Intmax2: A ZK-rollup with Minimal Onchain Data and Computation Costs Featuring Decentralized Aggregators

Erik Rybakken[1], Leona Hioki[1], and Mario Yaksetig[2]

[1] Intmax
paper@intmax.io
[2] University of Porto

Abstract. We present a novel stateless zero-knowledge rollup (ZK-rollup) protocol with client-side validation called Intmax2. Our architecture distinctly diverges from existing ZK-rollup approaches since essentially all of the data availability and computational costs are shifted to the client-side as opposed to imposing heavy computational requirements on the rollup aggregators. Moreover, the data storage and computation in our approach is parallelizable for each user. Therefore, there are no specific nodes to validate the contents of transactions. In effect, only block producers, who periodically submit a Merkle tree root containing all the transactions, are necessary.

Keywords: Zero-Knowledge Proofs · Stateless ZK-Rollup · Blockchain Scaling

1 Introduction

As the blockchain ecosystem continually evolves, so does the urgency for scalable solutions that preserve security, reduce transaction costs, and improve overall throughput. Layer 2 technologies, particularly rollups, have emerged as pivotal tools to overcome these challenges, and have thus gathered substantial attention. Among these, Zero-Knowledge rollups (or ZK-rollups) have shown great promise due to their unique capability to bundle numerous transactions into a single proof that can be verified quickly and cheaply on-chain. Existing ZK-rollups, while innovative in their own respect, require all necessary data for updating users' balances to be posted on the underlying Layer 1 (L1) blockchain. This data, in a worst-case scenario, includes the transaction sender, the index of the token, the amount, and the recipient for each transaction, leading to a significant load on the underlying blockchain.

1.1 Data Availability

A fundamental bottleneck for blockchains is what is known as data availability. Data availability means that transaction data needs to be available in order to be able to prove the current state, such as account balances, of the blockchain. This is a problem for both Layer 1 blockchains and rollups. Layer 1 blockchains usually achieve data availability by requiring that all transaction data is publicly available for a node to consider the blockchain valid. Rollups usually achieve data availability by requiring that all transaction data is posted to the underlying blockchain (e.g. using calldata or blob data on Ethereum). Due to the fact that this data needs to be replicated among a large set of nodes, there is a limit on how much data can be made available, given the upper limit on the number of transactions per second that the blockchain can support. While for smart contract blockchains it might be necessary to provide the complete transaction data, it turns out that for simple payment transactions it is only necessary to make available a commitment to the set of transactions in a block (such as a Merkle tree root), together with a set of senders who have signed their transaction in the block. Our rollup design, called Intmax2, uses this fact to achieve increased throughput compared to existing alternatives. In addition, the design allows having a permissionless set of block builders that can build blocks in parallel, without needing to coordinate with each other. This allows for a very simple design.

1.2 Our Design

We divide our design into four parts.

First, we consider the main idea our design is based upon. Namely, the idea that given a partial set of transactions on a blockchain and the set of senders in each block, a user can compute a lower bound on their balance. We note that this idea can be traced back to some existing Plasma designs.

Second, we consider the proving of individual balances. More concretely, we show how users can prove their balance by leveraging commitments to a set of transactions (e.g., Merkle root) and the associated inclusion proofs. For simplicity, we first show an example with an explicit proof which contains a large set of transactions, and then describe the use of recursive zero-knowledge proof schemes [12,13,17], which results in more efficient approach. This part also describes how senders must send a zero knowledge proof of sufficient funds to the corresponding recipient.

Third, we describe the existing block types (i.e., transfer, deposit, and registration) and introduce the use of signatures that support aggregation (e.g., BLS). In this part, we cover the block structure of each of these block types and how we leverage the aggregation feature of the signature scheme.

Last, we describe the rollup contract and how different functionalities work. Namely, how to add blocks, how to deposit funds, and how to perform the corresponding withdrawals. In this part, we highlight that this process is completely open and, therefore, decentralized. As a result, any system entity is able to perform these actions.

We believe that this represents a proper breakdown of the protocol and allows for a clean and incremental description of our design.

1.3 Our Contributions

Intmax2 is an efficient and stateless design that:

- Only posts the index[3] of the transaction senders, as well as a merkle tree root and an aggregated signature of it for each batch on L1.
- Each transaction contains many token transfers to many recipients while having a constant 4-6 bytes calldata consumption.
- Shifts the computational requirements from the aggregator to the client.
- Offers permissionless aggregators.
- Provides stronger privacy properties than traditional ZK-rollups.
- Is highly parallelizable to a large number of users.

2 Preliminaries

2.1 Zero-knowledge proofs

Zero-knowledge proofs, introduced in [9], allow a prover $\mathcal{P}$ to prove to a verifier $\mathcal{V}$ a relation between a statement x and a witness w. A non-interactive zero-knowledge (NIZK) proof is a trio of algorithms:

- $\mathsf{Setup}(\lambda) \rightarrow pp$. For a certain security parameter λ, the setup algorithm outputs pp, the public parameters of the system.
- $\mathsf{Prove}(pp, x, w) \rightarrow P$. Given the system's public parameters pp, a statement x, and a witness w, issue a proof P.
- $\mathsf{Verify}(pp, x, P) \rightarrow \mathsf{Accept/Reject}$. Upon receiving the public parameters pp, the public statement x and the proof P, the verifier $\mathcal{V}$ either accepts or rejects the proof depending on whether or not P is well-formed. In this case well-formed implies the successful proof of the relation between the statement x and the witness w.

Properties. A zero-knowledge proof scheme is considered sound if an adversary $\mathcal{A}$ attempting to prove the statement without knowing the secret witness w cannot produce a valid proof with probability greater than 2^{-k} for knowledge error k. A zero knowledge proof scheme is considered complete if there is a guarantee that if the prover and verifier are honest, then the verifier successfully accepts a proof that shows that the prover $\mathcal{P}$ knows the witness w. Additionally, a proof P is considered a proof-of-knowledge if the prover $\mathcal{P}$ must know the witness w to compute the proof for the pair (x, w), and such proof-of-knowledge is considered zero knowledge if the proof P reveals nothing about the witness w. Additionally, if the scheme produces succinct arguments, then it is a (zk)SNARK [3,4,7,10,14]. Quantum-secure similar constructions exist, as in [1,5].

[3]This can be as low as 4-6 bytes depending on the number of users in the system.

2.2 BLS Signatures

The BLS signature scheme [2] operates in a prime order group and supports signature aggregation. The scheme uses a bilinear pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$. This pairing is efficiently computable, non-degenerate, and all the three groups have prime order q. We assume g_0 to be the generator of group $\mathbb{G}_0$ and g_1 to be the generator of group $\mathbb{G}_1$. Moreover, this signature scheme uses a hash function $\mathcal{H} : \mathcal{M} \to \mathbb{G}_0$. The scheme is defined by the trio of algorithms:

- $\mathsf{KeyGen}(\lambda) \to (sk, pk)$. The secret key is a random value $\mathsf{sk} \xleftarrow{R} \mathbb{Z}_q$ and the public key is $pk \leftarrow g_1^{\mathsf{sk}} \in \mathbb{G}_1$
- $\mathsf{Sign}(sk, m) \to \sigma$. The signature is a group element $\sigma \leftarrow \mathcal{H}_0(m)^{\mathsf{sk}} \in \mathbb{G}_0$.
- $\mathsf{Verify}(pk, m, \sigma) \to \mathsf{Accept/Reject}$. If $e(g_1, \sigma) = e(pk, \mathcal{H}(m))$ output accept, otherwise output reject.

