

# ZK Bridge - Milestone 2

Eryx

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>What does a bridge do?</b>	<b>4</b>
2.1	Trusted vs Trustless bridges . . . . .	4
<b>3</b>	<b>Why a ZK bridge?</b>	<b>5</b>
3.1	Increased security . . . . .	5
3.2	It saves space and time . . . . .	5
<b>4</b>	<b>Our Proposal - Asset Transfer over a ZK Bridge</b>	<b>6</b>
4.1	A possible architecture . . . . .	6
4.2	System overview . . . . .	8
<b>5</b>	<b>Proving systems analysis</b>	<b>10</b>
<b>6</b>	<b>Different approaches to implement the bridge</b>	<b>10</b>
6.1	Native Alternative . . . . .	11
6.1.1	Ouroboros . . . . .	11
6.1.2	Block header validation in Ouroboros Praos . . . . .	11
6.1.3	Contextual Data Required for the Proof . . . . .	12
6.2	Mithril . . . . .	12
6.2.1	Certificate generation . . . . .	13
6.2.2	Mithril in our ZK bridge . . . . .	13
6.2.3	Mithril to update Stake Distribution . . . . .	13
6.2.4	Mithril to prove the inclusion of transactions . . . . .	14
6.2.5	Is Mithril adding more optimism? . . . . .	14
6.3	Alternatives to write ZK proof . . . . .	14

6.3.1	Risc0 . . . . .	14
6.3.2	SP1 . . . . .	15
6.3.3	Circom . . . . .	16
6.3.4	Advantages and Disadvantages of the Alternatives . . .	17
6.4	Conclusion . . . . .	17
<b>7</b>	<b>Bridge architecture</b>	<b>18</b>
7.1	First step: a one-way Bridge . . . . .	18
7.2	Parts of the system . . . . .	19
7.2.1	Locking contract . . . . .	19
7.2.2	Minting contract . . . . .	19
7.2.3	Stake distribution contract . . . . .	20
7.3	Protocol overview . . . . .	21
<b>8</b>	<b>Impact of future changes in Ouroboros</b>	<b>22</b>
<b>A</b>	<b>Future Ouroboros Versions</b>	<b>23</b>
A.1	Ouroboros Peras . . . . .	23
A.2	Ouroboros Leios . . . . .	25

# 1 Introduction

The blockchain ecosystem has grown significantly in the past years, but it has not reached true mass adoption. One of the main reasons is the fragmentation of the ecosystem: today, there are numerous blockchains operating independently, each with its own infrastructure, rules, and user base. This is troublesome for users and developers and limits the overall potential of blockchains.

Interoperability has become a critical need. Among the different solutions that have emerged, blockchain bridges are a promising approach. A bridge enables the transfer of assets and data across otherwise disconnected chains, helping to unify the ecosystem. For Cardano, a bridge is particularly important, as it would allow the network to connect with other blockchains and attract greater liquidity.

However, many existing bridge designs rely on external trust assumptions—often depending on federations or third-party validators. Such models introduce centralization risks and undermine the security guarantees that blockchains aim to provide. To address this, **our work focuses on a token-transfer bridge that leverages zero-knowledge techniques**. By using ZK proofs, we eliminate the need for trusted intermediaries while also reducing on-chain computation costs. The bridge would be implemented on the Cardano side, designed to interact with any blockchain that is either isomorphic to Cardano or that implements the necessary contracts and ZK verification of valid headers from its chain.

The main challenge of the project is the generation of ZK circuits to prove the occurrence of certain events in Cardano. We will explore two possible approaches to do this:

1. **Native validation:** constructing a ZK circuit that proves the correctness of Cardano block header validation as performed by a Cardano node. This requires a deep understanding of the Ouroboros consensus protocol, particularly the rules involving block header validation.
2. **Mithril-based validation:** leveraging Mithril to produce a ZK proof that a given transaction is included in a snapshot validated under a given Cardano stake distribution.

In addition, we conduct an analysis of different proving systems, evaluating their strengths and weaknesses to determine the most suitable option

for this bridge.

## 2 What does a bridge do?

Each blockchain is a different distributed system that advances at its own pace. This means, among other things, that it doesn't have, by itself, the ability to know things from outside its own state. For that kind of information, alternative sources of external information are needed.

In particular, a "bridge" between chains  $\mathcal{A}$  and  $\mathcal{B}$  is a source of information of the state of chain  $\mathcal{A}$  that gets stored in chain  $\mathcal{B}$ , and vice versa. Given that we believe that information is legitimate, this allows each chain to act knowing that a certain event occurred in another chain, thus achieving interoperability between them.

There are different strategies or technologies to build trust that the provided information is in fact legitimate, and this opens the door to different kinds of bridges, depending on where that trust comes from.

### 2.1 Trusted vs Trustless bridges

In a **trusted** bridge there needs to be a party that centralizes the information about the events. This would be, for example, if you bridge tokens by sending them to said trusted party in chain  $\mathcal{A}$ , and then it sends you the corresponding ones in chain  $\mathcal{B}$ . Another possibility would be that you lock funds in  $\mathcal{A}$  and then wait for the trusted party to update an oracle in  $\mathcal{B}$  that allows you to mint tokens there.

In a **trustless** bridge, as its name suggests, there's no single party that holds all the power. This means there aren't extra security assumptions, i.e. it's as secure as the underlying blockchains. There are mainly two flavors for this. The first one is **optimistic** bridges, which rely on game theory incentives for the actors in the blockchains to behave honestly, like commissions for nodes that behave correctly and slashes (the removal of your funds) for the ones that don't. This type of bridge often includes consensus algorithms that converge to a common truth that all nodes accept, under the assumption that some big proportion of them act honestly. The **pessimistic** bridges, on the other hand, rely on a system of smart contracts that accept cryptographic proofs of the trigger events in order to unlock the actions in the other chain.

It's convenient to use SNARKS for this because they're succinct, so they fit blockchain limitations in the verification stage.

## 3 Why a ZK bridge?

A bridge whose trust is based on cryptography offers advantages over an optimistic one, both in security aspects and in performance.

