oracle nodes, which cannot access personal user data.

Burrata takes an alternative, client-driven approach which maintains user privacy while enabling smart contracts to efficiently operate on Web$_2$ data. Instead of relying on third-party oracles, users drive Burrata to collect data from data hubs, which are then verified by lightweight contracts on the blockchain. The data verification is achieved with a single signature check, with a total cost of about 20,000 gas on Ethereum ($^1/_3$ the cost of an ERC-20 transfer). Burrata is fully flexible on the types of data which can be delivered, which can be structured, selectively disclosed, and even as complex as a zero knowledge proof. Since Burrata is client-driven, the contract can access personal data, such as data from websites the user has logged in to. Burrata delivers Web$_2$ data to Web$_3$ smart contracts in a single transaction, which simplifies the programming model by eliminating the need to design an asynchoronous upcall flow, and avoids the need to design aggregation protocols to prevent front-running. Burrata is scalable and deployed on multiple chains.[1]

The remainder of this paper is organized as follows:

- We provide background on existing techniques to deliver data to Web$_3$ and distinguish them from Burrata.

[1]Today, we are deployed on 8 EVM based chains and the NEAR protocol.

- We describe the Burrata architecture and show how the Burrata frontend works with data hubs and contracts to deliver data to Web_3.
- We show an example of how Burrata can be integrated with a smart contract and a corresponding decentralized application (dApp).
- We evaluate how Burrata compares to oracle and token based systems.
- We discuss future work.

2 Background

Web_3 is a platform which supports decentralized applications that are built with smart contracts running on blockchains. While anyone can inspect the data in these blockchains, the smart contract cannot access any data outside the blockchain. This is distinguished from traditional Web_2 services, which provide *personal data*, which is private and only accessible by authorized users.

Web_2 data, however, is critical for many emerging Web_3 applications. A DeFi application, for example, may wish to verify a user has a minimum bank account balance or credit score. A decentralized autonomous organization (DAO) [19] might want to verify that no human is allowed to have multiple votes (sybil resistance [4]). A rental application might want to check that a human has signed a legally binding document. These use cases all require a way to bridge Web_2 data into Web_3.

2.1 The Web_3 Glass Data Prison

Unlike traditional Web_2 applications, Web_3 smart contracts cannot simply make a request to an external service to fetch data. To understand why, we will briefly describe how a user interacts with a typical blockchain.

When a user wants to interact with a smart contract, the user signs a transaction which contains the address of the contract the user wishes to interact with. This transaction is submitted to the blockchain to a pool of transactions known as the mempool. The blockchain is made resilient to failures and malicious actors through a technique known as state machine replication (SMR) [22]. SMR divides the blockchain into replicas. At some interval, the replicas agree on a set of transactions from the mempool and aseemble these transactions into a block. Each replica executes the transactions in the block in the same order. If the execution of each transaction is deterministic and each replica is correct and not malfunctioning, then all the replicas will arrive at the same new state. In the case of the users' transaction, each replica will load the code of the smart contract, run it, and save the effects of running the transaction to its local state.

While SMR is a powerful construct to build blockchains upon, it also is limiting: if the transaction is not deterministic, then all the replicas will not arrive at the same new state. Accessing data from external sources is often not deterministic: for instance, two replicas might observe different results based on the time they read the data. In order to ensure that executing a smart contract is deterministic, blockchains limit the state contracts can access to internal state.

In effect, each blockchain is a glass data prison: while the internal state of the blockchain is visible to everyone, once transactions enter the blockchain, they cannot interact with the outside world.

2.2 Oracles to the rescue?

To build meaningful Web_3 applications, smart contracts must be able to consume external data. Oracles have become the basic primitive to bring data into blockchains. Oracle networks sit between Web_3 and Web_2: they listen for events on a blockchain, retrieve and form consensus on Web_2 data, and post the results back to the blockchain.

Oracles are used extensively in Web_3, and used by a wide variety of applications. Chainlink [8], for example, provides a programmable oracle network and provides price feeds, among other data, to projects such as Aave [2] and compound [14]. Wormhole [27] is another project which provides a cross-chain message passing protocol which uses a network of oracles called guardians to pass verified messages between chains. Pyth [26] is yet another project with a slight variation on the traditional oracle model by enabling users to stream and publish Web_2 data on a blockchain by calling a contract.

A major downside to oracles is that they are not private. Requests for data are published on blockchains, which opens the door to front-running attacks and requires the use of mitigations such as price aggregation. The data itself is also published, which may not be a desirable property in many applications. In addition, oracles typically cannot access individual private user data, such as websites which require a login.