Signature Aggregation. Given triples (pk_i, m_i, σ_i) for $i = 1, ..., n$, where n is the number of signers, anyone can aggregate signatures $\sigma_1, \ldots, \sigma_n \in \mathbb{G}_0$.

Rogue public-key attack. This signature aggregation method is, however, insecure due to an attack where and adversary $\mathcal{A}$ registers a maliciously crafted public key that then allows for the adversary to claim that an unsuspecting user, Bob, also signed a specific message.

3 Local computation of balances

In this section we will describe how to compute the balance of an account from partial transaction data. We let

$$Accounts$$

be a set of accounts, and

$$(Amounts, +, \leq)$$

be a partially ordered abelian group of amounts. For simplicity, the reader can just assume

$$Amounts = \mathbb{Z},$$

which gives a system that only supports one token. In practice, however, we would like to support multiple tokens, which can be done by letting *Amounts* be the set of functions

$$f : Tokens \to \mathbb{Z},$$

where *Tokens* is a set of tokens, the group operation is element-wise addition and where for functions $f, g : Tokens \to \mathbb{Z}$ we have $f \leq g$ if and only if $f(t) \leq g(t), \forall t \in Tokens$.

Definition 1 *A* transaction[4] *is a function*

$$t : Recipients(t) \to Amounts^+$$

from a subset $Recipients(t) \subset Accounts$ of accounts called the recipients *of the transaction, to the set $Amounts^+ \subset Amounts$ of all positive amounts, i.e. the set of all $amount \in Amounts$ where $amount \geq 0$.*

This definition of a transaction allows a sender to send funds to an unlimited number of recipients in a single transaction. A transaction does not include a sender. Instead, we will aggregate transactions together in *transaction maps*, which map senders to transactions.

Definition 2 *A* transaction map *is a function*

$$T : Senders(T) \to \mathcal{T}$$

where $Senders(T) \subset Accounts$ is the set of senders in the transaction map and $\mathcal{T}$ is the set of all transactions.

Notice that this definition enforces that each sender can only send one transaction in each transaction map, but this is not an issue, since each transaction can have an unlimited number of recipients.
We now define some utility functions for getting the total amount sent and received by an account in a transaction map.

Definition 3 *Let T be a transaction map. For each account $a \in Accounts$ we define*

$$sent(T, a) = \begin{cases} \sum_{r \in Recipients(T(a))} T(a)(r), & \textit{if } a \in Senders(T) \\ 0, \textit{otherwise} \end{cases}$$

and

$$received(T, a) = \sum_{\substack{s \in Senders(T), \\ a \in Recipients(T(s))}} T(s)(a).$$

Given a sequence of transaction maps, we can compute the balance of each account at each index of the sequence by applying the transactions in each transaction map. This balance will be called the *global balance* since it is computed from a complete set of transactions, in contrast with the *local balance* which we will define later which is based upon a partial set of transactions. We will require that every transaction in the transaction map sequence is valid, meaning that the sender of each transaction has sufficient balance for sending the transaction. To be able to have positive balances, we will designate a

[4]In practice, transactions should also contain a nonce to prevent replay attacks. For the sake of simplicity we will not deal with this issue here, but we note that it can easily be added.

source $\in$ *Accounts* which is always allowed to send a transaction even if it has insufficient balance. In the final rollup design, this account will be used to represent deposits from the underlying blockchain to the rollup. We will compute global balances as follows.

Definition 4 *Let $T_* = (T_i)_{i=1}^N$ be a sequence of transaction maps. We will say that the sequence is* valid *if for all $0 \leq i \leq N$ and $a \in Accounts \backslash \{source\}$ we have*

$$\sum_{j=1}^{i} received(T_j, account) - \sum_{j=1}^{i} sent(T_j, account) \geq 0.$$

If T_* is valid, we will call the above sum the global balance of account a in T_* at index i*, denoted by*

$$Bal_i(T_*, a).$$

Now, if the sequence T_* of transaction maps is publicly known by everyone, every user can compute the global balance of any account using the above definition. Suppose, however, that each user only knows some of the transactions in the transaction maps, i.e. a sequence of transaction maps $T'_* = (T'_i)_{i=1}^N$, where each T'_i is a restriction of T_i. The user would then want to compute a local balance of their account from the transactions in T'_*. The method of computing local balances should have both safety and liveness properties. Here, the safety property means that the local balance can never be greater than the global balance, and the liveness property means that honest user will be able to send transactions to each other and compute their balances from them. It turns out that to satisfy safety, we also need to use the set of senders $S_i = Senders(T_i)$ in each global transaction map when computing local balances. Otherwise, a malicious user could just omit transactions that they have sent in order to compute a local balance which is greater than their global balance. The local balance will be defined similarly to the global balance, except that we will also require that for each index i, if an account in the sender set S_i is missing from $Senders(T'_i)$, they will no longer be able to have any transactions in T'_*.

Definition 5 *Let $S_* = (S_i)_{i=1}^N$ be a sequence of account sets called* sender sets*, and let $T_* = (T_i)_{i=1}^N$ be a sequence of transaction maps such that $Senders(T_i) \subset S_i$ for all i. We will say that the sequence T_* is* valid *with respect to the sender sets S_* if it is itself valid, and if for all $a \in Accounts$ and $0 \leq i \leq j \leq N$ we have*

$$a \in S_i \backslash Senders(T_i) \Rightarrow a \notin Senders(T_j).$$

If T_* is valid with respect to S_*, we will define the local balance as

$$Bal_i(S_*, T_*, a) = \begin{cases} Bal_i(T_*, a), & \text{if } a \notin S_j \backslash Senders(T_j) \text{ for all } j \leq i \\ 0, \text{otherwise} \end{cases}.$$

The local balance is a generalisation of the global balance, where the two of them agree in the special case where for each index i, all accounts in the sender set S_i have a transaction in the transaction map T_i.
We will show two basic facts about local and global balances. The first is the fact that the local balance will increase or stay the same if we include more transactions in the computation, and that the local balance of an account is a lower bound on the corresponding global balance, where the two balances are equal in the case where all previous transactions sent by the account has been included in the local balance computation.

Proposition 1 *Let $S_* = (S_i)_{i=1}^N$ be a sequence of sender sets, and let $T_* = (T_i)_{i=1}^N$ and $T'_* = (T'_i)_{i=1}^N$ be sequences of transaction maps which are valid with respect to S_*, where each T'_i is a restriction of T_i. Then we have*

$$Bal_i(S_*, T'_*, a) \leq Bal_i(S_*, T_*, a) \leq Bal_i(T_*, a)$$

for all i and $a \in Accounts$. Also, if

$$a \notin S_j \backslash Senders(T_j)$$

for all $j \leq i$, the second inequality becomes an equality

$$Bal_i(S_*, T_*, a) = Bal_i(T_*, a).$$

Proof. The second inequality and the corresponding equality in the special case clearly follows from the definition of the local balance. We will now show the first inequality. We only need to verify the case where $a \notin S_i \backslash Senders(T_i)$ for all $j \leq i$, because otherwise we have $Bal_i(S_*, T_*, a) = 0$, which always satisfies the inequality. If $a \notin S_j \backslash Senders(T_j)$ for some $j \leq i$, we also have $a \notin S_j \backslash Senders(T'_j)$, since T'_j is a restriction of T_j. This means that

$$Bal_i(S_*, T_*, a) = Bal_i(T_*, a) = \sum_{j=1}^{i} received(T_j, a) - \sum_{j=1}^{i} sent(T_j, a)$$

and

$$Bal_i(S_*, T'_*, a) = Bal_i(T'_*, a) = \sum_{j=1}^{i} received(T'_j, a) - \sum_{j=1}^{i} sent(T'_j, a).$$

Now, since each T'_j is a restriction of T_j we have

$$received(T'_j, a) \leq received(T_j, a)$$

and

$$sent(T'_j, a) = sent(T_j, a)$$

for all $j \leq i$, so the inequality follows.