### 3.1 Increased security

An optimistic bridge relies on the fact that a certain proportion of the users are honest. This means that if an attacker takes over a big enough proportion of the nodes (or it creates enough new corrupt nodes) they could take over the system. Some mechanisms based on slashing even require that agents monitor transactions to detect fraudulent ones, which opens the door to some transactions to go under the radar, and also requires redundancy of computation. A ZK proof, on the other hand, cannot be generated if the preconditions aren't met, regardless of the proportion of malicious agents.

### 3.2 It saves space and time

ZK proofs offer the possibility to delegate computation, in addition to enhancing privacy. This helps save time and space on-chain. Regarding to space, there are some data structures involved (like stake distributions and lists of transactions) that are too big to handle them on-chain. But you can prove knowledge of those structures by committing to them instead and using the commitments as inputs to the proof verification instead of the full data, reducing the size of the on-chain computation. The same principle applies to long or costly algorithms.

Also, the cryptographic proofs only need to be generated once, whereas consensus algorithms require that different nodes perform the same computation so that they can share their results and agree on them.

## 4 Our Proposal - Asset Transfer over a ZK Bridge

### 4.1 A possible architecture

We based part of our design on [xie2022zkbridge] (see in particular section 3.1), which discusses a similar product that has components that are network-agnostic and others that are focused on Ethereum and its particular challenges. Their architecture consists of two different layers: one that keeps an updated state of one chain in another, and the particular applications' layer on top of the first one. Let us assume that we want to build an application that requires information about the state of chain  $\mathcal{A}$  in chain  $\mathcal{B}$ . An example of this can be seen in figures 1 and 2, where the user locks funds in chain  $\mathcal{A}$  to be stored in  $\mathcal{B}$ .

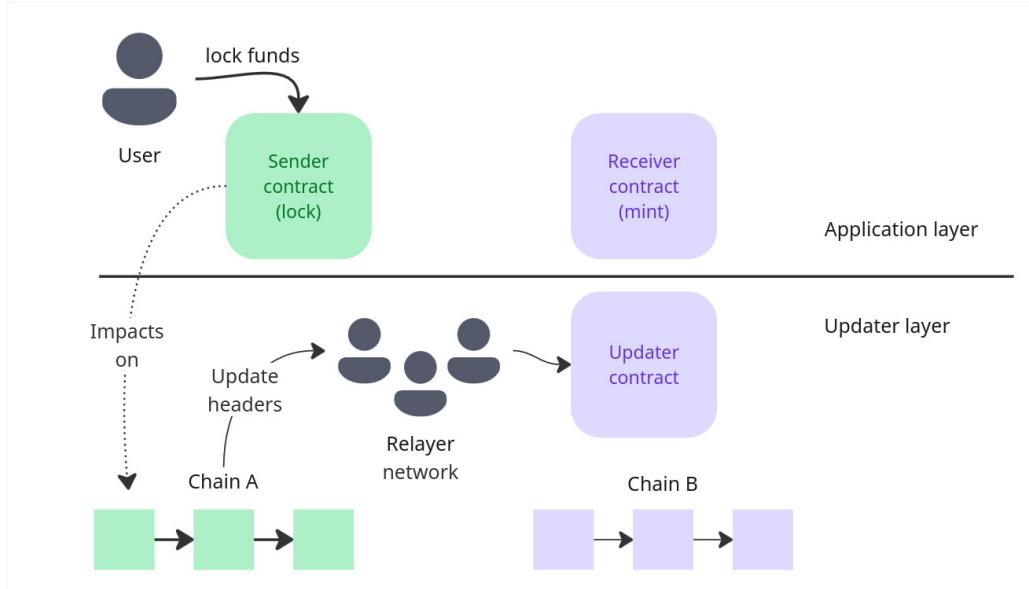


Figure 1: This shows the flow os the user locking funds and the actors participating in that action.

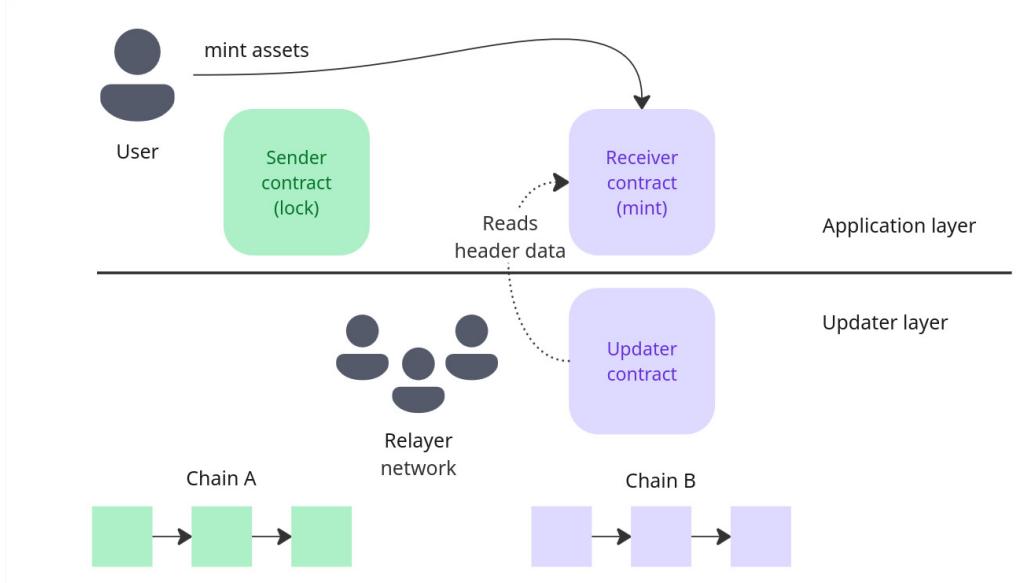


Figure 2: This shows the minting action of the bridge.

In this case, the updater layer consists of a single **updater contract** in  $\mathcal{B}$ , which is supported by a **relayer network** that interacts with it to keep it effectively updated. The updater contract keeps the headers of all blocks of the chain, because it would be impractical to store the whole chain and because the headers suffice for different use cases. The relayer network is a set of nodes that observe blockchain  $\mathcal{A}$  and produce proofs of correctness of new block headers to feed the updater contract in  $\mathcal{B}$  along with the headers themselves.

