Orbis System Architecture

Orbis Labs

March 4, 2022

Orbis is a general purpose ZK rollup layer 2 scalability solution for Cardano. ZK rollups are a mechanism by which transactions are processed off-chain, and their results are published on the blockchain using zero-knowledge proofs. More precisely, ZK rollups use zkSNARKS, which stands for *zero-knowledge succinct non-interactive arguments of knowledge*.

By processing data off-chain, ZK rollups allow for higher transaction throughput on the blockchain and in the ecosystem as a whole. By using a secure, decentralized layer 1 blockchain and zkSNARKS, ZK rollups inherit certain security properties of the layer 1 solution: namely, that we need not assume any party is trustworthy and that no one is able to cause unlawful (i.e., protocol-violating) results to be produced on-chain. Additionally, Orbis will contain no single point of failure, meaning that once a mature implementation is in place, then no individual or organization's ongoing participation is required for the proper, intended functioning of the solution.

ZK rollups are an increasingly popular choice for layer 2 scalability on Ethereum, with multiple choices of ZK rollup solutions available for use, including zkSync, StarkNet, and Loopring. [1] In contrast, Orbis Labs knows of no prior art for a ZK rollup-based layer 2 solution on Cardano. However, we believe that ZK rollups will be an important part of the Cardano ecosystem moving forward, just as they are in Ethereum today.

This is the system architecture document for Orbis. This document defines the key components of the Orbis protocol and their interactions with each other and the outside world, as well as the rationale for these design decisions. This document deviates from a whitepaper or a technical spec in that it does not attempt to provide enough information to implement Orbis. Instead, one can think of it as a sketch. Technical specs will fill the picture in much greater detail, sufficient to guide implementation, while a technical

whitepaper will describe the Orbis protocol in a form more suitable for general consumption. Instead, this document simply provides an overview of the Orbis protocol and its defining features for a technical audience.

1 Background

This section covers the background information necessary to understand the design of Orbis and ZK rollup solutions more generally.

A blockchain uses a consensus algorithm executed by a decentralized network of nodes to process financial transactions and other information. Thanks to the consensus algorithm, parties can be assured that their money and other informational assets stored on the blockchain will not be used in a manner they did not authorize, without the need to vet the trustworthiness of any particular participant in the protocol.

Blockchains are an important innovation in financial technology; for starters, they have the potential to liberate humanity from the gatekeeping associated with centralized systems of finance. However, such liberation will only be possible to the extent that blockchain developers can achieve massive scalability, to the point that blockchains could feasibly replace centralized financial technology as a transaction processor. For example, Visa reportedly processes an average of over 150 million transactions per day and is capable of processing more than 24,000 transactions per second. [2]

Scalability of blockchain solutions is a widely recognized problem. However, a few angles of attack are available to address it. One angle is to increase the transaction throughput of the blockchain itself. This is an important task but also a difficult one. Another angle is to multiply the number of blockchains in use, such as in the case of Kadena's Chainweb. [3] A third angle is to use so-called layer 2 solutions, whereby transactions are processed off-chain and their results are later synced onto the chain. With layer 2 solutions, the blockchain can simply be used as a settlement layer while application domain business logic lives off-chain, greatly reducing the amount of state and state changes stored on the chain.

ZK rollups are an example of a layer 2 solution strategy. ZK rollups make use of zkSNARKs. zkSNARKs are a type of proof. A *proof*, for our purposes, is a piece of information that evidences the truth of some statement. In the context of formal logic, a proof is a sound or valid argument in which the premises are true. A *valid* argument, speaking informally, is an argument

that, in virtue of its logical structure, guarantees the conclusion is true provided that the premises are true. This is *not* what we mean by a proof, here. In this context, a *proof* is a piece of information whose existence demonstrates only a negligible probability that the statement to be proven is not true. In this sense of the word "proof," for example, an ED25519 signature is a "proof" that a person who possesses the private signing key ran the signing algorithm to sign the piece of information.