The second fact is that if we have two transaction map sequences which are valid with respect to a sequence of sender sets, and if they agree on their common intersections, we can combine them to form a combined sequence of transaction maps which is still valid with respect to the sender sets.

Proposition 2 *Let $S_* = (S_i)_{i=1}^N$ be a sequence of account sets and let $T_* = (T_i)_{i=1}^N$ be a sequence of transaction maps such that*

$$Senders(T_i) \subset S_i$$

for all i. Furthermore let

$$T_*^1 = (T_i^1)_{i=1}^N$$

and

$$T_*^2 = (T_i^2)_{i=1}^N$$

be sequences of transaction maps that are valid with respect to S_* and where for all i we have that T_i^1 and T_i^2 are restrictions of T_i with

$$Senders(T_i^1) \cup Senders(T_i^2) = Senders(T_i).$$

Then we have that the combined sequence of transaction maps T_* is valid with respect to S_*.

Proof. We will first verify the condition that for all $a \in Accounts$ and $0 \leq i \leq j \leq N$ we have

$$a \in S_i \backslash Senders(T_i) \Rightarrow a \notin Senders(T_j).$$

This follows, since we have

$$\begin{aligned} a \in S_i \backslash Senders(T_i) &\Rightarrow a \in S_i \backslash Senders(T_i^1) \text{ and } a \in S_i \backslash Senders(T_i^1) \\ &\Rightarrow a \notin Senders(T_j^1) \text{ and } a \notin Senders(T_j^2) \\ &\Rightarrow a \notin Senders(T_j). \end{aligned}$$

It remains to show that T_* is valid, i.e. that for all $0 \leq i \leq N$ and $a \in Accounts \backslash \{source\}$ we have

$$\sum_{j=1}^{i} received(T_j, a) - \sum_{j=1}^{i} sent(T_j, a) \geq 0.$$

Since T_j^1 and T_j^2 are restrictions of T_j for all j, we have

$$received(T_j, a) \geq received(T_j^1, a)$$

and

$$received(T_j, a) \geq received(T_j^2, a)$$

for all j. Also, since T_*^1 and T_*^2 are both valid with respect to S_*, we have either

$$\sum_{j=1}^{i} sent(T_j, a) = \sum_{j=1}^{i} sent(T_j^1, a)$$

for all j or

$$\sum_{j=1}^{i} sent(T_j, a) = \sum_{j=1}^{i} sent(T_j^2, a)$$

for all j. It follows that either

$$\begin{aligned} &\sum_{j=1}^{i} received(T_j, a) - \sum_{j=1}^{i} sent(T_j, a) \\ \geq &\sum_{j=1}^{i} received(T_j^1, a) - \sum_{j=1}^{i} sent(T_j^1, a) \\ \geq &\ 0 \end{aligned}$$

or

$$\begin{aligned} &\sum_{j=1}^{i} received(T_j, a) - \sum_{j=1}^{i} sent(T_j, a) \\ \geq &\sum_{j=1}^{i} received(T_j^2, a) - \sum_{j=1}^{i} sent(T_j^2, a) \\ \geq &\ 0. \end{aligned}$$

4 Balance proofs

In order for a user to be able to prove their balance in a sequence of transaction maps $T_* = (T_i)_{i=1}^N$ from a sequence of restricted transaction maps, they need a way to prove that the transactions in the restricted transaction maps are actually in T_*. In order to do this, we will use commitments to the transaction maps in T_*.

4.1 Transaction map commitments

For committing to a transaction map, we will assume a commitment scheme that takes a transaction map T and returns a commitment $C \in \mathcal{C}$, where $\mathcal{C}$ is a set of possible commitments, and inclusion proofs $\pi(s) \in \Pi$ for every sender $s \in Senders(T)$, where Π is the set of possible inclusion proofs. The commitment scheme has a verifying function, which takes a transaction, a sender, a commitment and an inclusion proof of the transaction by the sender in the committed transaction map, and returns *true* if the inclusion proof is valid

and *false* otherwise. The commitment scheme should have the *binding property* that it should be computationally infeasible to construct valid inclusion proofs of two different transactions by the same sender in a transaction map. One way to realize such a scheme is to construct a merkle tree from a transaction map T where the merkle tree stores the hash of the transaction $T(s)$ of each sender $s \in Senders(T)$ at the leaf whose merkle path is determined by s. The commitment of T will be the merkle root of this merkle tree.[5][6]
We will now construct a simple block sequence where each block consists of a commitment to a transaction map together with the set of senders who have signed the commitment.

Definition 6 *A* simple block *is a tuple* (S, C)*, where* $S \subset Accounts$ *and* $C \in \mathcal{C}$*. We denote by*

$$\mathcal{B}_{simple}$$

the set of all simple blocks.

In practice, we also need to verify that the senders in a block has actually signed the commitment in the block. This will be done in Section 5. For now, we will just assume that it is always the case that all senders in a simple block have signed the commitment in the block.
Signing the transaction map commitment serves two purposes. First, it serves as a confirmation that the sender intended to send their transaction in the committed transaction map. Second, it provides a way to prevent data withholding attacks by the actor who constructs the transaction map commitments, since honest users will not sign the commitment of a transaction map unless they know an inclusion proof of their transaction in it.

4.2 Explicit balance proofs

A simple way to prove the balance of an account in a simple block sequence B_* is to provide a sequence of transaction maps T_* that is valid with respect to the sender sets in B_*, together with inclusion proofs that these transactions are in B_*. Then, the balances in B_* are proven to be at least the local balances in T_*. We will call these proofs *explicit balance proofs.*

Definition 7 (Explicit balance proof) *Let* $B_* = ((S_i, C_i))_{i=1}^N$ *be a simple block sequence. An* explicit balance proof *for* B_* *is a tuple*

$$p = (T_*, \pi_*(*))$$

where

$$T_* = (T_i)_{i=1}^N$$

[5]Note that it is possible to construct an invalid merkle tree, where some of its leaves are not valid transactions. This would not pose an issue, since the binding property still holds in this case.

[6]We note that KZG commitments can also be used instead of merkle trees.

is a sequence of transaction maps which is valid with respect to S_*, and

$$\pi_*(*) = \big(\pi_i(s)\big)_{1\leq i\leq N,\ s\in Senders(T_i)}$$

is an indexed family of inclusion proofs where $\pi_i(s)$ is a valid inclusion proof of transaction $T_i(s)$ by sender s in the commitment C_i for all $1 \leq i \leq N$ and $s \in Senders(T_i)$. For each index i and account $\in Accounts$, we say that p is an explicit proof *of the balance $Bal_i(S_*, T_*, account)$ of the account at index i in B_*, and we say that p is an* explicit validity proof *of the transactions in T_*.*

Explicit balance proofs can be combined as follows.