The application layer typically consists of a contract in each of the chains, which the paper calls the **sender and receiver contracts**. The sender contract in  $\mathcal{A}$  produces an effect that would trigger an action in  $\mathcal{B}$ . Then the underlying updater layer stores the headers in the contract in  $\mathcal{B}$ . In parallel, the application creates a cryptographic proof for the trigger action, which can be verified using said headers. This proof can then be verified by the receiver contract using the blockchain  $\mathcal{A}$ 's state (stored in  $\mathcal{B}$ 's updater contract) as auxiliary information.

While it is not necessary to keep these layers apart (even less so in a prototype), it is useful to know that they exist separately as they solve two

different aspects of building the bridge. We think that, as the exploration of bridging in Cardano advances, it is desirable to generate a generic updater layer that can support all kinds of applications, but at the same time it is too abstract to generate an updater layer without any applications to use it. We then prefer for the layers to grow apart more organically.

## 4.2 System overview

The particular application that we're building transfers assets from chain  $\mathcal{A}$  to chain  $\mathcal{B}$ .

Suppose we have two chains,  $\mathcal{A}$  and  $\mathcal{B}$ , and we want to transfer liquidity from one to the other. The objective is to mint a token  $T_b$  in chain  $\mathcal{B}$  using some quantity of another token  $T_a$  in chain  $\mathcal{A}$  as collateral. This is achieved by locking  $T_a$  in a smart contract, so it gets pulled out of circulation in  $\mathcal{A}$ . Then people can operate with the new  $T_b$  in chain  $\mathcal{B}$ . This new token has a value that is tied to  $T_a$  because, at any point, any user should be able to burn some amount of  $T_b$  in chain  $\mathcal{B}$  and, as a consequence, unlock a corresponding one of  $T_a$  from the locking contract in chain  $\mathcal{A}$ .

As we said before, the ability to know that an event happened in a chain in the context of another one is key for the bridge to work. Here this happens in two instances:

- When some funds are locked in chain  $\mathcal{A}$  and this allows minting in chain  $\mathcal{B}$ ..
- When some funds are burned in chain  $\mathcal{B}$  and this allows the unlocking of funds in  $\mathcal{A}$ .

Here, we can see that the full circuit of the bridge needs information flow in two directions: from  $\mathcal{A}$  to  $\mathcal{B}$  in locking-minting, and from  $\mathcal{B}$  to  $\mathcal{A}$  in burning-unlocking. In the first direction, the locking contract would be the sender, and the minting would be the receiver. In the second one, the burning would be the sender and the unlocking, the receiver.

Producing proofs for this is a very hard problem and there are different strategies to approach it. We will talk about some of these strategies later in this document. For now, let's assume that we have the ability to produce perfect certificates for these events when they happen, based on the information of the updater layer. Let's also assume that we have a way to prevent the same certificate from being used twice (there are easy ways to do that).

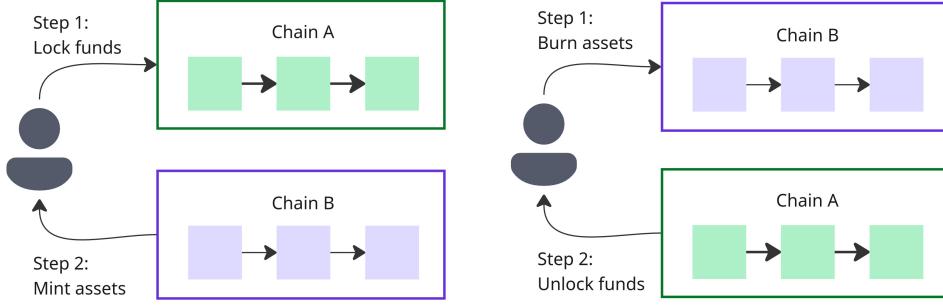


Figure 3: Interactions of a token transfer Bridge

Under these assumptions, this system can be interacted with in four ways:

1. Lock  $T_a$  in  $\mathcal{A}$  and receive a certificate of locking.
2. Give a certificate of locking in  $\mathcal{B}$  and mint  $T_b$ .
3. Burn  $T_b$  in  $\mathcal{B}$  and receive a certificate of burning.
4. Give a certificate of burning and unlock  $T_a$ .

Of these, (1) and (4) are interactions with contracts in chain  $\mathcal{A}$ , and (2) and (3), in chain  $\mathcal{B}$ . This requires building, a priori, a hybrid system with some contracts in each blockchain. As a proof of concept, we simplified the problem by making Cardano both chains  $\mathcal{A}$  and  $\mathcal{B}$ . This doesn't have a utility as a product by itself, but it allows us to work with only one stack of technologies. In addition to the value as a proof of concept, different parts of the bridge could be adapted to implement different sides of a working bridge between two different chains. Also, if the second chain is isomorphic to Cardano, the full circuit of the bridge could be used almost as-is.

The scope of this project is to begin the work by analyzing feasibility and implementing interactions (1) and (2), because this allows us to focus only on one of the directions of communication. Since we're doing a bridge between Cardano and itself, the updating part for the other direction will be identical, and the application, very similar.

## 5 Proving systems analysis

As described in [[aiken-zk-repo](#)], in which we carried an in-depth analysis of the different proving systems and their integration with Cardano for a past grant, the only widely used proving systems whose proofs fit into a Cardano transaction are Groth16 and Plonk.

However, we could also use a STARK-based prover as Risc0 or SP1, to generate the necessary proofs to carry out the bridge protocol, and then wrap those verifications into a Groth16 or Plonk circuit that will be verified on-chain. That way, we can fit STARK proofs into Cardano thanks to recursion.

“Groth16 and Plonk over BLS12-381 are currently the only suitable proving systems for on-chain verification that have battle-tested implementations. The main reason for this is the smaller proof size and verification keys of these proving systems. The most mature implementations of these proving systems are gnark, snarkjs and arkworks. In the future, other FRI-based off-chain provers could be leveraged with the help of recursion, for example by wrapping a STARK in a SNARK proof. However, as a first step Groth16 and Plonk proving systems are the main options.”

## 6 Different approaches to implement the bridge

In this section, we describe two alternatives we considered for implementing a ZK bridge for Cardano.

The main challenge in building a ZK bridge is being able to construct a ZK proof that demonstrates a block is valid and has been added to the source blockchain. In particular, we must prove that a block was successfully added to Cardano. During this research, we explored two possible ways to achieve this. One option is to build the proof natively, by demonstrating the execution of Cardano’s block header validation algorithm in Ouroboros. The other option is to leverage Mithril snapshots to construct the proof.