By definition, a zkSNARK is a zero-knowledge succinct non-interactive argument of knowledge. In this description, "argument of knowledge" means "proof" (in the sense of "proof" defined in this context for our purposes). A zkSNARK has the characteristics of being zero knowledge (meaning it does not prove anything more than the statement to be proven), succinct (meaning the proof can be represented as a small number of bytes), and non-interactive (meaning the proof can be represented as a single message as opposed to a multi-step interaction between the proving entity and the proof-checking entity).

For our purposes, the most relevant characteristics of zkSNARKs are their succinctness and non-interactivity. It's possible that we could get by with just SNARKs, succinct non-interactive proofs that aren't necessarily zero-knowledge. However, we are using zkSNARKs because the bulk of the research available is on zkSNARKs, and we believe that zkSNARK solutions are the most mature and relevant SNARK solutions available.

The reason succinctness and non-interactivity are key properties here is that we need to store the proofs in transactions, which are short, single messages, on the blockchain.

ZK rollups are, in essence, a simple concept. In a pure blockchain protocol (without a layer 2 solution), all transactions, smart contract code, and smart contract state data are normally stored on the blockchain. In a ZK rollup solution, some of this code and data are stored off-chain. In its place, the chain stores proofs that this information exists. The zkSNARKs stored on-chain prove that, for some set of inputs and some set of outputs, a set of transactions exist that were signed by the relevant parties and that followed the rules of the relevant smart contracts. Those transactions never need to be recorded on the chain; we need only record the inputs, the outputs, and the proof that those outputs lawfully (according to the blockchain protocol) resulted from the inputs.

zkSNARKs allow for a potentially unlimited number of transactions to be performed with a constant $O(1)$ amount of on-chain information. There-

fore, this ZK rollup strategy potentially allows for very efficient usage of the blockchain's limited information storage capacity.

Similarly, the computational complexity of verifying a zkSNARK proof is $O(1)$. Therefore, the ZK rollup strategy allows for a potentially unlimited amount of computation to be performed off-chain to validate transactions, which are all verified by a single proof. The on-chain computation required to validate those transactions is limited to the relatively small and $O(1)$ amount of computation required to check the proof. This makes for a potentially very efficient usage of the blockchain's limited computational capacity.

1.1 Decentralized data stores and throughput

A blockchain is only one type of decentralized data store. Its two main ingredients are a decentralized data store for a sequence of transactions (i.e., a distributed ledger) and a means by which people can agree on how to update it (i.e., a consensus algorithm).

A blockchain is a very general-purpose piece of equipment that solves a hard problem. A difficult part of this problem is, how do we agree on how to extend the ledger when parties may have submitted conflicting (individually satisfiable but jointly unsatisfiable) requests? The current state of the tech allows for solutions to this problem that are reliable but limited in terms of throughput.

Other ways of implementing decentralized data stores (other than blockchains) exist, and they may provide greater performance by not solving the entirety of the problem that a blockchain solves. One example of this is a content-addressed distributed filesystem, e.g. Dat [4] or IPFS [5]. A content-addressed filesystem does not require the same type of sophisticated consensus algorithm that a blockchain requires. If the only operations that mutate the data store are additions of new data, and that data is content-addressed and, thus, contains no possibility of conflict between two additions of data, then the data store can simply accept all requests to add data without needing a conflict resolution mechanism. This simplification of the problem may allow for a more performant implementation with higher throughput.

A decentralized filesystem carries a performance penalty compared to a traditional filesystem. In a traditional filesystem, all files are stored on one drive, with blocks of data organized into a tree structure to make them easy to find. In a decentralized filesystem, blocks of data are still organized into a tree structure, but this structure is stored across many drives connected

to many different computers. Therefore, searching the tree requires network communication between peers.

For storing a small amount of data, a decentralized filesystem is probably not the right kind of decentralized data store. If the amount of data to be stored is small, then all of the data can be held in RAM. Updating such a decentralized data store still carries a cost premium because it still requires network communication between peers. However, this kind of decentralized data structure eliminates the need to search through a large amount of data stored in tree structures on drives. For this, we can expect a performance benefit compared to using a decentralized filesystem.