Oracles are also asynchoronous, which increases programming complexity and can make user transactions unpredictable. For instance, a user which submits a transaction to a smart contract which uses an oracle will have to wait for the oracle network to post the response via an upcall. By the time the oracle issues the upcall, the data could change, resulting in the transaction failing. In addition, the user must pay, directly or indirectly, for the cost of both transactions.

2.3 Data Tokenization

Data tokenization is another alternative to oracles used for personal data, used by projects such as polymath [25] and embodied in proposals such as the soulbound token [10, 35, 39]. In the tokenization model, users are issued a token which is non-transferrable. This token can contain data from Web_2, or the issuance of the token itself can be dependent on some condition in Web_2 (for instance, if the user has an account on a specific service). Smart contracts which wish to consume this data can then check for the presence of this token and read data from the token when a user interacts with the contract.

While tokens enable users to bring personal data into Web_3, users must mint a token, which can increase friction in using a Web_3 service. The data in a token can become stale, and a user will incur transaction fees in order to refresh and update the token. If the data in the token becomes invalid, a Web_2 service will have to monitor the data and submit a transaction to revoke the token. While the token revocation transaction is pending, a contract may work on data which is no longer valid. Finally, this data is published on the blockchain, and contracts the user does not interact with may use data from their token.

2.4 Verifiable User Data

Systems such as DECO [38] and TLSNotary [32] enable users to access Web_2 data via TLS and SSL with third-party observers which can only view the encrypted contents of the session. The user can then generate a zero knowledge proof which third-party observers can verify using the transcript of the encrypted session.

While DECO and TLSNotary enable third parties to selectively verify data was accessed, they do not deliver data to blockchains on their own. Furthermore, users must establish a TLS session using a special library, which may be incompatible with traditional Web_2 tooling. Establishing the 3-way TLS session may especially be a challenge if started from the client, due to idiosyncrasies in browsers.

2.5 Introducing Burrata

In this paper, we introduce Burrata, which takes another approach to bringing Web_2 data into Web_3. With Burrata, users deliver trusted Web_2 data with the transactions they submit instead of waiting for an oracle to asynchronously retrieve Web_2 data. Unlike tokenization, data does not become part of the blockchain's world state, and minting data is not required. To provide a fully trustless environment, Burrata leverages systems such as DECO and TLSNotary to enable the verification of data retrieved from Web_2.

3 Burrata Architecture

The Burrata architecture consists of three components:

Frontend The frontend manages interactions with users. Decentralized application (dApp) developers call on the frontend from the users' browser to initiate collection of Web_2 data that is submitted to the developers' smart contracts.

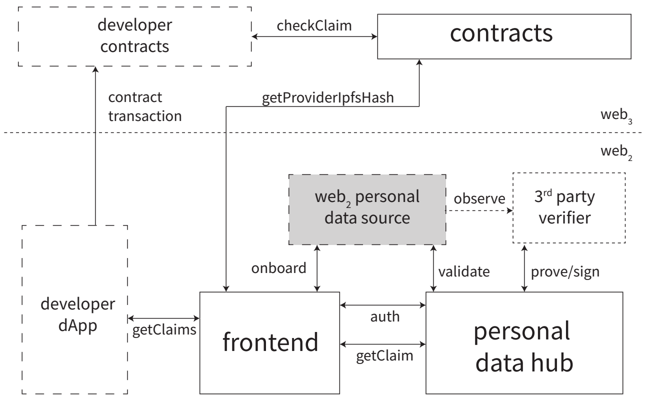

Figure 1: The high-level architecture of Burrata, showing the interaction between the frontend, data hub and contracts. Burrata components are solid boxes, developer-provided components are in dashed boxes, the Web_2 data source is in a shaded box, and 3^{rd} party components are in a dotted box. A dotted arrow indicates an encrypted communcation for which the destination does not have the decryption key.

Data Hub The data hub authenticates users and accesses Web_2 data on their behalf to issue Burrata claims. The claims that the data hub are verifiable and have a limited lifetime. The data hub tracks claims that have been issued, manages refresh and revocation, and supports conversion between different claim formats.

Contracts The Burrata contracts are called by developer's smart contracts to check the validity of claims. The contracts manage which data hubs are valid, and support the emergency revocation of claims.

The overall architecture of Burrata is shown in Figure 1. While Burrata runs on multiple blockchains, such as the Ethereum Virtual Machine (EVM) [36] and Near [30] with support for Move-based chains such as Sui [31] in the near future, for the purposes of this paper, we focus on the EVM implementation.

To guide the description of Burrata, we first describe an example dApp called Web_3Cabin.