Below, we describe both alternatives along with their pros and cons.

## 6.1 Native Alternative

### 6.1.1 Ouroboros

Ouroboros is the consensus protocol used by Cardano. At present, Cardano operates under **Ouroboros Praos**, so this version of the protocol will be the focus of this research. In the future work section, we discuss Ouroboros Peras and Ouroboros Leios, and the changes the bridge would require once those versions are deployed in production.

To generate a ZK proof, it is necessary to understand the Ouroboros Praos algorithm that ensures a block is valid and was added by the correct slot leader.

### 6.1.2 Block header validation in Ouroboros Praos

The validation algorithm involves the following steps:

- Verify the VRF was correctly executed: Check the VRF proof and output, together with the block producer's VRF public key, and ensure that the output is indeed the result of executing the VRF with the proper input. The input consists of the epoch nonce agreed in the previous epoch and the slot number where the block was added. The VRF proof, output, public key, and block slot are all included in the block header, while the epoch nonce is derived each epoch from the blocks of the previous epoch.
- **Verify the slot leader election:** Using the VRF output and the relative stake of the stake pool (SP) that produced the block, check the inequality  $Evaluate(slot, epoch\_nonce) < \phi(stake)$  holds.
- **Verify the operational certificate of the stake pool:** This certificate confirms that the hot KES key used to sign the block is valid for the pool. The certificate signature and its sequence number must be verified (no certificate with a higher sequence number should have been presented).
- **Verify the KES signature validity:** Ensure that the KES key has not expired and that the block's KES signature is valid.

### 6.1.3 Contextual Data Required for the Proof

To build and verify a ZK proof showing that a Cardano block header is valid, the circuit must have access to a large amount of contextual information from the blockchain as public inputs.

First, the circuit must know the relative stake of the SP producing the block. In Cardano, the stake of an SP is the sum of all ADA owned by *stake addresses* that delegated to that SP through *delegation certificates* in the ledger. To compute this, one must review all delegation certificates in the blockchain, check the ADA balance of each *payment address* linked to the stake address in the certificate, and add the amounts from reward accounts associated with those stake addresses.

This means that the circuit needs to be provided with all this information in some form. And all these events should also be proven to the receiving chain. This process would need to be carried out once per epoch and would be a major challenge to implement the bridge.

Second, the circuit must also prove the correct *epoch nonce* for each epoch. This nonce, generated from the previous epoch, is used by stake pools as input to the VRF that determines slot leader election. It must be included as a public input to the circuit, and to ensure soundness, we must guarantee that the provided nonce is correct.

## 6.2 Mithril

Mithril[[mithrildocs](#)] is a cryptographic protocol in Cardano that produces efficient proofs of the blockchain state. Instead of requiring every participant to process the full chain, Mithril allows the aggregation of signatures from stake pool operators, weighted according to their stake, to generate compact certificates that can be quickly verified. This design makes it suitable for light client protocols, where users with limited storage or computing resources can still validate the integrity of the ledger without depending on a trusted third party. By relying on stake as the foundation of its security, Mithril extends the role of proof-of-stake beyond block production, providing a mechanism for faster synchronization and trust in Cardano’s decentralized infrastructure.

### **6.2.1 Certificate generation**

In Mithril, the core mechanism for enabling efficient verification is the generation of certificates (often referred to as snapshots). These certificates are created periodically to represent a trusted checkpoint of the blockchain state. The process works by having stake pool operators produce cryptographic signatures over a snapshot of the ledger at a given point in time. Since the security of Mithril is tied to Cardano’s proof-of-stake model, the weight of each signature corresponds to the amount of stake controlled by the signer. Once a threshold of stake is reached, the individual signatures can be aggregated into a single signature to form a compact certificate.

This certificate acts as a proof that a sufficient portion of the stake in the network has attested to the validity of that snapshot. Light clients or new nodes can then download only the latest snapshot and verify it against the Mithril certificate, rather than replaying the entire history of transactions. As a result, synchronization becomes significantly faster while still maintaining the security guarantees derived from the stake distribution.

### **6.2.2 Mithril in our ZK bridge**

These certificates will come in handy in two ways. We deep dive into each one of them, following:

### **6.2.3 Mithril to update Stake Distribution**

To verify certificates, blocks, or even epochs, we need the current state of the Stake Distribution. That is, who are the Stake Pool Operators and how much stake they are delegated. This Distribution is fixed during a single epoch, so it should only be updated with each new epoch to keep track of the PoS consensus layer of Cardano and for the bridge to be able to verify the inclusion of locking transactions.

Given this challenge, Mithril would enable us to cheaply update the Stake Distribution of each epoch by issuing a specific type of certificate, whose message is a commitment of the new Stake Distribution. That is, the current Pool Operators agree on which is going to be the next Stake Distribution and sign a certificate of that. This way, the Stake Distribution of epoch  $n$  ensures the Stake Distribution of epoch  $n+1$ , allowing us to keep validating the state of Cardano through the epochs.

#### 6.2.4 Mithril to prove the inclusion of transactions

Mithril also has an API that, given a transaction hash, retrieves a certificate that contains (1) a Merkle Root of the set of transactions up to some point of Cardano and (2) a Merkle Proof that our transaction is included in that set. The verification of these certificates would let us generate inclusion proofs of any kind of transaction. Therefore, the receiver blockchain could verify, given a Mithril certificate and the Stake Distribution of that epoch, that a certain locking transaction was included in a valid block.

#### 6.2.5 Is Mithril adding more optimism?

Since the set of Mithril signers is included in the set of Ouroboros' SPOs, this solution doesn't add any trust assumptions beyond the inherent trust of the Cardano consensus protocol on the Pool Operators. Instead of trust, it adds more responsibilities to the SPOs: the responsibility of generating and signing Mithril certificates. That's why we think that utilizing Mithril doesn't compromise the trustless condition of our bridge.

### 6.3 Alternatives to write ZK proof

In the next sections we will talk about some frameworks that were analyzed to write our ZK proofs and some advantages and disadvantages of each of them.

#### 6.3.1 Risc0

Risc0 [[risczerodocs](#)] is a tool that allows the execution of Rust programs and generates a ZK proof of that execution using a zkVM. The output is a STARK proof, which can be recursively wrapped into a SNARK proof. This enables the generation of a ZK proof of Rust code execution that can be verified on Cardano using a proving system such as Groth16.