Let's imagine a decentralized data structure based on two assumptions: the amount of data to be stored is small, and the ways in which it can be updated do not conflict with each other. For example, consider a decentralized data store designed to store a set with less than 1,000 elements. The only way to update it is to add an element to the set. Each element is timestamped with the time when it was added. When the cardinality of the set reaches 1,000, the oldest element of the set is dropped to make room for an additional element. This is an example of a conflict-free replicated data type (CRDT). This type of decentralized data store seems particularly amenable to an efficient implementation with high throughput due to its relative simplicity of negotiating updates to the data store, the entirety of which can be held in RAM.

We can extend this idea to the concept of an ordered set, where the elements are ordered by the timestamp. This is, again, a CRDT. One variation on this idea is a decentralized queue, which contains two operations: adding to the end of the queue and removing from the start of the queue. This type of queue can be considered a CRDT depending on how we define the removal operation. If the removal operation states "remove this element from the queue if it exists and is the beginning of the queue," then the removal operation is conflict-free. This type of removal operation is subject to race conditions, where different orderings of removal operations may have a different end result. However, if we assume that a removal operation is only initiated on an element x when x is or recently was the start of the queue, then there are no race conditions between removal operations. If two identical removal operations are submitted around the same time, then the result will be the same regardless of the order in which they occur.

2 System components

Orbis has two main components: the prover and the verifier. See Figure 1 for a picture of these components and their contact points with each other and the rest of the world.

The prover is an off-chain web service, and its essential purpose is to construct zkSNARKs that verify transaction occurrence. The prover has an API similar to a blockchain node; it allows for posting transactions and inspecting state data. Unlike a blockchain node, which allows for posting transactions to the blockchain and inspecting the state of the blockchain, the prover allows for posting transactions to a rollup and inspecting the state of a rollup. Here, "rollup" refers to a sequence of transactions and a collection of state data that will, at some future time, be used to create outputs on the blockchain. The rollup is stored in the prover's persistent state mechanism (i.e., a database) rather than on the blockchain. Therefore, while the prover has an API similar to that of a blockchain node, it is something else entirely.

The verifier is an on-chain smart contract. Its essential purpose is to settle transactions performed on-rollup. The verifier contract accepts on-chain inputs, locking them up in the contract so they can be used on the rollup without the risk of them being double-spent once they're on-chain and on-rollup. Also, the verifier contract validates prover-created transactions, which contain outputs from the rollup and proofs that those outputs resulted from a series of valid transactions based on inputs provided to the verifier contract.

Orbis provides an off-chain context in which smart contract validator code can run. Instead of being run to create transactions on the chain, validator code is run to create transactions on the rollup and proofs that those transactions belong on the rollup.

It is important to keep two terms distinct here: *verifier* and *validator*. "Validator" refers to a function that takes a transaction as input and outputs a boolean value indicating whether the transaction satisfies the rules of a smart contract. "Verifier" refers to the smart contract validator which validates transactions over the rollup smart contract. The verifier is a validator, but other validators are not verifiers. To avoid confusion, let the term "validator" refer to the validator for some smart contract running on the *rollup*, as opposed to the verifier, which runs on the *blockchain*.

Orbis will use Halo 2 [7, 6, 8] as the zkSNARK proving system. Halo 2 has two key properties that make it suitable for this application. First, it