3.1 An example dApp

Web_3Cabin is a simple application which allows hosts to rent a cabin in the woods to renters who rent the cabin for a short-term stay. The rental is automated, and the host and guest have recourse in the real world if either party fails to fulfill their obligations. Both host and renter remain anonymous to each other unless their obligations fail to be fulfilled.[2]

[2]The Web_3Cabin code is open source and can be found on Github.

Hosts When a host creates a rental, they create a contract that governs the terms of the rental and place a Web$_2$ connected lockbox in front of the cabin.

Renters To rent a cabin, a renter deposits payment and signs the contract. When the renter arrives, the lockbox releases the keys to the renter, and when the renter checks out, they return the key to the lockbox.

The smart contract can programatically exchange assets on the blockchain. For instance, it can generate a token if the cabin is available and the renter deposits payment. However, without external help, the contract does not know about the state of the external world, such as whether the rental contract was signed, or if the key was returned to the lockbox.

3.2 Burrata Claims

To facilitate the transfer of data between Web$_2$ and Web$_3$, users collect Burrata claims from data hubs, which can be verified by Burrata contracts. Claims are typed and namespaced. For instance, the Web$_3$Cabin might use a claim of type `cabin.contract` to represent a signed contract, and `cabin.lockbox` to represent the state of the lockbox. Claims can also contain key-value pairs of data, where a key is a 16 bit integer. For instance, the `cabin.contract` claim may contain a document id field at key 100, and the `cabin.lockbox` claim can contain a state field at key 100.

Data hubs can sign and present claims in a number of presentation formats, depending on the use case. The default claim format is the EVM-optimized format (Figure 2), which is designed to minimize gas usage on EVM blockchains. Other blockchains can use different formats depending on what primitives are available. For instance, on chains with native ed25519 support [9], such as Near [30], Sui [31] and Solana [37], ed25519 is used and the data hub public key is added, since recovery is unsupported. Burrata data hubs can also present verifiable data to be used in a Web$_2$ context as well, and issues claims conforming to the W3C Verifiable Credentials specification [21, 34], and in the case of identity, conforms to the W3C Decentralized Identifier (DID) specification [18, 33].

Claims also support services, which turn on additional verification by the Burrata contracts. These services inject their data into Burrata claims as a key-value pair separate from the claim data. For example, a verifiable data service like DECO or TLSNotary may add information about the data request and a signature at key 100.

Importantly, claims have a limited lifetime, which minimizes the number of blockchain interactions required when a claim is revoked. Each hub keeps track of the “live” claims that is has issued and only needs to revoke claims which are “in flight”. Users need to refresh claims, but this is handled automatically by the frontend: as long as a previously issued

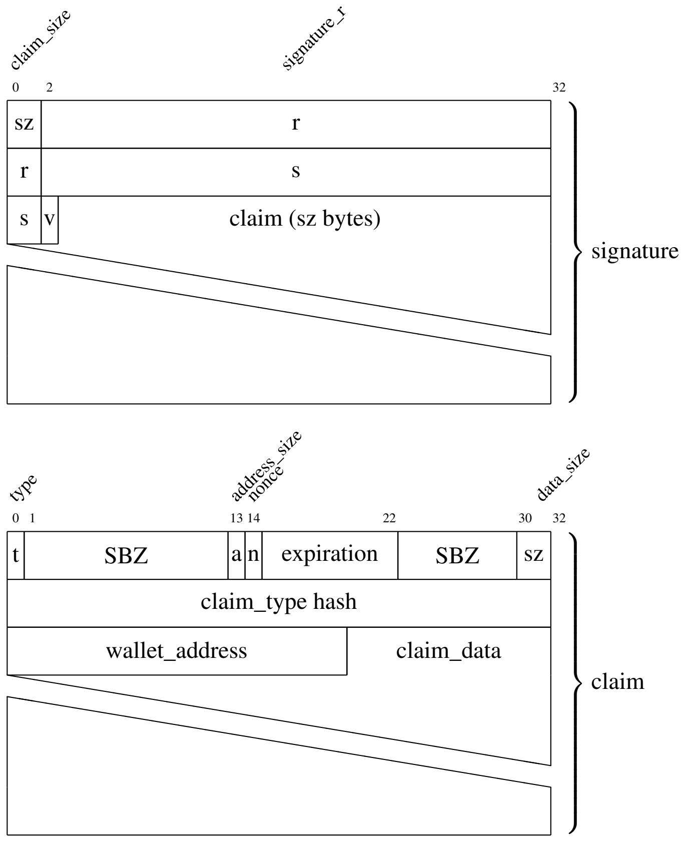

Figure 2: A Burrata claim, in an EVM-optimized presentation format.

```
async function handleRent(e : event) {
    const rental_id = e.target.getAttribute("
        rentalId");
    const document_id = e.target.getAttribute("
        documentId");
    const claim = getClaim({
        key: DOCUMENT_HUB_PUBKEY,
        type: "cabin.contract",
        args: {
            document_id
        },
        input_claims: [{
            key: IDENTITY_HUB_PUBKEY,
            type: "cabin.identity.name"
        }]
    })
    return rentalContract.rent(rental_id, claim, {
         value: deposit } );
```

Figure 3: Modifications to the Web3Cabin frontend code, showing the new `getClaim()` call.

claim has not been revoked, a data hub will sign the same claim, updating the expiration and put it back in the “live” set.