This means that we can write a Rust program that runs all the verifications we need to prove in our bridge and then run that program using Risc0 to get our ZK proof.

This is used by other blockchains for block validation. Risc0 reported 44 seconds to validate an Ethereum block. They say [[risc0-blog-realtimeproving](#)] that their upcoming ~\$120K proving cluster will prove their Ethereum proving times to ~9.25s.

In case we opt for the Native approach, where we'd need to prove that a Cardano block header is valid, we could code a Rust program that implements the block header validation logic used by Cardano nodes. Then we could execute the program using Risc0 to obtain a ZK proof of the verification process.

For this purpose, one could leverage Amaru [[amaru](#)], a Rust implementation of a Cardano node. While the project is still a work in progress, it is under active development and could be an actual implementation of a Cardano node in the near future. The code for block header validation is complete and could serve as the basis for our objective.

In case we opt for the Mithril approach, we could directly use Mithril's code, which is already written in Rust, to perform all needed validations and get our ZK proof automatically.

However, Risc0's performance degrades significantly when handling elliptic curve operations such as those involved in some parts of the block header verification algorithm as VRF proof verification or BLS signatures. We didn't perform an exhaustive benchmark yet but this kind of operations takes several minutes in general. Since these operations are required to validate a Cardano block or a Mithril certificate, producing such proofs for actual blocks would likely take a long time.

Nevertheless, Risc0 is under active development and could be considered if its proof generation times improve.

### 6.3.2 SP1

Similar to Risc0, SP1 [[SP1docs](#), [SP1assessmentreport](#)] is a zkVM that runs Rust code compiled to RV32IM (which is an extension of RISC-V with 32-bit words and integer multiply) and builds a ZK proof of its execution. It also produces a STARK, that can be wrapped in a SNARK like Groth16.

Applied to Cardano header validation, SP1 also allows reusing Amaru or Mithril components, with all the advantages this carries. Performance depends on using precompiles for cryptographically heavy tasks such as hash functions, elliptic-curve evaluation, and pairing-based operations. Without using precompiles, the time for proving grows impractically.

If no precompiles are used, SP1 suffers in performance in the same order of magnitude as Risc0.

However, if SP1 precompiles are used for the Amaru implementations of Cardano cryptographic primitives (as for example, their VRF, KES algo-

rithms) or Mithril, it would reduce the time for proving them. Experiments conducted by Succint [[riscv-video](#)] have shown that the proof generation time can be reduced by at least two orders of magnitude for some cryptographic primitives like KECCAK256 or SHA256 . This is an advantage for the zkVM path viability for what we try to achieve. However, this is subject to Amaru’s or Mithril’s implementations. We will do a deeper research of this in the milestone 4 of this project when we will build the needed ZK proofs.

### 6.3.3 Circom

Circom [[circom](#)] is a domain-specific language designed for constructing arithmetic circuits that serve as the basis for ZK proofs. Instead of writing proofs directly, you can define computations as circuits composed of constraints, and these constraints enforce the logical and mathematical rules that the computation must satisfy. Circom provides an abstraction layer that simplifies the process of translating high-level logic into the low-level algebraic representation required by modern ZK-SNARK and ZK-STARK proving systems. Afterwards, this algebraic representation is consumed by a proving backend (such as snarkjs [[snarkjs](#)], which is the most used one) to carry out the cryptographic parts of the zero-knowledge pipeline such as the trusted setup, the witness generation and the proving itself.

In case we use Circom, it will be necessary to do more detailed work, where we must implement a circuit that proves either the validation of a Mithril certificate or the validation of a Cardano header in Ouroboros Praos.

The main benefit of this approach would be that writing a dedicated circuit would enable us to fully optimize the different aspects of the proof, such as proving time, memory usage, or key handling, to mention a few. We can make modifications to the circuit to focus on improving one or many of these aspects in order to make the bridge more efficient. Also, the possibility of changing and experimenting with different proving systems would be beneficial in choosing the framework that best fits for this project.

Additionally, our team has already worked with Circom in previous productive environments, such as a grant funded [[poi-project](#)] in Catalyst Fund13. This ensures low friction for Circom implementations since we already have the expertise.

### 6.3.4 Advantages and Disadvantages of the Alternatives

The approach of using tools such as Risc0 or SP1 to automatically generate proofs from code (for example, Amaru’s implementation) offers a significant advantage in terms of maintainability. When Cardano’s consensus algorithm evolves (with new versions of Ouroboros currently in development), it would not be necessary to reimplement the circuit. A potential “disadvantage” could be the dependency on an external project such as Amaru; however, this limitation would no longer apply once Amaru reaches a stable cycle of releases. This same advantage holds for the Mithril approach, where we wouldn’t need to reimplement our circuits if Mithril implementation changes.

In contrast, the option of implementing a circuit manually using Circom carries the drawback of having to reimplement it each time the consensus algorithm changes (like when future versions of Ouroboros are released) or if Mithril’s signature scheme changes.

On the other hand, solutions like Risc0 or SP1 face performance challenges when executing complex operations such as verifying signatures or VRF proofs. Meanwhile, a hand-crafted and optimized implementation in Circom could achieve lower proof generation times.

There are alternatives we can use to attack the performance issue like using precompiles. Precompiles can swap tens of thousands of RISC-V emulation cycles for wide and specialized tables. This approach massively reduces trace size, the number of constraints, and the latency in the proof generation process (which can be reduced by at least two orders of magnitude).

Besides, a witness in the Circom circuit is produced for the whole constraint system, meaning we cannot naturally split the circuit into independently provable chunks unless designing additional aggregation/recursion circuits. A zkVM usually can separate a long program in fragments or chunks of some millions of cycles with built-in recursive composition, allowing to prove many segments in parallel across cores or computers (like a cluster or cloud computing power) and fold those generating STARK proofs into one proof with a recursion tree (which is optimized as, for example, SP1 uses a recursion VM different from their other zkVM).

## 6.4 Conclusion

After our research on both approaches to implement a ZK bridge, we think that using Mithril certificates is the best option to prove both Cardano’s

stake distribution and the inclusion of locking transactions on a receiving chain.