Definition 8 *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence, and let*

$$p^1 = (T_*^1, \pi_*^1(*))$$

and

$$p^2 = (T_*^2, \pi_*^2(*))$$

be two explicit balance proofs for B_* that we know of. Assuming the binding property of the commitment scheme, we have that for each i, the transaction maps T_i^1 and T_i^2 agree on their common intersection. Then, we will define the combination of p^1 and p^2 *to be the explicit balance proof*

$$p = (T_*, \pi_*(*)),$$

where each T_i is the unique extension of T_i^1 and T_i^2 to the union $Senders(T_i^1) \cup Senders(T_i^2)$, and the inclusion proofs $\pi_*(*)$ are formed by taking the inclusion proofs for the transactions in T_*^1 and T_*^2, where we will take the first inclusion proof in the case where we have two different inclusion proofs of the same transaction[7]. We have that the combined explicit balance proof p is actually an explicit balance proof, since by Proposition 2 the merged sequence of transaction maps is still valid with respect to S_*.

4.3 Completing transactions

In order for the recipients of a transaction to be able to add the received funds to their balances, the transaction sender must complete the transaction by sending to each recipient a validity proof of the transaction. This proof can simply be an explicit validity proof, but in practice we will instead use an inclusion proof of the transaction in the simple block sequence together with a ZK-proof *that an explicit validity proof of the transaction exists.* In details, we define the completed transactions as follows.

[7]This might be possible, depending on the chosen commitment scheme, since we do not require that there is at most one inclusion proof of a transaction in a committed transaction map.

Definition 9 *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence. We define the* completed transactions *in B_* to be the sequence of transaction maps $T_* = (T_i)_{i=1}^N$ consisting of all transactions where both the sender and the recipients of the transaction have a validity proof of the transaction in B_*. The binding property of the commitment scheme implies that this sequence of transaction maps is well defined. If the validity proofs used are explicit validity proofs, we call T_* the* explicitly completed transactions*, and if ZK-proofs are used, we call T_* the* ZK completed transactions*.*

4.4 Completing transactions with explicit validity proofs

We will first consider the case where explicit validity proofs are used to complete transactions. Suppose a user wants to complete a new transaction they have sent in the simple block sequence B_*. By definition, the user has explicit validity proofs for all completed transactions they have received in B_*. These proofs can then be combined to form a single explicit validity proof

$$p = (T_*, \pi_*(*))$$

for the completed transactions received by the user. Now, suppose the following two conditions are met, which we call the *preconditions for completing a transaction*:

1. The user knows an inclusion proof of the transaction in B_*
2. The sequence of transaction maps T'_* obtained by adding the new transaction to T_* is still valid with respect to the sender sets in B_*

Then, the user will get a new explicit validity proof

$$p' = (T'_*, \pi'_*(*)),$$

where $\pi'_*(*)$ is obtained by adding the new inclusion proof to $\pi_*(*)$, which proves the validity of the new transaction. This proof can then be sent to each recipient to complete the transaction.
The first precondition will be satisfied if the user never signs a transaction map commitment unless they have an inclusion proof of their transaction in it. The second precondition amounts to the conditions that

$$\sum_{j=1}^{i} received(T_j, account) - \sum_{j=1}^{i} sent(T_j, account) \geq 0,$$

meaning that the user had sufficient balance for the new transaction, and

$$account \notin S_j \backslash Senders(T_j)$$

for all $j \leq i$, meaning that all previous transactions sent by the user are already included in T_*. To satisfy this precondition, the user can verify that

they have sufficient balance for the new transaction before signing the transaction map commitment.
To summarise, a transaction sender should only sign a transaction map commitment if the following conditions are met, which we call the *protocol rule.*

1. The sender knows an inclusion proof of their new transaction
2. The sender has no pending transactions, meaning that the simple block sequence currently contains all transaction map commitments that the user has signed.[8]
3. The sender knows that they have sufficient funds for the new transaction, meaning that the sum of all completed transactions in B_* they have received *minus* the sum of all transactions they have sent in B_* exceeds the amount sent in the new transaction.

We now prove that the protocol rule guarantees that the preconditions for completing a transaction is met.

Proposition 3 *Let $B_* = ((S_i, C_i))_{i=1}^N$ be the current simple block sequence, and suppose a user has always followed the protocol rule.*[9] *Then the preconditions for completing the user's transactions are met.*

Proof. The first precondition is met since the user will have inclusion proofs for all of their transactions. To see that the second condition is met, observe that there will be no transactions sent by a in B_* in the time period from when the commitment of the transaction map that includes the transaction was signed by the sender, and the time when the new commitment is included in the simple block sequence. This means that if T_* are the completed transactions in B_* and T'_* is the transaction map sequence consisting of every transaction the user has sent in B_*, the amount

$$\sum_{j=1}^{i} received(T_j, a) - \sum_{j=1}^{i} sent(T'_j, a)$$

can only increase while the new transaction is waiting to be included. Since this amount was sufficient for the new transaction at the time the commitment was signed, it will still be sufficient by the time the new commitment is included in B_*.

Theorem 1. *Let B_* be a simple block sequence and let T_* be the explicitly completed transactions in B_*. If a user has always followed the protocol rule, they will be able to explicitly complete all of the transactions they have sent in B_*, as well as generating an explicit proof that the balance of their $account \in Accounts$ at index i in B_* is at least $Bal_i(T_*, account)$.*

[8]This condition can be relaxed to requiring that the account has sufficient balance for *all* of its pending transactions

[9]We must also suppose, since we have omitted transaction nonces for simplicity, that there hasn't been any replay attacks, meaning that all commitments in B_* are unique.

Proof. The user can construct an explicit balance proof

$$(T'_*, \pi_*(*))$$

by combining the explicit validity proofs of all transactions they have received in T_*, which they have by the definition of T_*, as well as all transactions they have sent in B_*. This combined explicit balance proof proves that their account has sufficient balance for all of their transactions. If the user then completes all of their transactions, the combined explicit balance proof will prove the balance of the user's $account \in Account$ at index i in B_* to be at least $Bal_i(T_*, account)$.

4.5 Zero knowledge balance proofs for a simple block sequence

The explicit method of proving balances has several drawbacks. First, it requires each user to store a large amount of transactions and inclusion proofs. Second, it is computationally expensive to verify an explicit balance proof, since it requires computing the result of applying all the transactions in the proof. Third, it is not private, since the sender of a transaction needs to give their whole history of transactions to the recipient. All these drawbacks will be solved by replacing the explicit proof of the balance of an account with a ZK-proof that such an explicit proof exists.
In order to avoid having to know the complete simple block sequence when verifying the ZK-proof, we will instead use the *hash* of the simple block sequence, which we define as follows.

Definition 10 *Suppose we have a hash function*

$$Hash : \{0,1\}^* \to \{0,1\}^n$$

and an encoding of simple blocks

$$enc : \mathcal{B}_{simple} \to \{0,1\}^*.$$

Let $B_* = (B_i)_{i=1}^N$ *be a simple block sequence. For each index* $0 \leq i \leq N$ *we define the* hash of B_* at index i, *written* $H_i(B_*)$ *inductively as follows. For* $i = 0$ *we define*

$$H_0(B_*) = Hash([0]),$$

and for $1 \leq i \leq N$ *we define*

$$H_i(B_*) = Hash(H_{i-1}(B_*)||Hash(enc(B_i))).$$

The following circuit will be used to prove an account's balance in a simple block sequence.