3.3 Using Burrata

To use Burrata, the Web3Cabin developer links to both the Burrata frontend SDK, and the Burrata contracts. Suppose that the Web3Cabin developer has already built the frontend that the hosters and renters use to manage and make reservations, as well as the contracts that facilitate payment and handle reservations. Now, the developer needs to bring in Web2 data which validates that the renter has signed the contract and the state of the lockbox.

First, the developer needs to create the claim types which carry this data. The developer can either choose to deploy their own data hub, or use a data hub deployed by Burrata with existing integrations[3]. For the purposes of this paper, the developer chooses the Burrata deployed integrations, and creates a claim for document signing called `cabin.contract` and a claim form the lockbox called `cabin.lockbox`. In addition, the developer would like to ensure that the name of the person signing the document matches a legal identity document in their possession, so they create a claim called `cabin.identity` as well. `cabin.identity` is not used on the blockchain. Instead, `cabin.contract` is configured to only be issued if the name of the signer matches the name in the `cabin.identity` claim.

Next, the developer modifies their frontend to make a call to the Burrata frontend SDK before calling their contracts. Before making the call to `rentCabin()` function, the developer calls the Burrata `getClaims()` function, indicating the public keys of the data hubs and the types of claims to be retrieved. For each claim, `options` can also be specified which are passed on to the hubs. For example, the `cabin.contract` claim takes a `document_id` field. In addition, an `input_claims` field specifies which claims should be retrieved and sent to the hub. `cabin.contract` takes a `cabin.identity` so that the document signing data hub can ensure that the signature matches the legal name of the user. The developer then adds the collected data as the final parameter to their contract call. The collected claims data is sent as calldata in the users' transaction. When the user's browser executes the `getClaims()` function, the user is presented with an interface which collects the required data.

[3]Burrata comes with a number of pre-deployed integrations which provide identity, financial data, document signing and more. For more details, see http://burrata.xyz.

```
contract RentalContract {

    ...

    function rent payable (bytes32 rental_id,
        bytes calldata claim) {
        bytes claim_data = Burrata.checkClaim(
            claim, "cabin.contract");
        bytes32 document_id = bytes32(Burrata.
            extractClaimData(claim_data, 100));
        if (document_id != rentals[rental_id].
            contract_id) {
            revert WrongContractSigned();
        }
        if (msg.value < rentals[rental_id].value)
            {
            revert DepositTooSmall();
        }
        ...
    }
}
```

Figure 4: Modifications to the Web3Cabin solidity contract code.

The contract code for Web3Cabin now needs to be modified to take the updated data and verify that it is correct. The changes to Web3Cabin are shown in Figure 4. The collected claims data is a calldata parameter, which costs only 16 gas per non-zero byte. An average 100 byte claim costs 1600 gas, which is about 2% of a typical ERC-20 transfer [15]. The calldata is then passed to the `checkClaims()` function, which takes the name of the claim to be checked, and returns the claim data. The function errors out if the claim is invalid. Since the contract needs to check that the `document_id` matches the contract the host has created, the contract calls `extractClaimData()` with the 100, the key for `document_id`, and checks that it matches what is stored for the host. In the `checkout()` function, a similar mechanism is used, but the `lockbox_id` (100) and `lock_state` (101) fields are checked instead.

3.4 Contracts

The Burrata contracts contain the logic for checking whether a claim is valid or not as well as the valid data hubs and revoked claims. The API of the contracts are shown in Figure 5. The Burrata contract is replicated across all chains that we support. We expect the state of the contract to be updated infrequently, only when a new data hub or claim type is added, or when a claim is revoked.