As said in section 6.1.3, the option of natively generating a proof by proving the execution of the Ouroboros Praos block-header validation algorithm faces a major challenge: proving the stake of a Cardano stake pool. This would require proving all stake-delegation transactions on Cardano—and not only those, but also how many ADA each stake address holds and how many rewards the reward accounts associated with those addresses contain. This would force us to prove all Cardano transactions in each epoch. This is substantially more expensive than simply validating a Mithril stake-distribution certificate per epoch and then validating another certificate for each locking transaction.

As for the alternatives evaluated to write a ZK proof, we have not reached a conclusion yet. Both Risc0 and SP1 are advancing very quickly and it is possible that they will have optimizations in the coming months that will improve their performance in what we need. We believe the best approach is to wait for milestone 5 of the project to evaluate these alternatives more extensively, with more information, and see which one best suits our needs. There is also a chance that tools like Risc0 or SP1 Our plan is trying to get an implementation of the circuit using Risc0 or SP1, but if performance becomes an issue, we will switch to a lower level approach like Circom.

## 7 Bridge architecture

In this section, we will describe the bridge protocol, that will use Mithril certificates to prove the inclusion of a locking transaction in a source chain and mint the corresponding tokens in the destination. We will describe how the bridge protocol would work when Cardano acts as both the source and the destination chain. This allows us to implement the full prototype in Cardano to both receive and send funds. In the case of implementing a bridge with other blockchain in the future, the corresponding contracts would need to be deployed on the other chain to enable token transfers with Cardano.

### 7.1 First step: a one-way Bridge

One clarification we want to make is that this project only contemplates a one-way bridge, in which the funds locked to be minted in another chain

cannot be unlocked. In the future, we could enable this by presenting proof that some tokens received on the destination chain were burned. The cost of this work would depend on the specifications of the counterpart chain used, which would determine the required flow for creating and verifying the proof of burning.

## 7.2 Parts of the system

Here we describe each contract in the system to be implemented in more detail.

### 7.2.1 Locking contract

By sending funds to this contract, a user locks them so that their counterpart can later be minted on the destination chain. The corresponding Aiken validator should guarantee that these funds remain locked on Cardano and cannot be unlocked. This is because we are implementing a one-way bridge. If we wanted a full bridge, unlocking would be possible by presenting a burning certificate from the other chain. Note that certificates could have a larger size than Cardano transactions permit. That's why we will need to use commitments to work with certificates on chain.

The locking transaction must also identify the target chain and the bridge being used to ensure that the locked funds cannot be minted on two different chains, or twice in the same chain using two bridges. In addition, it must include the address to which the funds should be minted on the destination chain to prevent proof-stealing.

### 7.2.2 Minting contract

In this contract, a proof of inclusion of a locking transaction in Cardano must be presented in order to mint the equivalent tokens. Since we are prototyping in Cardano, we will also describe the minting phase from Cardano's perspective.

To mint, we must, on one hand, receive a proof of inclusion of the locking transaction on the source chain, ensuring that the target chain of that transaction is indeed the destination chain and the bridge is the minting contract's bridge. Once that proof is verified, the equivalent tokens can be issued on the destination chain.

But that is not all. Even if all the information is correct, we have to forbid double-spending by using the same proof twice in the same bridge. To achieve this, we need to keep a record of already-used locking transactions for the minting contract to verify that the locking transaction has not been previously used. This list can grow too large, so we can keep a commitment to it instead (in our case, a Merkle commit) to prevent the registry from taking up more space as the bridge continues to be used.

How would this work in Cardano? The token to be minted should have a minting policy that requires the locking transaction with its inclusion proof as input. It must also receive a proof that the locking transaction has not already been used, and additionally enforce that the registry of used locking transactions is updated to include this transaction. In other words, the token minting operation and the update of the already-used locking transactions registry must execute atomically (either both succeed or neither does).

As a final point, since we will use Mithril certificate validity proofs as inclusion proofs for locking transactions, the minting contract must also be provided with a valid certificate of the stake distribution for the epoch corresponding to the locking transaction. To avoid repeating that certificate for every transaction in the same epoch, we add another contract to our protocol to store these certificates: the **stake distribution update contract**.

### 7.2.3 Stake distribution contract

This contract is updated once per Cardano epoch and must receive a validity proof of a Mithril stake distribution certificate. The contract must verify that the new certificate is valid and that a certificate already exists for the previous epoch. Note that this would require the deployment of a first certificate that won't have a previous stake distribution on chain. This would need to be addressed separately in the bootstrapping of the bridge.

This contract will be used by the Minting contract as an oracle, providing the stake distribution for the required epoch.

### 7.3 Protocol overview

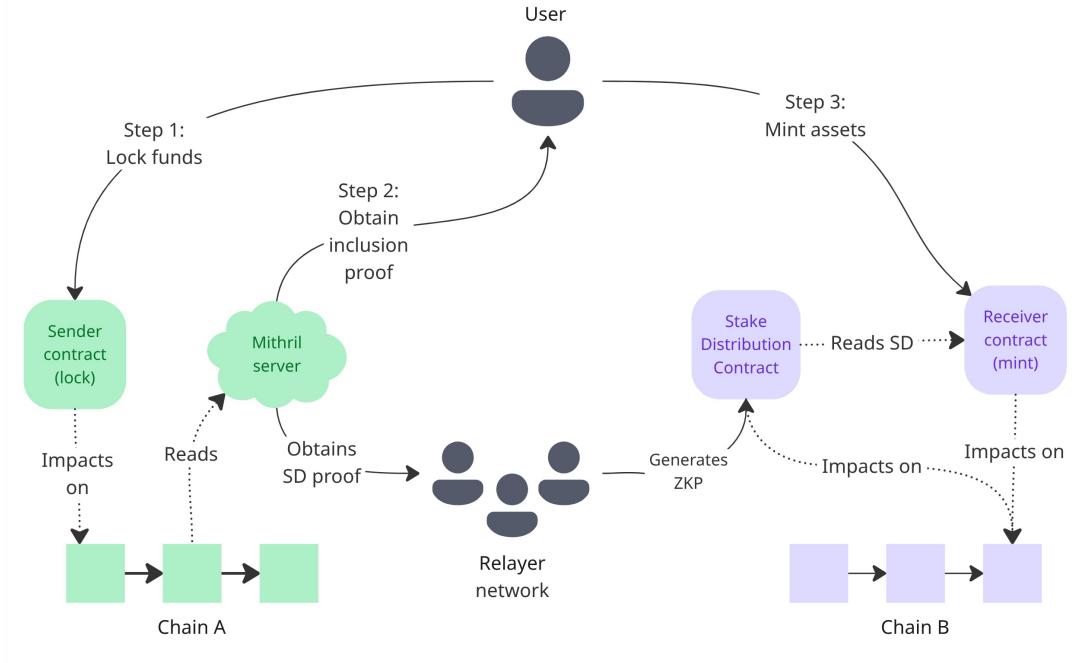


Figure 4: Protocol overview

The protocol begins with a locking action initiated by a user. In this action, the user interacts with the locking contract to send a certain amount of ADA to another chain, specifying the destination chain, the bridge being used and the address to which the funds should be sent.