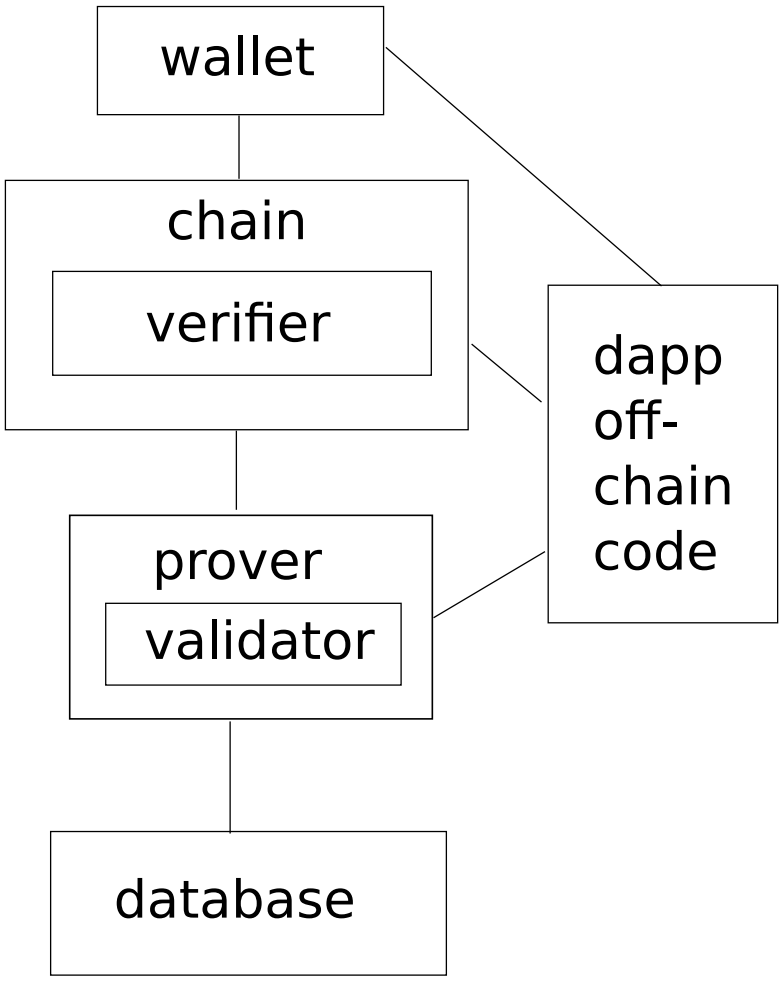

Figure 1: Key components of and related to Orbis.

has small proof sizes, of several kilobytes or less, small enough to fit in a Cardano transaction. Second, it supports proof carrying data, which can be used to build recursive proofs.

Proof carrying data refers to the process of giving a zkSNARK as input to an arithmetic circuit in order to generate another zkSNARK, such that the generated zkSNARK proves that the input zkSNARK is valid, indirectly proving the truth of whatever statement the input zkSNARK proves. Proof carrying data enables recursive proofs when a statement checked by an arithmetic circuit takes as input a zkSNARK for the same statement.

Recursive proofs are key to Orbis' strategy for building a ZK rollup which can scale to any required level of throughput. See Section 4 for more about this.

Using Orbis somewhat complicates the process of developing a dapp (decentralized application), in the case where the dapp does not operate exclusively on the rollup, but also does some of its operations on the chain directly. In that case, the dapp must be aware of and control the movement of information from the chain to the rollup and back again. It must publish and subscribe to data not only on the chain but also on the rollup. The dapp interacts with the rollup by calling the prover API. The next section covers the processes involved in more detail.

3 Processes

There are three main processes involved in a dapp using Orbis: moving inputs from the chain to the rollup, posting transactions on the rollup, and moving outputs from the rollup to the chain. These processes are pictured in Figures 2, 3, and 4.

3.1 Process of posting a transaction to a rollup

To post a transaction to a rollup, the dapp code must first call the prover API to lock the on-rollup input UTXOs. The purpose of this step is to avoid resource contention problems that would prevent the transaction from going through. These locks should be temporary and expire automatically after a brief time sufficient to complete the transaction. To avoid denial of service (DoS) attacks based on this locking mechanism, the user requesting the transaction should be required to post collateral that will be lost if the