When a developer calls the `checkClaim()` function, a contract call is made into the Burrata core contract. The core contract first uses the `ECRECOVER` precompile to recover the public key of the data hub that signed the claim and retrieves the hub data. It then checks to make sure that the hub is authorized to sign messages for the given claim type. This check consumes 2 `SLOAD` operations, the costliest part of the check. The contract then checks if `tx.sender` matches the wallet address the claim was issued to. The reason `tx.origin` is used for the check instead of `msg.sender` is to ensure that no one else can use the claim through an intermediate smart contract. Support for multi-signature wallets is handled through the frontend, which inserts the correct wallet address into the claim based on the final sender of the claim. Using `tx.origin` is not a security risk because it not used for authorization but to check that the claim owner holds the private key to the wallet. Otherwise, a claim could be used by multiple users. The `checkClaim()` function then checks if services are enabled for that claim type. For instance, if data verification is turned on, the the service handler for data verification is called (100) with the payload in the claim. The nonce for the claim is then checked to make sure that an explicit revocation has not occurred. Finally, the expiration time of the claim is checked. The expiration time is expected to be fuzzy, so an error of ± 1 minute is expected and the contract is not vulnerable to timing attacks. If all checks pass, the function returns with the claim data which the developer can access using the library `extractClaimData()` function. Otherwise, the function errors and causes the transaction to revert.

Another contract, the Burrata Registry, is responsible for maintaining the Web_2 endpoints of the data hubs. This contract is primarily used for clients to find data hubs and not called by other smart contracts. To store flexible document-based data, the contracts only store IPFS [6] hashes, and the IPFS document contains the full data for the registry.

Finally, the Burrata governance contract handles governance, such as deciding which hubs and types are enabled. The details of the governance contract are outside of the scope of the paper.

3.5 Data Hubs

The data hubs are at the core of the system, bridging data from Web_2 to Web_3. While we refer to the data hubs as one component in the rest of the paper, they are actually composed of a user-facing interface, known as the onboarding mechanism, and a backend, which services claim requests. Each data hub services one or more integrations, which are connections to a Web_2 provider.

3.5.1 Onboarding

The onboarding mechanism is the user-facing part of a data hub. The purpose of onboarding is to collect data from the user's browser and forward that data to the backend. To that end, onboarding is usually a static page, served in IPFS.

When the onboarding page is opened by the frontend, it is given an `onboarding_payload`, which starts a flow in the users browser. For instance, in the case of an identity claim, a user can be asked to take a picture of their identity documents, or for a signature claim, the user can be presented with a document to sign.

Importantly, the onboarding page is completely untrusted: the users browser can do anything with the page, but it is expected that this interaction results in a change that the backend can query. For instance, in the case of an identity service, the `onboarding_payload` could consist of a token which opens the identity provider's web flow, and the backend could then query whether the flow succeeded or not. Optionally, the flow may return data to the hub, `onboarding_return`. However, this data is untrusted but could be used to check for instance, if a user logged in successfully via OAuth.

This flow matches many familiar Web_2 services where developers are issued a "publishable" key and a "private" key for their backends. In fact, many of our integrations simply link with the Web_2 services' SDK.

3.5.2 Backend

The backend component interacts with the frontend to issue claims. It exposes a single websocket, which is used by the client to manage claims in an RPC-like fashion, the API is shown in Figure 6.

The first operation the backend expects is authentication of a users' wallet. In order to authenticate the user's wallet, the frontend sends a message signed by the user in an `AuthRequest` message. For EVM, the hub expects a signed message in the EIP-4361 (Sign In With Ethereum) format [12], while for other chains, we expect a message signed in CAIP-122 (Sign in with X) format [17]. The hub validates the URI, signature and the nonce in the message. The request ID field of the message contains the public key of a client-generated key, the session key. We use a session key because it eliminates the need for the user to constantly authorize signing prompts in their wallet. The hub responds with an `AuthResponse` message, containing a challenge. The frontend then responds with a `ChallengeRequest`, signing the challenge with the session key, and the authentication process is completed when the data hub responds with

Method	Description
Burrata Core Contract	
`checkClaim(claim, type)` → *data*	Extract claim of *type* from array of claims *claim* and return unparsed *data* if present.
Burrata Registry Contract	
`getProviderIpfsHash(provider)` → *ipfs_hash*	Return *ipfs_hash* for a *provider* public key.
`getClaimIpfsHash(claim)` → *ipfs_hash*	Return *ipfs_hash* for a hashed *claim* type.
Burrata Utility Library	
`extractClaimData(data, field)` → *field_data*	Extract *field* from claim *data* and return *field_data*.

Figure 5: The Burrata contract API, showing the public APIs of the core contract, registry contract and utility library provided to developers.

Method	Description
`auth(message)` → *challenge*	Authenticate wallet with signed *message* and session key, returning a *challenge*.
`challenge(signed)` → *success*	Show *signed* challenge with session key, return *success*.
`getClaim(type, format, data)` → *claim*	Retrieve a *claim* of *type* in *format*, providing optional *data*.
`onboard(type, format, data)` → *payload*	Retrieve onboarding *payload* for *type* in *format*, providing optional *data*.