Circuit 1 Circuit for verifying the balance of an account in a simple block sequence

```
Public input:
    a hash $h$
    $account \in Accounts$
    $balance \in Amounts$
Private input:
    a simple block $B = (S, C)$
    a hash $prev\_h$
    an amount $prev\_balance \in Amounts$
    a transaction map $T$
    inclusion proofs $\pi(s)$ for every $s \in Senders(T)$
    for every $s \in Senders(T) \backslash \{source\}$ a ZK-proof $P(s)$ that the balance of $s$ in the
    simple block sequence with hash $prev\_h$ is at least $amount(T(s))$
    a ZK-proof $P_{prev\_balance}$ that the balance of $account$ at $prev\_h$ is at least
    $prev\_balance$
 1: if $h = Hash([0])$ then
 2:     Verify $balance = 0$
 3: else
 4:     Verify $h = Hash(prev\_h || Hash(enc(B))$
 5:     Verify $P_{prev\_balance}$
 6:     Verify the inclusion proofs $\pi(*)$
 7:     Verify the ZK-proofs $P(*)$ of sufficient balance
 8:     Verify that $account \notin S \backslash Senders(T)$
 9:     Verify that $balance \leq prev\_balance - sent(T, account) + received(T, account)$
10: end if
```

Note that the circuit only verifies that the provided balance is *less than or equal to* the computed balance. This allows for greater privacy, since a user only needs to provide a proof that they have a sufficient balance for a transaction without revealing their whole balance.

Definition 11 *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence, let account $\in$ Accounts, let balance $\in$ Amounts and let $1 \leq i \leq N$. A* ZK-proof *that account has at least balance in B_* at index i is a ZK-proof that Circuit 1 can be satisfied with the public inputs $h = H_i(B_*)$, account and balance. A* ZK validity proof *of a transaction in B_* consists of a valid inclusion proof of the transaction in B_* together with a ZK-proof that the sender had sufficient balance for the transaction in B_* at the index immediately preceding the index of the simple block containing the transaction.*

Theorem 2. *Let B_* be a simple block sequence and let T_* be the ZK completed transactions in B_*. Then, if a user have always followed the protocol rule, they will be able to ZK complete all of their transactions in B_*, as well as generate ZK-proofs that the balance of their account $a \in Accounts$ is at least $Bal_i(T_*, a)$ for each index i.*

Proof. Let T'_* be the result of adding the transactions already sent by the user's account in B_* to T_*. Since the user has always followed the protocol rule, we have that T'_* is valid with respect to the sender sets in B_*. The user will then construct, for all indices i, a ZK-proof

$$P_i$$

that their account $a \in \mathit{Accounts}$ has the balance $\mathit{Bal}_i(T'_*, a)$ at index i in B_* inductively as follows.

Base case: $i = 0$ The user can trivially generate a ZK proof P_0 that circuit 1 is satisfied by the public inputs $h = \mathit{Hash}([0])$, $\mathit{account} = a$, and $\mathit{balance} = 0$, no matter the values of the private inputs. Since $\mathit{Bal}_0(T'_*, a) = 0$ by definition, this ZK-proof proves that the balance of a at index 0 is at least $\mathit{Bal}_0(T'_*, a)$.

Inductive clause: $1 \leq i \leq N$**) and the user has constructed** P_{i-1} The induction hypothesis states that the user has constructed a ZK proof P_{i-1} that the balance of a in T'_* at index $i-1$ is at least $\mathit{Bal}_{i-1}(T'_*, a)$. We also know, by the definition of T_*, inclusion proofs $\pi_i(*)$ of the transactions in T'_i and ZK-proofs of sufficient balance $P_{i-1}(s)$ for every $s \in \mathit{Senders}(T'_i) \backslash \{\mathit{source}\}$. Then, it is a matter of verifying each line of circuit 1 that the circuit is satisfied by the public and private inputs $h = H_i(B_*)$, $\mathit{account} = a$, $\mathit{balance} = \mathit{Bal}_{i-1}(T'_*, a) + \mathit{received}(T'_i, a) - \mathit{sent}(T'_i, a)$, $B = B_i$, $\mathit{prev\_h} = H_{i-1}(B_*)$, $\mathit{prev\_balance} = \mathit{Bal}_{i-1}(T'_*, a)$, $T = T'_i$, $\pi(*) = \pi_i(*)$, $P(*) = P_{i-1}(*)$ and $P_{\mathit{prev\_balance}} = P_{i-1}$.

These ZK-proofs are proofs of sufficient balance for every transaction sent by a in B_*, so the user can complete all of their transactions by sending inclusion proofs and ZK-proofs of sufficient balance to the recipients of each transaction. After the user has completed their transactions, we have that each P_i is a ZK-proof that the balance of their account a is at least $\mathit{Bal}_i(T_*, a)$.

Note that since each ZK balance proof is produced recursively by combining a ZK-proof of the previous balance and ZK validity proofs of received transactions, a user does not need to keep all previous ZK balance proofs. However, if a user receives a ZK validity proof of a transaction in a previous block, they need to regenerate all the ZK balance proofs after the received transaction. For this reason, each user should keep the ZK validity proofs for all transactions they have received, inclusion proofs of all transactions they have sent, as well as some checkpoint ZK proofs of their own balance.

4.6 Soundness of balance proofs

We will now show that the method of constructing ZK balance proofs is sound, meaning that it is not feasible for a user to construct a ZK-proof that the balance of their account is greater than its true balance in B_*. In order to do so, we will first show that the explicit proving method is sound.

Theorem 3 (Soundness of explicit balance proofs). *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence and let $T_* = (T_i)_{i=1}^N$ be the sequence of transaction maps constructed by combining* all *transaction map sequences that are both valid with respect to S_*, and where an inclusion proof of each of their transactions in B_* is collectively known (i.e. known by someone). This combined sequence of transaction maps is well defined, since by the binding property of the commitment scheme, we cannot construct inclusion proofs of two different transactions by the same sender. Then, if a user has an explicit balance proof*

$$(T'_*, \pi_*(*))$$

for B_*, we have

$$Bal_i(S_*, T'_*, account) \leq Bal_i(T_*, account)$$

for every i and $account \in Accounts$.

Proof. We have that each T'_i is a restriction of T_i, since if the inclusion proof of a transaction is known by one user, it is also known collectively. Then, the result follows from Proposition 1.

Theorem 4 (Soundness of ZK balance proofs). *Assuming that the ZK proving system is sound, the ZK balance proof is sound.*

Proof (Idea). The idea is to first prove that the only way for a single prover to generate a ZK-proof of the balance of an account at an index in a simple block sequence is to know an explicit balance proof of the same statement. Then, Theorem 3 gives us the wanted result.

5 Signatures

In this section we will add a signature mechanism for ensuring that each sender in a block has actually signed the transaction map commitment in the block. In doing so, we will define three kinds of rollup blocks, namely *transfer blocks*, used for transacting between rollup users, *registration blocks*, used for registering a new user on the rollup and *deposit blocks*, used for depositing funds to the rollup from an underlying blockchain. Given a sequence of rollup blocks, we will then derive a sequence of simple blocks by taking the sender set and commitment of the rollup blocks that have valid signatures. The derived simple block sequence can then be used to prove account balances as we described in Section 4.
We will start by fixing the set of accounts. We let

$$Accounts = \{source\} \cup L2\_accounts \cup L1\_accounts,$$

where $L2\_accounts = \mathbb{N}$. Here, *source* is the source account, used for depositing into the rollup, *L2_accounts* is the set of L2 addresses used for regular

transacting on the rollup, and *L1_accounts* is the set of L1 addresses which are used to withdraw funds from the rollup to L1.
Since the senders are signing the same commitment in each block, we will use BLS signature aggregation for efficient representation and verification of the signatures.

5.1 Transfer blocks

Transfer blocks are used for sending transactions on the rollup.