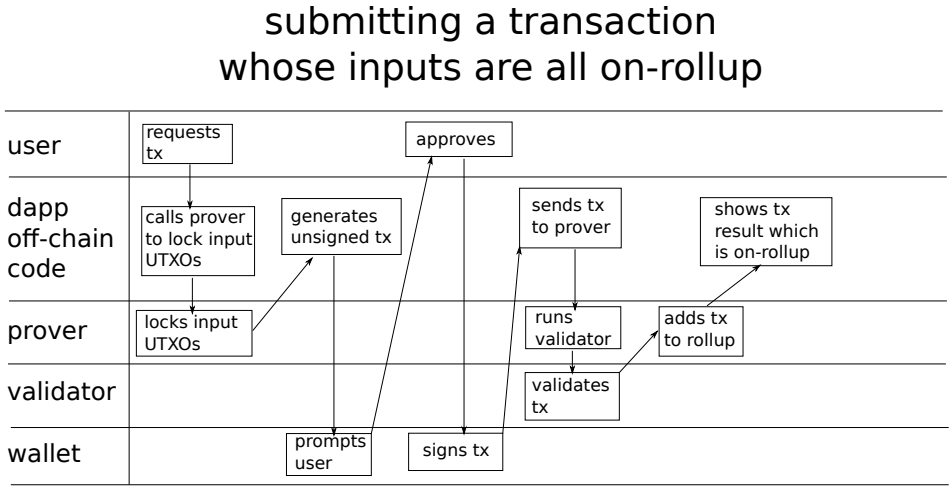

Figure 2: Process of posting a transaction to a rollup.

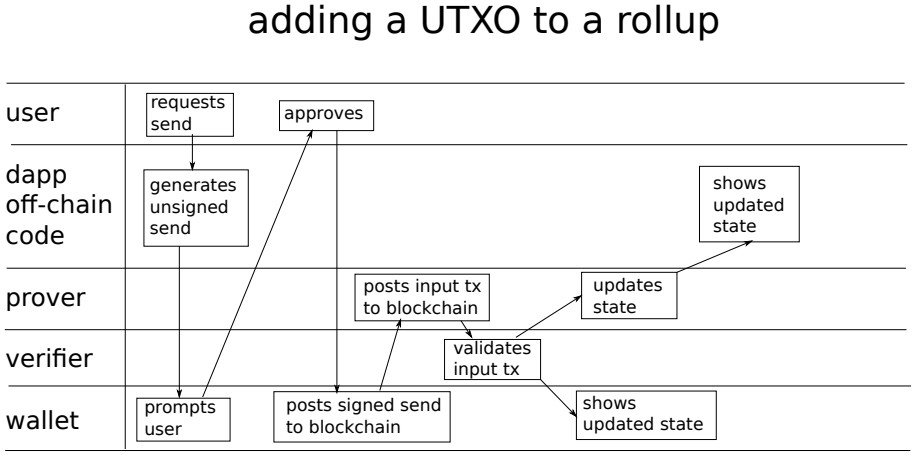

Figure 3: Process of adding an input to a rollup.

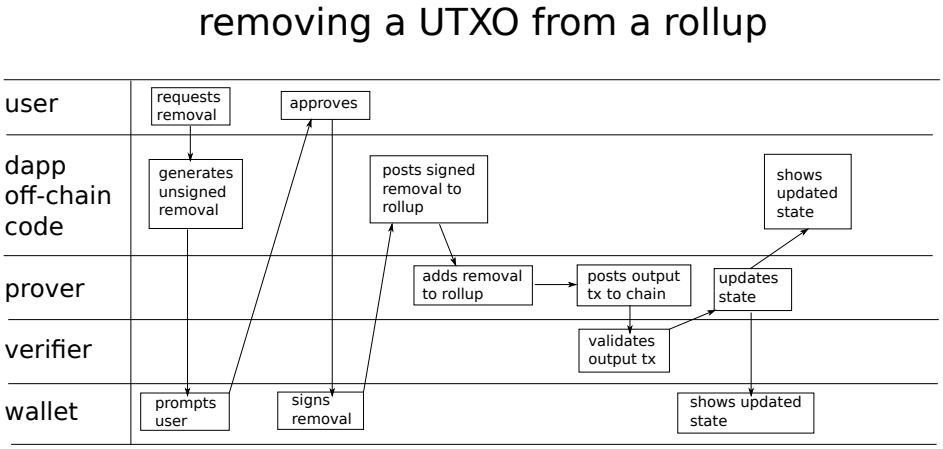

Figure 4: Process of removing an output from a rollup.

locks expire without the inputs being used, or some similar measure. The collateral can, for example, be one of the locked inputs. Every transaction will have some monetary input, at least to cover the transaction fee needed to generate the proof. In response to the locking request, the prover provides some sort of key that the dapp may use to consume the on-rollup inputs.