Figure 6: The Burrata Data Hub API. The API consists of RPCs driven by the frontend, so the `auth()` call consists of an `AuthRequest` from the frontend and an `AuthResponse` from the data hub.

`ChallengeResponse`, indicating success.

Once the data hub is assured that the user possesses the given wallet, it is ready to issue new claims. The frontend calls `GetClaimRequest` with the type of claim requested, the desired claim format, and any additional data needed to issue the claim. If no user browser interaction is necessary (for instance, the user has already finished the onboarding process), then a `GetClaimReponse` is issued with the claim. Otherwise, the frontend begins the onboarding process by calling `OnboardingRequest` with the same data. The hub then responds with an `OnboardingResponse` containing the URI of the onboarding mechanism and a `onboarding_payload`. Once complete, the frontend calls `GetClaimRequest` again to retrieve the claim. Usually, in the onboarding request, the backend associates the users wallet address (or hash of), so that the users' data can be retrieved the next time `GetClaimRequest` is called. A few services do not permit either association or search of metadata. In this case an external key-value could be used (e.g. Redis) to store the association.

In some cases, the Web$_2$ service may take some time to update. For instance, an identity service may require manual verification by a person. In this case, the hub responds with a `GetClaimResponsePending` which informs the user when they need to check back to retrieve the claim.

3.5.3 User Data Verification

To ensure that the data hub actually communicates with the data source, user data verification enables a data hub to generate a proof that enables third parties with access to the encrypted TLS session transcript to verify that the communication has actually taken place. Third parties never have

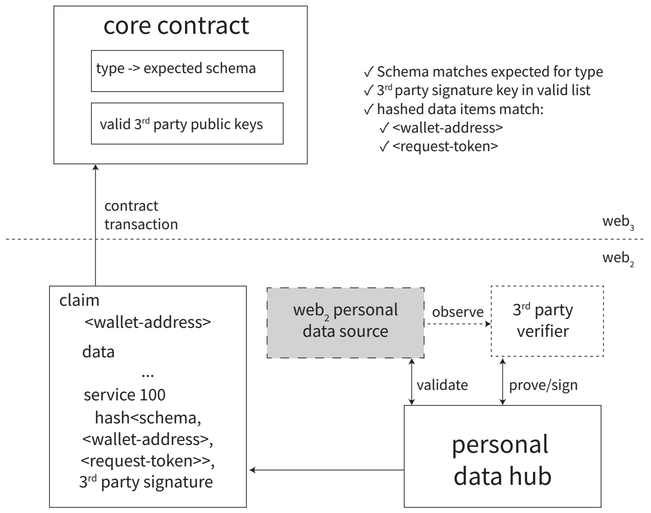

Figure 7: The Burrata User Data Verification Architecture. When validating onboarding from a Web$_2$ data source, the data hub establishes a three-way TLS session with a 3rd party verifier, which observes the encrypted traffic. The data hub generates a zero knowledge proof which the verifier signs, and the verifcation is inserted into the service section of the claim, which the Burrata core contract verifies.

access to the raw data, rather, the data hub provides a zero knowledge proof.

The overall architecture of the data hub's user data verification feature is shown in Figure 7. The data hub provides a `fetch()`-like interface [23] to enable integrations to generate these proofs and to establish the required 3-way TLS session. In addition to the parameters used for the `fetch()` request, integration developers specify a schema, which is a `jq` query [13] on the returned data and expected return result. The data hub takes care of establishing the 3-way TLS session and collecting signatures from verifiers. The returned claim contains the signatures and hash of the schema used, which is a hash of the `jq` query and hash of the expected query result.

When the Burrata contract has the service for user data verification turned on, the contract checks that the signatures matches what are expected for the service, and that the correct schema and inputs were used.

Notably, when the data hub and the Web$_2$ service belong to the same trust domain (i.e., if the data hub is compromised, then the Web$_2$ service is comporomised), user data verification can be turned off as an optimization.

3.6 Frontend

The frontend is responsible for managing a users' claims, authenticating the user and storing session keys, as well as interacting with data hubs. While the frontend supports multiple blockchains, the core abstractions (Figure 8) are designed to be chain-agnostic so that support for new blockchains can be easily added.

The dApp developer does not need to be aware of the frontend internals: after initializing the library they simply call `getClaims()` with target hub public key, claim type and request data.

3.6.1 Finding The Data Hub

The first task the frontend code must accomplish is finding the data hub. To do this, the frontend locates the Burrata registry contract and makes a call to `getHubInfo()` using the hub's public key. This returns an IPFS hash containing the data for the hub. The frontend retrieves this page through IPFS [6], which contains a list of all the current valid URIs of hubs that can serve that claim type. The frontend then, for load balancing, randomly selects a hub to connect to.