Definition 12 *A* transfer block *is a tuple* $(senders, root, signature)$*, where* $senders \subset L2\_accounts$*, root is a value in the set of possible merkle roots, and signature is a value in the set of possible BLS signatures.*

In order to be able to quickly add transfer blocks to the rollup, we will not require that *signature* is a valid BLS signature, or that *root* is the merkle root of a merkle tree in the definition of a transfer block. Instead, the transfer blocks with invalid signatures will be filtered out when deriving the simple block sequence from a rollup sequence. Also, it is the responsibility of each user to never sign a merkle root if they do not have an inclusion proof of their transaction in the merkle tree.

5.2 Registration blocks

Before a user can transact on the rollup, they need to be registered in a *registration block*. The purpose of the registration is two-fold. First, the registration will assign a unique rollup account to each user which will be used in the encoding of the sender set in a transfer block to save space. Second, in order to prevent the rogue key attack on BLS signatures, each user must prove that they know the private key corresponding to their public key before they can transact on the rollup. This is achieved by including a signature of the user's public key by the corresponding private key in the registration block.

Definition 13 *A* registration block *is a tuple* (pk, σ) *where* $pk \in \mathbb{G}_1$ *and* $\sigma \in \mathbb{G}_0$*. The registration block is* valid *if pk is a valid BLS signature of the message "I am registering the BLS public key* pk *on Intmax2".*

In order to be able to quickly add registration blocks to the rollup, we will not enforce every registration block to be valid. Instead, the invalid registration blocks will be filtered out when deriving the simple block sequence from a rollup sequence.

5.3 Deposit blocks

Since transfer blocks only allows sending from L2 accounts, we will need a different kind of block, called *deposit blocks*, for making deposits into the rollup.

Definition 14 *A deposit block is a tuple* $(recipient, amount)$*, where* $recipient \in L2\_accounts$ *and* $amount \in Amounts$*.*

Protocol 1 Registration Protocol.

Registration

Before a user can transact on the rollup, user Alice must register a BLS public key. To do so, user Alice performs the following steps:

– Alice generates a BLS secret key $x \overset{R}{\leftarrow} \mathbb{Z}_q$
– Alice obtains the corresponding public key $pk \leftarrow g_1^x \in \mathbb{G}_1$
– Alice produces a signature $\sigma \leftarrow \mathcal{H}(m)^x \in \mathbb{G}_0$, where m is the message "I am registering the BLS public key pk on Intmax2", which is a registration message exclusive to Alice and is cryptographically binding.
– Alice outputs the following registration block: (pk, σ)

Upon successful registration, an L2 address is assigned to Alice.

In this step, the signature proves that each user knows the private key corresponding to their public key, preventing the rogue key attack on BLS signatures. When a user registers a new account, the account is given an L2 address. For simplicity, this address can be seen as an integer that increments for each new account.

Protocol 2 Deposit Protocol.

Depositing to rollup

Upon completing the registration, and to start transacting, users must have a token balance on the rollup. To have such a balance, the user can either receive funds from another L2 user or deposit the funds themselves. We now describe the setting where Alice performs her own deposit of funds. To do so, user Alice performs the following steps:

– Alice creates a deposit block containing the destination L2 address, the token type (i.e., the token ID), and the amount of tokens to be deposited.
– Alice submits the deposit block to the rollup smart contract.

In this step, the destination L2 address does not necessarily have to belong to Alice, as she may be attempting to deposit funds into someone else's account.

5.4 Rollup sequence

Definition 15 *We let the set of rollup blocks $\mathcal{B}$ be the set of all* registration blocks*,* deposit blocks *and* transfer blocks*.*

Definition 16 *A* rollup sequence *is a finite sequence of rollup blocks.*

5.5 Signature validation

In order to verify the signatures in each transfer block in a rollup sequence, we need to keep track of the public keys assigned to each L2 account. We do this as follows.

Definition 17 *Let $B = (B_i)_{i=1}^N$ be a rollup sequence. For each $0 \leq i \leq N$ we define the sequence $pk(i)_* = (pk(i)_j)_{j=1}^{K(i)}$ of BLS public keys inductively as follows.*
We define $pk(0)_*$ to be the empty sequence. For $1 \leq i \leq N$ we define

$$pk(i)_* = (pk(i-1)_1, \ldots, pk(i-1)_K(i-1), key)$$

if $B_i = (pk, \sigma)$ is a valid registration block, and

$$pk(i)_* = pk(i-1)_*$$

otherwise.

Given a rollup sequence, we can derive a simple block sequence by taking all deposit blocks and all transfer blocks where the BLS signature is valid. Recall that a simple block is a tuple (S, C), where S is a set of accounts and C is an element of the set $\mathcal{C}$ of possible commitments. In order to support committing to both the transactions in a transfer block and the transaction in a deposit block, we will let

$$\mathcal{C} = \mathcal{B}_{deposit} \bigsqcup \{0,1\}^n,$$

i.e. a transaction map commitment is either a merkle root or a deposit block. An inclusion proof of a transaction in a commitment would then be either a merkle proof in the case where the commitment is a merkle root, or a trivial proof in the case where the commitment is a deposit block.[10]

Definition 18 *Let $B_* = (B_i)_{i=1}^N$ be a rollup sequence. We define the* simple block sequence derived from *B_* to be the simple block sequence $B'_* = ((S_j, C_j))_{j=1}^M$ constructed as follows. We start with the rollup sequence B_* and consider only the deposit blocks and the transfer blocks $B_i = (senders, root, signature)$ where signature is a valid BLS signature of root under the BLS public keys $(pk(i)_s)_{s \in senders}$. We then convert these deposit and transfer blocks into simple blocks by converting each deposit block $B_i = (recipient, amount)$ to the simple block $(\{source\}, (recipient, amount))$ and each transfer block $B_i = (senders, root, signature)$ to the simple block $(senders, root)$. Finally we reindex the resulting sequence of simple blocks to get B'_*.*

[10]Since the deposit transactions are public knowledge by the fact that they are included in deposit blocks in the rollup sequence, and since their inclusion proofs are trivial, all deposits are completed by default. This means that the recipient of the deposit will always be able to add the deposited amount to their balance proofs.

It it evident that the way we derive a simple block sequence from a rollup sequence is correct, meaning that an account will be in the sender set of a block in the simple block sequence derived from a rollup sequence if and only if either the sender is the source account or the sender has signed the commitment in the simple block with their BLS public key, and this signature is part of an aggregated signature of the commitment in one of the transfer blocks in the rollup sequence.

5.6 Zero knowledge balance proofs for a rollup sequence

We will now describe how to prove the balance of an account in a rollup sequence. This involves first proving what the simple block sequence derived from the rollup sequence is, and then using a ZK-proof of the balance in the simple block sequence.

We will start by defining the hash of a rollup sequence.

Definition 19 *Suppose we have an encoding of rollup blocks*

$$enc_{rollup} : \mathcal{B} \to \{0,1\}^*.$$

Let $B_* = (B_i)_{i=1}^N$ *be a rollup sequence. For each index* $0 \leq i \leq N$ *we define the* hash of B_* at index i*, written* $H_i(B_*)$ *inductively as follows. For* $i = 0$ *we define*

$$H_0(B_*) = Hash([0]),$$

and for $1 \leq i \leq N$ *we define*

$$H_i(B_*) = Hash(H_{i-1}(B_*)||Hash(enc_{rollup}(B_i))).$$

The following pseudocode describes the circuit for verifying that a simple block sequence is derived from a rollup sequence. It takes a hash h of the rollup sequence, the hash h' of the simple block sequence, and the sequence of public keys pk_* in the rollup sequence, and proves that the sequence of public keys are correct, and that the simple block sequence with hash h' is derived from the rollup sequence with hash h.