The next step is for the dapp to generate an unsigned transaction. The dapp sends the unsigned transaction to the user's wallet to be signed. The wallet signs the transaction with the user's approval. The dapp then receives the signed transaction from the wallet and posts it to the rollup by calling the prover API. The prover runs any validator scripts required and generates a proof. The transaction, its outputs, and the proof are added to the rollup, and the on-rollup inputs are marked as consumed. The dapp observes the rollup state changes via the prover API and makes them visible to the user.

3.2 Process of adding an input to a rollup

Adding an on-chain input to a rollup is essentially a two-step process. First, the user sends the input to the rollup contract address. Second, the input gets picked up by an input transaction. The input transaction, which is on-chain, also takes as input the rollup state UTXO. Once the transaction is authorized by the prover, it outputs the new rollup state UTXO. The rollup state UTXO contains the monetary values added to the rollup (in aggregate form, without tracking who owns what). It also contains whatever state data must be stored on-chain for the rollup contract.

3.3 Process of removing an output from a rollup

Removing an output from the rollup and putting it on the chain is similarly a two-step process. First, the user posts a removal transaction to the rollup. Second, the prover posts an output transaction to the chain and updates its internal state. An output transaction describes a set of output UTXOs and a proof (checked by the verifier contract) that shows the outputs as lawful. One output transaction on-chain settles many removal transactions on-rollup.

3.4 Authorization for updating the rollup

The rollup contract will accommodate two methods of authorizing a transaction. These transactions should only be performed by authorized parties to

prevent UTXO resource contention. The two methods of authorization are signing the transaction with a signing key (whose public key hash is stored in the rollup contract state) or spending an authorization token (described in Section 4).

4 Distribution and decentralization of the prover

Due to the high computational complexity of generating zkSNARK proofs to verify many transactions, the prover must be made a distributed system. By making use of recursive zkSNARK proofs, we can split the proof generation task into pieces that can then be farmed out to various computers.

This architecture style assumes it is possible to split the validation of a sequence of transactions $\vec{t}$ and the generation of the corresponding proof into parallel subtasks. This is true if the validation sequence can be split into two sequences $\vec{u}$ and $\vec{r}$, where some interpolation of the elements of $\vec{u}$ and $\vec{r}$ is equal to $\vec{t}$, and no transaction in $\vec{u}$ takes as input an output of a transaction in $\vec{r}$, and no transaction in $\vec{r}$ takes as input an output of a transaction in $\vec{u}$.

This setup is assumed to be possible given dapp designs that were built to do this exact thing. Thus, we assume that we can split a sequence $\vec{t}$ of transactions into as many subsequences as needed to distribute the proof generation tasks among many computers and, therefore, achieve sufficient throughput.

As per usual for a distributed system, it should be designed to be fault tolerant, so that if one computer in the system fails, the system as a whole will continue to function as intended without human intervention.

Distributing the prover is also a first step towards full decentralization of the prover. Full decentralization of the prover means no computer, individual, or organization exists as either a single point of failure or as a trusted entity within the prover protocol. Full decentralization of the prover is not within scope of the Orbis Project's initial release, but is an eventual goal and commitment of the project. We are designing for an initial release that features a smooth upgrade path, transparent from an end-user perspective, to a fully decentralized prover.

In the distributed prover, one computer is, at any given time, designated as the leader of the system, responsible for coordinating all prover activities.

It produces the final output transaction and posts it to the blockchain. In the centralized distributed prover, this leader is controlled and designated by Orbis Labs—or, more generally, the organization that controls the signing key(s) for the rollup contract instance. In the decentralized prover, the leader will be periodically determined by a leader election process, to be designed later.

In the centralized prover, the prover is authorized to update the state of the rollup contract instance by signing the transactions with the signing key(s), whose public key hashes are stored in the contract instance state. In the decentralized prover, another mechanism is needed, and the recommended mechanism is authorization tokens.

Authorization tokens allow their holders to perform certain actions within the system. In this application, authorization tokens allow holders to update the rollup contract instance in lieu of signing the transaction. The token holder provides the authorization token as an input to the transaction, and the transaction outputs it to an address. The token's output address is stored in the rollup contract instance state and can be set by a transaction signed with the authorized signing key(s). Once Orbis moves to full decentralization, the intention is, at some point, to freeze the token distribution protocol by removing all of the authorized signing keys from the rollup contract state, thus disabling further updates to the distribution protocol and heralding the start of the leader election protocol.