At the same time, a relayer will continuously monitor the Cardano blockchain to retrieve the Mithril stake distribution certificate for each epoch, and generate a ZK proof of its validity, submitting it to the blockchain via the stake distribution contract. The stake distribution validator will verify de ZK proof using the relevant public inputs such as the new certificate and a reference to the previous certificate stored on-chain.

After locking funds, the user must request Mithril a certificate of inclusion of the locking transaction and generate a ZK proof of its validity. Once obtained, the user must also retrieve the registry of already-used locking transactions and generate a proof of its update with the new one. This

proof must also show that the locking transaction was not previously in the registry.

Once both proofs are obtained, the minting contract is used to mint the equivalent tokens of the locking transaction on the destination chain, to the address specified by the transaction. The minting contract will verify the inclusion proofs (using the stake distribution verified by the stake distribution contract) and that the transaction registry has been correctly updated.

## 8 Impact of future changes in Ouroboros

Part of the time for this milestone was dedicated to analyzing different aspects of Ouroboros and how these could impact our bridge. On one hand, we investigated the current version of the protocol, Ouroboros Praos, which has already been mentioned in previous sections of this document. On the other hand, we also studied the two upcoming changes to the protocol: Ouroboros Peras and Ouroboros Leios. An appendix provides detailed research on both protocols.

These future changes would have had a significant impact on our bridge if we had chosen the alternative of implementing ZK proofs of the execution of the Ouroboros header validation algorithm, since we would likely have had to reimplement the ZK circuit to adapt to those changes.

However, since we opted for the alternative of using Mithril, these changes are abstracted away by that tool and do not directly affect our bridge, as our circuits will validate Mithril certificates rather than anything directly related to Ouroboros.

We also investigated the current state of the Mithril project and upcoming changes. Of particular interest to this project is the ongoing research on using SNARKs to reduce the size of certificates and the verification time of signatures. Over the coming months, we plan to be in contact with the team working on this and closely follow their progress.

## A Future Ouroboros Versions

### A.1 Ouroboros Peras

Ouroboros Peras introduces a significant change in the chain selection rule and in how fast finality is achieved in Cardano. This is an optimistic protocol, based on the fact that the longest fork in Cardano history is of length 3. While in earlier versions of the protocol (Praos and Genesis) transaction stability depended only on the persistence parameter  $k$  (which is currently  $k = 1260$  and together with the active slot coefficient parameter  $f = \frac{1}{20}$  means one transaction can only be considered as stable after about 12 hours), Peras adds an additional mechanism based on boosting via voting certificates that reduces the expected settlement time to approximately two minutes, for adversaries controlling less than 25% of the stake. Boosting literally applies a “boost” to the chain’s weight, resulting in “locking in” the chain: including permanently (except with small probability) all transactions up to that point. The protocol divides each epoch in rounds, whose length is measured by a parameter  $U \ll k$ , and it potentially boosts one block in each round. The recommended value of  $U$  is 90 slots.

A transaction will be considered confirmed with overwhelmingly high probability once it is included in a block that has reached a quorum of more than  $\tau = 75\%$  of its committee. Blocks can be selected for voting only after  $L = 30$  slots have passed, so any previous block will be followed by at least  $L$  slots.

In practice, most proofs within the ZK bridge will be the same as in Praos or Genesis because they rely on the standard  $k$ -block confirmation. The support for Ouroboros Peras requires extending the definition of valid chain to include weight and certificate-based boosting, while maintaining backward compatibility with existing flows. The additional logic is only needed when certificates are present, and according to Cardano official documentation, certificates are not frequently included in the blockchain and the common case remains unchanged. The impact is incremental because we are extending verification with additional conditions. The main ZK bridge architecture remains without modifications.

## Chain Selection Rule

Before Ouroboros Peras, the proof had to demonstrate that a transaction is included in the longest or densest chain past the  $k$ -block window. With Peras, the valid chain is determined by a weighted chain rule (i.e., a chain can prevail before  $k$  blocks if it contains boosted blocks backed with certificates). The proof must extend the verification logic in the ZK proof to handle a new chain weight function, given by:

$$\text{weight}(\mathcal{C}) = \text{length}(\mathcal{C}) + B \times \text{certificates}(\mathcal{C}),$$

where  $B$  is Peras' certificate boost parameter,  $\text{length}(\mathcal{C})$  is the number of blocks in chain  $\mathcal{C}$ , and  $\text{certificates}(\mathcal{C})$  is the number of blocks in  $\mathcal{C}$  that have a certificate. From the perspective of the ZK bridge, the inclusion proof for a transaction must include this new weight function. The recommended value of  $B$  is 15 blocks.

When there is a voting process for a block, this chain selection rule is used by honest voters to decide which chain's block they should choose for their vote.

## Block Certificates and Votes

Certificates themselves appear as an optional field `block_certificate` in the block body; the proof should establish their validity whenever they appear, since they are not mandatorily included in the blockchain. This requires showing that the certificate corresponds to an already completed round, that it references a valid block of that round, and that the aggregated voting proof is sound.

For the implementation of Ouroboros Peras, the traditional model of aggregated signatures (like BLS Signatures) is replaced in Peras by a new primitive ALBA (Approximate Lower Bound Argument). The new ALBA primitive is not an aggregated signature but a proof that the set of valid votes has at least the required size. KES keys are still used to sign the votes; therefore, SPOs do not need new keys for using this protocol. If there are  $n$  votes in a round (which is also a parameter), it takes  $O(n)$  time to construct the ALBA, but its size remains polylogarithmic in  $n$  and the verification time is  $O(\log n)$ .