Circuit 2 Circuit for verifying that a simple block sequence is derived from a rollup sequence

```
Public input:
    a hash $h \in \{0,1\}^n$
    a hash $h' \in \{0,1\}^n$
    a sequence $pk_* = (pk_i)_{i=1}^M$ of public keys
Private input:
    a hash $prev\_h \in \{0,1\}^n$
    a hash $prev\_h' \in \{0,1\}^n$
    a sequence $prev\_pk_* = (prev\_pk_i)_{i=1}^{prev\_M}$ of public keys
    a rollup block $B$
    a simple block $B'$
    a ZK-proof $P$ that the rollup sequence with hash $prev\_h$ generates a validated
    block sequence with hash $prev\_h'$ and the sequence of public keys $prev\_pk_*$
 1: Verify $h = Hash(prev\_h||Hash(enc_{rollup}(B)))$
 2: Verify $h' = Hash(prev\_h'||Hash(enc(B')))$
 3: Verify that $pk_*$ is the sequence of public keys obtained by adding any new regis-
    tration in $B$ to $prev\_pk_*$.
 4: Verify the ZK-proof $P$
 5: if $B$ is a transfer block $B = (senders, root, signature)$ where $signature$ is a valid
    BLS signature of $root$ by the BLS public keys $(pk_s)_{s \in senders}$ then
 6:     Verify $B' = (senders, root)$
 7: else if $B$ is a deposit block $B = (recipient, amount)$ then
 8:     Verify $B' = (\{source\}, (recipient, amount))$
 9: else
10:     Verify $h' = prev\_h'$
11: end if
```

We can combine Circuit 2 with Circuit 1 to get the following circuit for verifying the balance of an account in a rollup sequence.

Circuit 3 Circuit for verifying the balance of an account in a rollup sequence

```
Public input:
    a hash $h \in \{0,1\}^n$
    $account \in Accounts$
    $balance \in Amounts$
Private input:
    a sequence $pk_* = (pk_i)_{i=1}^M$ of public keys
    a hash $h' \in \{0,1\}^n$
    a ZK-proof $P_1$ of Circuit 2 proving that the rollup sequence with hash $h$ gener-
    ates the public keys $pk_*$ and the derived simple block sequence with hash $h'$
    a ZK-proof $P_2$ of Circuit 1 proving that $account$ owns $balance$ in the simple
    block sequence with hash $h'$
 1: Verify $P_1$ and $P_2$
```

Our decision of splitting the balance proofs for a rollup sequence into a balance proof for a simple block sequence and a proof that the simple block sequence was derived correctly from the rollup sequence, has been made for performance reasons. Indeed, the ZK proof that the simple block sequence was derived correctly only needs to be generated once and distributed to all users. Then, users only need to prove their own balances, and not that the signatures are valid.

6 Rollup contract

We implement our design as a rollup on an underlying blockchain by deploying a rollup smart contract on the underlying blockchain. The rollup contract will keep track of a rollup sequence, and it has functions for adding new blocks to the sequence, for depositing funds from the underlying blockchain to the rollup, and for withdrawing funds from the rollup to the underlying blockchain.
The rollup contract fixes a rollup sequence, which we will refer to as *the rollup*, by storing the hashes of the rollup sequence in its storage. Each time a new block is added to the rollup, the new hash is computed by the rollup contract and added to its storage. In order to have high liveness guarantees, we will allow anyone to add new blocks to the rollup via the rollup contract. When withdrawing from the rollup to an L1 address, the rollup contract will verify a ZK-proof of the balance of the L1 address on the rollup and subtract from this balance the amount of funds that has previously been withdrawn to the L1 address (which is also stored in the contract). One implementation detail is that since this ZK-proof needs to refer to a specific hash, and the current hash of the rollup can change at any time (because blocks are added concurrently), we need to allow using a previous rollup hash. For this reason, the rollup contract will store a set of of recent rollup hashes, and not just the latest one.
In details, the rollup contract state is defined as follows.

Definition 20 *The* rollup contract state *is a tuple*

$$(h_*, total\_withdrawn\_amount)$$

where

$$h_* = (h_i)_{i=0}^N$$

are the hashes of the rollup sequence at each index, and

$$total\_withdrawn\_amount : L1\_address \to Amounts$$

is a map which stores the total amount that has previously been withdrawn to each L1 address.

We note that in order to save storage costs on the underlying blockchain, the rollup contract can have a feature where instead of adding each new rollup hash to a new slot in the contract storage, the new hash can instead overwrite the oldest hash in the storage. We can also have a feature where all hashes that are added in the same L1 block are written to the same storage slot. If we implement these two features together, we get a guarantee that each hash will be stored in the contract storage for the duration of at least $t \cdot n$, where t is the time between each block of the underlying blockchain, and n is the number of hashes that are stored in the contract.

6.1 Adding blocks to the rollup

New blocks are added to the rollup by calling the following contract function which can be called by anyone.

Rollup contract function 1 Adding a new block to the rollup.

```
Require:
    A rollup block $B$.
    An amount $deposited\_amount \in Amounts$ that is included with the L1 transac-
    tion. This would be non-zero only when adding a deposit block.
    The current state $(h_* = (h_i)_{i=0}^{N}, total\_amount\_withdrawn)$ of the rollup contract.
Ensure: $(h_*, total\_amount\_withdrawn)$ is the new state of the rollup contract.
 1: if $B = (recipient, amount)$ is a deposit block and $amount = deposited\_amount$
    or if $B$ is a non-deposit block then
 2:     $h_* \leftarrow (h_0, h_1, \ldots, h_N, Hash(h_N || enc_{rollup}(B)))$
 3: end if
```

6.2 Withdrawing from the rollup

To withdraw funds from the rollup to an L1 address, the following contract function must be called.[11] We refer the reader to Protocol 3, where we describe an overview of the withdrawal protocol.

[11] We mention that to increase performance, it is possible to implement batched withdrawals, where many ZK balance proofs are aggregated together by an aggregator into one proof that can be used to withdraw to many L1 addresses.

Protocol 3 Withdrawal Protocol.

Withdrawing funds

To withdraw funds, user Alice performs the following steps:

– Alice produces a zero knowledge proof P that proves that the rollup account representing her L1 account has a certain balance at a previous index of the canonical rollup: $\mathsf{Prove}(pp, x, w) \rightarrow P$. Moreover, those funds have not been spent since and the current block state builds on such a previous state.
– Alice submits the balance proof P to the withdrawal function in the rollup contract.

Upon receiving the proof, the withdrawal function performs the following steps:

– Withdrawal function verifies that the zero-knowledge proof verification outputs true: $\mathsf{Verify}(pp, x, P) \rightarrow \mathsf{Accept}$
– Checks the provided rollup hash is in the list of previous rollup hashes.
– Transfers the amount to the L1 address and updates the total amount withdrawn to the L1 address accordingly in the contract storage.

We highlight that anyone can add a new block to the rollup by creating a transaction to the rollup contract with the content of the new block. Furthermore, it is important to note that if a specific use case allows for the constant use of the funds in the rollup, then a user does not necessarily have to withdraw funds from the rollup and can constantly use the existing funds and subsequently deposit (or receive) funds on an ongoing basis.