The mechanism to update the authorization token output address is intended to be part of the smooth upgrade path from a centralized prover to a decentralized prover. To be sufficiently transparent to end users, the upgrade path must not involve any changes to the rollup contract. Such changes would necessitate moving funds from one contract to the other, which cannot be done securely without manual intervention. We could include a provision in the rollup contract allowing funds to be transferred using the authorized signing key(s), but this would open a security vulnerability where the person(s) in control of the signing key(s) could make off with the funds. In the envisioned smooth upgrade path, we instead use a signed transaction with the rollup contract to set an authorization token's output to a contract address that handles the leader election process, ensuring that an authorization token passes to the next elected leader.

In the decentralized version of the prover, we must take additional security considerations into account. We can assume, in the centralized version, that prover nodes (i.e., the computers that constitute the computing cluster of

the prover) are not malicious. In the decentralized version, we cannot make this assumption. Thus, we will need to perform in-depth security analysis to guard against acts of malicious prover nodes. To the greatest extent possible, we should use zkSNARK proofs to ensure that such malicious acts do not occur. Where this will not work, we can use an enforcement mechanism that requires prover node operators to post collateral that will be burned if the nodes act maliciously. (This solution depends on designing reliable mechanisms for detecting the occurrence of such malicious acts.)

5 Money trap avoidance

One of the risks of ZK rollup solutions is denial of service (DoS). The prover *cannot* post any unlawful outputs to the chain, but it *can* refuse service. Given that the prover is the only entity able to sign input and output transactions, if it ceases operations, then the funds stored in the rollup contract would be lost. In a similar DoS case, the prover might refuse service to certain market participants, causing them to lose money they stored in the rollup contract. How can we mitigate these risks?

The Orbis protocol uses a model that gives one prover exclusive access to a given rollup contract UTXO. This design choice was made to avoid resource contention, which could cause performance issues and complex situations—such as the double-spending of input UTXOs. Such a situation would not result in actual double-spends on-chain, but it would prevent both provers from merging their on-rollup transaction outputs onto the chain.

This exclusive access model (in contrast to a permissionless model) leads to the aforementioned money trap risk that occurs when a prover stops operating. What if a prover stops operating and no other prover steps up or is authorized to update the rollup contract state? Decentralization mitigates this risk, but even in a decentralized protocol, there is no guarantee that somebody will volunteer to run an Orbis prover node.

To mitigate the money trap risk in the case of a lack of prover node services, Orbis will provide a backup mechanism for removing funds from the rollup while a prover is not operating. "A prover is not operating" is operationally defined as meaning that the rollup contract state UTXO has not been updated in a certain number of blocks. Under such conditions, the rollup contract allows for permissionless withdrawal transactions which contain a zkSNARK proving that the funds transferred from the rollup con-

tract to the recipient(s) belong to the addresses of the recipient(s). These "permissionless offline withdrawal transactions" allow people to recover their funds from the rollup contract without interacting with a prover.

This permissionless fail-safe withdrawal mechanism has some notable limitations. Firstly, it will allow for withdrawing funds held at a wallet address, but it will not allow for withdrawing funds locked in smart contracts, which would require a prover to be running. Secondly, it will be limited in throughput by the throughput of the chain. However, it should nonetheless allow people to eventually recover their funds helds on the rollup at their wallet addresses, as long as the Cardano chain is continuing to function, even if the prover goes offline and never comes back online.

Expiring exclusive access addresses the money trap risk when a prover stops operating. However, it does *not* address the money trap risk that occurs when a prover refuses service to certain market participants. The latter risk will need to be addressed in a different way. Our non-binding recommendation is to process input and output transactions on a first-come, first-served basis. This would mean that the first inputs sent to the contract address are the first added to the rollup, and the first removal transactions posted to the rollup are the first processed in an output transaction.