We could have chosen to publish the hub key to URI mapping on a Web$_2$ page. However, using a contract keeps Burrata decentralized, and a governance decision is required to update the IPFS hash for each page.

This logic summarizes the EVM-based version of the frontend. Due to differences in wallet implementations across blockchains, the logic may slighty differ from chain to chain, but abstractly, for each chain we implement `getHubUrl()`, which returns a URL for a given public key.

3.6.2 Establishing a Data Hub Connection

Each data hub advertises a WebSocket which requires that the client authenticate ownership of a wallet address. Most wallets require user interaction every time data is signed, which makes it difficult to create dynamic flows, and would require user interaction every time a new data hub is accessed. Instead, we opt for the user to sign a token which contains a public key, and the corresponding private key is stored in the browser's local storage. When the frontend connects to a hub, it follows the protocol described in section 3.5.2, using the session key to authenticate. This authentication method is similar to Sign-on-With-Ethereum and Sign-in-With-X.

Abstractly, the per-chain implementation needs to implement `signTokenMessage()`, which takes the token and signs it with the user's wallet.

3.6.3 Retrieving Web$_2$ data

If the claim can be generated without input from the user, the data hub can generate the claim without further input. Most claims, however, will require data from the user. As we indicated previously, it is important to note that the user's data is untrusted: we cannot directly embed data from the client into a claim.

As outlined in section 3.5.1, the data hub can initiate an onboarding flow, which passes an `onboarding_payload` to the browser. The frontend presents the URL contained in this payload, which could, for example, ask the user to login to a page or sign a document. The data hub then issues a claim based on the conditions configured for that integration.

The frontend saves retrieved claims in the browsers' local storage. The claims have a short expiration time, but the data hub will "refresh" expired claims on request as long as they have not been revoked.

Retrieving claim data is blockchain agnostic, but the presentation format can differ from chain to chain, as discussed in section 3.2. The data is returned in the `getClaim()` function as a byte array.

3.6.4 Submitting Data To The Contract

Once the claim data is retrieved, the developer adds the byte array returned by `getClaim()` as a parameter to their contract call. The `checkClaim()` call will revert if the claim is invalid, and `getClaim()` can be called again to refresh the claim if necessary.

4 Future Work

Burrata is a continual work in progress. Apart from constantly adding support for new chains, we also plan on supporting these new features in our immediate roadmap:

Method	Description
`getHubUrl(public_key)` $\rightarrow$ *ipfs_cid*	Get the *ipfs_cid* for the data hub with *public_key*.
`signTokenMessage(message)` $\rightarrow$ *signature*	Sign a *message*, returning the *signature*.
`getClaimData(claim, type)` $\rightarrow$ *claim_data*	Extract the *claim_data* from the *claim* of *type*, returning failure if the claim is invalid.
`getClaimType()` $\rightarrow$ *type*	Retrieve the *type* of claim required by this blockchain.

Figure 8: The Burrata Frontend Core Abstractions. Support for additional blockchains requires implementing these functions.

On-chain user data verification A number of new blockchains, such as Sui [31] natively support zero knowledge verification of algorithms such as Groth16 [16]. This makes it possible to perform user data verification without relying on third-party verifiers. Instead, the TLS transcript can be submitted on-chain for verification.

In addition, in situations where Web$_2$ data can be accessed through multiple 3rd party verifiers, the data hub may not need to provide a proof, since the verifiers can access the data themselves.

Deeper Web$_2$ integration Burrata data hubs already support outputting W3C standards compliant verified credentials and decentralized identifiers, which have great use cases in the Web$_2$ environment as well. We plan on making a version of the contract-level `checkClaims()` function available to Web$_2$ applications, both in the browser and in backends.

Self Sovereign Identity Self sovereign identity is an emerging paradigm which enables users to hold their identity credentials without relying on a third party to persist their data. A number of solutions now exist: data stores such as Kepler [20], Privy [29] and Ceramic [11] enable users to store their data locally and authenticate with a number of providers, and Microsoft Entra [24] enables users to access identity through the W3C Decentralized Identifier (DID) [33] standard.

Burrata already supports interoperability with these standards and generates W3C standardized VCs [34] and DIDs, and users can already manually store their Burrata claims with these providers. Future work includes deeper integration with self-sovereign identity stacks and enabling users to store and access Burrata claims seamlessly through these providers.