This implies that the circuit must:

- recompute the `round_nonce` as  $H(\text{epoch\_nonce} \parallel \text{"peras"} \parallel \text{round\_number})$  and validate it,
- verify the uniqueness of the votes,
- check the correctness of the votes using their KES signatures and their CDDL structure,
- check the correctness of the ALBA,
- confirm that the quorum threshold  $\tau$  was achieved.

From the January 2025 certificates analytics, Peras also introduces persistent and non-persistent voters, similar to the model later described in Ouroboros Leios. Persistent voters are selected once per epoch, while non-persistent voters are selected for each round, allowing the `block_certificate` size to be reduced. This is achieved using the Fait Accompli scheme from committee selection when applied to the stake distribution. The certificate records the set of voters, proof of their eligibility, and the quorum achieved (remember the `voting_proof` parameter in the `vote` CDDL). When validating the certificate, we are checking the proof of their eligibility.

There is a cooldown phase triggered when quorum is not reached, for the chain to heal, achieve quality, and attain a common prefix. Since it has no explicit on-chain representation, proofs do not need to model cooldown state and can simply validate the information actually stored in the blocks.

## A.2 Ouroboros Leios

This Ouroboros version is still in development, so the concepts included in this appendix are subject to future changes as the discussion about Leios evolves among the community.

Ouroboros Praos faces scalability limitations: its parameters cannot be tuned to increase throughput, which results in congestion during periods of high demand. These situations are commonly referred to as *protocol bursts*, and they happen when the transaction load grows abruptly for a short time. For example, when someone wants to distribute an NFT between lots of people. Praos is also difficult to scale against an *equivocation block attack*, where an adversary floods the network with invalid blocks, forcing nodes to provide large amounts of evidence to prove invalidity, wasting computational

power and resources. Ouroboros Leios is a protocol that addresses these challenges.

Leios introduces a new architecture that can be applied to any secure low-throughput PoS protocol. When applied to Ouroboros Praos, the result is Ouroboros Leios, although in practice both protocols coexist, with Praos remaining the primary Cardano consensus mechanism. Leios follows the honest-majority MPC paradigm, is optimistic, and is based on voting. The protocol introduces two distinct block types and a new system of votes and certificates (for something different than Peras).

## Ranking Blocks and Endorsement Blocks

In Ouroboros Leios, blocks are divided into Ranking Blocks (RBs) and Endorsement Blocks (EBs). RBs play the role of conventional Praos blocks, maintaining a similar size and serving as the structural part of the chain. An RB may additionally announce an EB in its header, which is `announced_eb` referencing the EB's block hash, references its size `announced_eb_size`, indicates with a bit whether an EB has been certified (i.e., `certified_eb`), and includes the corresponding certificate `eb_certificate` in the block body.

EBs are created by the same stake pool that produced the RB announcing them and always appear in the same slot as that RB. Their purpose is to carry transactions that could not fit into RBs, in order to increase throughput for Cardano if needed. An EB does not include the raw transactions themselves but only a list `transactions_references` consisting, for each transaction, of its hash `tx_hash` and its size `tx_size`. Since the ledger state evolves according to the sequential application of these transactions, their order is meaningful. Between the moment an EB is announced and the moment it is integrated into the ledger (by another RB, not necessarily created by the same node), there is a voting process where nodes validate its transactions; once quorum  $\tau$  is reached, a certificate is formed and eventually included in an RB. As of today, it remains unclear if the full transactions in EBs will be fully stored in the chain in later RBs.

## Votes and Certificates

There are two categories of voters in Ouroboros Leios: persistent and non-persistent. Persistent voters are selected once per epoch by a stake-weighted sortition algorithm, allowing them to be identified in a more compact way.

Their votes include an election identifier `election_id`, a persistent voter identifier `persistent_voter_id` identifying the stake pool for this epoch, an `endorser_block_hash` (the hash of the RB header that announced the EB), and a vote signature `vote_signature`.

Non-persistent voters are selected independently for each EB, again by a stake-weighted sortition algorithm. Their votes include a stake pool identifier `pool_id` and an eligibility signature `eligibility_signature`, instead of the `persistent_voter_id`.

During a vote, the EB is sent to all eligible voters. It is possible that not all nodes have every transaction whose hash is included in their mempool. Therefore, they may validate the EB against the transaction hashes it references and may use another dedicated sub-protocol, `LeiosFetch`, to retrieve missing transactions. This is subject to changes since CIP-0164, which is still in progress and receiving commits weekly.

From these votes, nodes can construct a certificate. Nodes can aggregate valid votes they receive until the quorum threshold  $\tau$  is reached, at which point they construct a certificate that will be attached to an RB. This certificate confirms the validity of the EB's transactions and causes the ledger state to evolve by applying them in order. Certificates are designed to be under 10KB and to keep RBs within a target size of roughly 100KB. Each `leios_certificate` contains `election_id` and `endorser_block_hash` with the same definition as EBs, the persistent voters list `persistent_voters`, a mapping of non-persistent voters to their BLS signatures `nonpersistent_voters`, and the aggregated signatures over all votes `aggregate_vote_sig`. The specification uses BLS as a reference, but explicitly states that the protocol can use any efficient aggregated signature scheme or succinct commitment scheme that Cardano may adopt. Like Ouroboros Peras, it would be possible to implement these certificates using ALBAs.

## Forks

For fork resolution, Leios does not alter the chain selection rule: it continues to rely on the Genesis definition of the densest chain (considering only RBs, since EBs influence the ledger state but do not affect which chain is selected). This makes Leios composable with the changes introduced by Ouroboros Peras.

From the perspective of the ZK bridge, Ouroboros Leios introduces significant new considerations. Proof of inclusion must account not only for RBs

(Praos blocks) but also for the existence and validity of EBs, their certificates, and the validity and correctness of the associated voting process. The ZK circuit must be able to verify certificates produced by the chosen scheme (BLS, ALBA, etc.). Since transaction persistence relies on a storage layer that is still evolving, it is reasonable to consider an integration between the bridge and Cardano off-chain storage protocols to guarantee that referenced transactions remain verifiable. The chain rule selection is the same, so the logic for fork resolution can be reused without major changes. The overall impact is not disruptive, but expands the set of requirements that must be considered in the ZK bridge implementation.

## License

 This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.