Our non-binding recommendation for implementing the first-come, first-served principle is to post transactions to a decentralized data structure, such as a queue, implemented as a decentralized, conflict-free replicated data type (CRDT). The principle would be enforced as part of the leader election protocol by burning the collateral of the leaders who violate it. It seems it would be possible to check for compliance via the zkSNARK proofs posted to the rollup verifier contract. However, that would require us to either build enforcement of the principle into the initial verifier contract release, or to update the verifier contract post launch. The latter would obscure the smooth decentralization upgrade path from end users.

Enforcing the first-come, first-served principle is beyond the scope of an initial release. For the initial release, we do not require a solution to the money trap risk associated with selective denial of service by a prover. This is deemed an acceptable risk upon initial release because Orbis Labs will, at first, be the only authorized prover for the rollup contract instance.

6 Fees

The fee structures for on-rollup and on-chain rollup transactions need to cover the costs of operating the system, as well as balance supply and demand to ensure availability of services. Defining the fee structures is beyond the scope of this document. For on-rollup transactions, charging fees that cover the operation costs of the rollup should be sufficient, since we should be able to scale to meet all demand for the on-rollup services. For on-chain rollup transactions, an upper limit exists to the throughput we can achieve pending further advances in the blockchain technology. Therefore, for on-chain rollup transactions, we will need to design a fee structure that includes "availability surchages" to ensure availability of the services. The availability surcharges will be computed off-chain and charged by the rollup contract. Periodically, those fees will be collected by an authorized transaction with the rollup contract, which will send the fees to a specified address (which can also be updated by another authorized transaction with the rollup contract).

7 Database

The Orbis architecture does not specify the nature of the database used to store the current rollup state and on-rollup transaction history. For the sake of transparency and fault tolerance, using a decentralized data store might be the best option. For the sake of performance, using a traditional database, such as PostgreSQL, might be the best option. It might make sense to use more than one database to address different considerations. It might make sense to store currently needed data in one database and archival data in another database.

8 Verifer contract state data

Here is a speculative, non-binding description of some state data that must be stored in the rollup UTXO.

1. A nonce that gets incremented on every output transaction. The purpose of the nonce is to help ensure on-rollup UTXOs are used only once.

2. A hash of all the on-rollup UTXOs. The mentioned hash may be a recursive hash, i.e., a hash of UTXOs and a previous hash. The purpose of the state hash is to serve as an input to the proof-checking algorithm.

3. The public key hashes of the authorized signing key(s).

4. The authorization token output address.

5. The most recent time at which the prover posted an input transaction.

6. The most recent time at which the prover posted an output transaction.

7. The amount of collected availability surcharges currently stored in the rollup contract.

8. The address to send collected availability surcharges to.

References

[1] Ethworks. *Zero-Knowledge Blockchain Scalability.* Ethworks Reports, 2020. https://ethworks.io/assets/download/zero-knowledge-blockchain-scaling-ethworks.pdf

[2] Visa. *Power your retail business beyond the point of sale.* Accessed December 17, 2021. https://usa.visa.com/run-your-business/small-business-tools/retail.html

[3] Will Martino, Monica Quaintance, and Stuart Popejoy. *Chainweb: A Proof-of-Work Parallel-Chain Architecture for Massive Throughput.* DRAFT v15. https://d31d887a-c1e0-47c2-aa51-c69f9f998b07.filesusr.com/ugd/86a16f_029c9991469e4565a7c334dd716345f4.pdf

[4] Maxwell Ogden, Karissa McKelvey, Mathias Buus Madsen, Code for Science. *Dat - Distributed Dataset Synchronization And Versioning.* May 2017 (last updated: Jan 2018). https://github.com/datprotocol/whitepaper/blob/master/dat-paper.pdf

[5] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System.* Accessed Dec 22, 2021. https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf

[6] The Electric Coin Company. *The halo2 Book.* 2021. https://zcash.github.io/halo2/index.html

[7] The Electric Coin Company. *halo2.* 2022. https://github.com/zcash/halo2

[8] Sean Bowe, Jack Grigg, and Daira Hopwood. *Recursive Proof Composition without a Trusted Setup.* IACR Cryptol. ePrint Arch., 2019, #1021. https://eprint.iacr.org/2019/1021