5 Conclusion

For Web$_3$ to fulfill its promises, it must be able to seamlessly operate on data from Web$_2$. Burrata enables Web$_3$ smart contracts to consume Web$_2$ data without increasing friction for users or developers. Burrata also opens a path to many applications which need access to personal data efficiently on-chain. We look forward to seeing the new classes of Web$_3$ applications that Burrata enables.

References

[1] 1inch. 1inch, one-stop access to decentralized finance. https://1inch.io, Accessed on Nov 22, 2022.

[2] aave. Aave — open source liquidity protocol. https://aave.com/, Accessed on Nov 22, 2022.

[3] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.*, 2021.

[4] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. {FARSITE}: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, 2002.

[5] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020.

[6] Juan Benet. IPFS-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[7] Abdeljalil Beniiche. A study of blockchain oracles. *arXiv preprint arXiv:2004.07140*, 2020.

[8] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. *Chainlink Labs*, 2021.

[9] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of ed25519: theory and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1659–1676. IEEE, 2021.

[10] Buzz Cai. EIP-5484: Consensual soulbound tokens. https://eips.ethereum.org/EIPS/eip-5484/, Accessed on Nov 22, 2022.

[11] Ceramic. Ceramic: Composable Web3 data network. https://ceramic.network/, Accessed on Nov 22, 2022.

[12] Wayne Chang, Gregory Rocco, Brantly Millegan, and Nick Johnson. EIP-4361: Sign-in with ethereum. https://eips.ethereum.org/EIPS/eip-4361, Accessed on Nov 22, 2022.

[13] Stephen Dolan. jq is a lightweight and flexible command-line json processor. https://stedolan.github.io/jq/, Accessed on Nov 22, 2022.

[14] Compound Finance. compound. https://compound.finance/, Accessed on Nov 22, 2022.

[15] Ethereum Foundation and Fabian Vogelsteller. ERC-20 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, Accessed on Nov 22, 2022.

[16] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 339–358. Springer, 2006.

[17] Haardik and Sergey Ukustov. Sign in with x (siwx). https://github.com/ChainAgnostic/CAIPs/blob/master/CAIPs/caip-122.md, Accessed on Nov 22, 2022.

[18] Harry Halpin. Vision: A critique of immunity passports and w3c decentralized identifiers. In *International Conference on Research in Security Standardisation*, pages 148–168. Springer, 2020.

[19] Samer Hassan and Primavera De Filippi. Decentralized autonomous organization. *Internet Policy Review*, 10(2):1–10, 2021.

[20] Kepler. Kepler is self-sovereign storage. https://www.kepler.xyz/, Accessed on Nov 22, 2022.

[21] Romain Laborde, Arnaud Oglaza, Samer Wazan, François Barrere, Abdelmalek Benzekri, David W Chadwick, and Rémi Venant. A user-centric identity management framework based on the w3c verifiable credentials and the fido universal authentication framework. In *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–8. IEEE, 2020.

[22] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[23] mdn web docs. Fetch api. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API, Accessed on Nov 22, 2022.

[24] Microsoft. Link your domain to your decentralized identifier (did). https://learn.microsoft.com/en-us/azure/active-directory/verifiable-credentials/how-to-dnsbind, Accessed on Nov 22, 2022.

[25] Polymath Network. Polymath network. https://polymath.network/, Accessed on Nov 22, 2022.

[26] Pyth Network. Pyth network. https://pyth.network/, Accessed on Nov 22, 2022.

[27] Wormhole Network. Wormhole. https://wormhole.com/, Accessed on Nov 22, 2022.

[28] OpenSea. Opensea, the largest NFT marketplace. https://opensea.io, Accessed on Nov 22, 2022.

[29] Privy. Privy. https://www.privy.io/, Accessed on Nov 22, 2022.

[30] Near Protocol. Near. https://near.org/, Accessed on Nov 22, 2022.

[31] The MystenLabs Team. The sui smart contracts platform. https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf, Accessed on Nov 22, 2022.

[32] TLSNotary. Tlsnotary — proof of data authenticity. https://tlsnotary.org/, Accessed on Nov 22, 2022.

[33] W3C. Decentralized identifiers (dids) v1.0. https://www.w3.org/TR/did-core/, Accessed on Nov 22, 2022.

[34] W3C. Verifiable credentials data model v1.1. https://www.w3.org/TR/vc-data-model/, Accessed on Nov 22, 2022.

[35] E Glen Weyl, Puja Ohlhaver, and Vitalik Buterin. Decentralized society: Finding web3's soul. *Available at SSRN 4105763*, 2022.

[36] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[37] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

[38] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1938, 2020.

[39] Micah Zoltu. EIP-5114: Soulbound badge. https://eips.ethereum.org/EIPS/eip-5114/, Accessed on Nov 22, 2022.