Rollup contract function 2 Withdrawing from the rollup.

```
Require:
    A hash $h \in \{0,1\}^n$.
    An index $0 \le i \le N$
    An address $address \in L1\_addresses$ which we will withdraw to.
    An amount $balance \in Amounts$.
    A ZK-proof $P$ of Circuit 3 proving that $address$ owns $balance$ in the rollup sequence with hash $h$.
    The current state $(h_* = (h_i)_{i=0}^N), total\_amount\_withdrawn)$ of the rollup contract.
Ensure:
    $(h_*, total\_amount\_withdrawn)$ is the new state of the rollup contract.
    $withdrawn\_amount \in Amounts$ is the amount to be withdrawn.
1: if $h = h_i$ and if $P$ is a valid ZK-proof then
2:     $withdrawn\_amount \leftarrow balance - total\_amount\_withdrawn[address]$
3: else
4:     $withdrawn\_amount = 0$
5: end if
6: $total\_amount\_withdrawn[address] \leftarrow total\_amount\_withdrawn[address] + withdrawn\_amount$
```

7 Conclusion

We presented Intmax2, a novel zk-rollup approach that completely shifts away from traditional zk-rollup approaches. By leveraging the fact that aggregators do not need to perform computationally intensive zero-knowledge proofs, and instead moving the computation is on the side of the users in the system, our design provides a novel, practical, and resilient solution to L2 scaling.
In contrast with previous approaches, our solution does not require the posting of all transaction data on the underlying L1, and provides better liveness guarantees. On a final note, we highlight that unlike the majority of the deployed ZK-rollups platforms, our design allows for a much simpler path to the decentralization of the aggregator role, thus addressing one of the main existing problems in the rollup space.

References

1. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046 (2018), `https://eprint.iacr.org/2018/046`, `https://eprint.iacr.org/2018/046`
2. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. J. Cryptol. **17**(4), 297–319 (sep 2004). https://doi.org/10.1007/s00145-004-0314-9, `https://doi.org/10.1007/s00145-004-0314-9`
3. Bünz, B., Fisch, B., Szepieniec, A.: Transparent snarks from dark compilers. Cryptology ePrint Archive, Paper 2019/1229 (2019), `https://eprint.iacr.org/2019/1229`, `https://eprint.iacr.org/2019/1229`
4. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., Ward, N.: Marlin: Preprocessing zksnarks with universal and updatable srs. Cryptology ePrint Archive, Paper 2019/1047 (2019), `https://eprint.iacr.org/2019/1047`, `https://eprint.iacr.org/2019/1047`
5. Chiesa, A., Ojha, D., Spooner, N.: Fractal: Post-quantum and transparent recursive proofs from holography. Cryptology ePrint Archive, Paper 2019/1076 (2019), `https://eprint.iacr.org/2019/1076`, `https://eprint.iacr.org/2019/1076`
6. Dompeldorius, A.: Springrollup. `https://github.com/adompeldorius/springrollup` (2021), accessed: July 11, 2023
7. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953 (2019), `https://eprint.iacr.org/2019/953`, `https://eprint.iacr.org/2019/953`
8. Garg, S., Goel, A., Jain, A., Policharla, G.V., Sekar, S.: zkSaaS: Zero-knowledge snarks as a service. Cryptology ePrint Archive, Paper 2023/905 (2023), `https://eprint.iacr.org/2023/905`, `https://eprint.iacr.org/2023/905`
9. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing. p. 291–304. STOC '85, Association for Computing Machinery, New York, NY, USA (1985). https://doi.org/10.1145/22145.22178, `https://doi.org/10.1145/22145.22178`

10. Groth, J.: On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260 (2016), https://eprint.iacr.org/2016/260, https://eprint.iacr.org/2016/260
11. Hioki, L.: Intmax: Trustless and near-zero gas cost token transfer payment system. https://ethresear.ch/t/intmax-trustless-and-near-zero-gas-cost-token-transfer-payment-system/13904, accessed: July 11, 2023
12. Kothapalli, A., Setty, S.: Supernova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Paper 2022/1758 (2022), https://eprint.iacr.org/2022/1758, https://eprint.iacr.org/2022/1758
13. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370 (2021), https://eprint.iacr.org/2021/370, https://eprint.iacr.org/2021/370
14. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Paper 2019/099 (2019), https://eprint.iacr.org/2019/099, https://eprint.iacr.org/2019/099
15. Nguyen, W., Boneh, D., Setty, S.: Revisiting the nova proof system on a cycle of curves. Cryptology ePrint Archive, Paper 2023/969 (2023), https://eprint.iacr.org/2023/969, https://eprint.iacr.org/2023/969
16. team, B.F.: Plasma prime design proposal. https://ethresear.ch/t/plasma-prime-design-proposal/4222, accessed: July 11, 2023
17. Team, P.Z.: Plonky2: Fast recursive arguments with plonk and fri. https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf (2022), accessed: July 11, 2023

A Discussion

A.1 Tracing the Path to Intmax2

Plasma Prime [16] is the starting point for the path that lead to Intmax2. Plasma Prime incorporates RSA accumulators and is based on the UTXO model, where each unspent output represents ownership of a specific segment. The concept of range chunking is also introduced, and is used to compress transaction history to simplify block verification. This design also features the use of a SumMerkleTree for efficient overlap verification between transaction segments and inclusion proof generation.

Springrollup [6] is a Layer 2 solution that introduces a new type of zk-rollup, that aims to use less on-chain data and enhance privacy. The rollup state is divided into on-chain and off-chain available states, with the design ensuring users' funds remain safe even if the off-chain state is withheld by the operator. The operator can modify the rollup state by posting a rollup block to the L1 contract, which includes the new merkle state root, a diff between the old and new on-chain states, and a zk-proof of valid operations. The system also includes a frozen mode for situations where the operator doesn't post a new rollup block within 3 days.

Intmax [11] introduces a design where the aggregator maintains a global state that is used when the aggregator makes new rollup blocks. This state is not necessarily known by anyone other than the aggregator, and can be withheld by the aggregator. This means that to allow multiple aggregators for the rollup, each aggregator must be trusted to provide the updated rollup state off-chain to the next aggregator in order to keep the rollup alive. This results in two things: First, since each aggregator needs to build upon the previous block, this method requires the complexity of a leader selection method to determine which aggregator can create the next block. Second, and more importantly, the rollup will halt if one of the aggregators fails to provide the data to the next aggregator, and all users would need to exit the rollup. This means that all aggregators need to be trusted in order to guarantee liveliness.

Intmax2 (this work), solves these problems by modifying the protocol so that block production becomes stateless, meaning that new blocks can be added to the rollup without having to know the previous blocks at all, allowing aggregating to become decentralized.

A.2 Liveness

We highlight that if a user receives a transaction and then remains offline for an extended period of time, the user is still able to perform withdrawals at a future point in time when they are online again. While it is recommended that a user continuously performs the update of the recursive zero-knowledge balance proof that allows for the withdrawal of funds, the user can remain offline for a certain time period and then, when back online, can perform a synchronization process and calculate the corresponding recursive zero knowledge proof (e.g., [15]).

A.3 Privacy of Intmax2

Our proposed solution does not post any transaction data on the underlying layer 1. Also, since aggregators do not need to verify transactions, the transaction data can also be hidden from the aggregators. As a result, the details of user transactions are only revealed to the recipients. As the importance of privacy on blockchains continues to grow, our proposed solution offers a promising path towards a privacy-focused future.

A.4 Delegating Zero-Knowledge Proof Generation

The emergence of new research on delegating the generation of zero-knowledge proofs [8], brings exciting prospects for the wider adoption of these technologies, particularly among light clients like mobile phones. This development holds great promise in overcoming the computational limitations of resource-constrained devices and enabling them to actively engage in zero-knowledge proof protocols. By delegating the generation of zero-knowledge proofs to more powerful devices or servers, the burden of computationally intensive tasks can be alleviated, paving the way for enhanced participation and utilization of zero-knowledge proofs.
As the research continues to evolve and mature, we anticipate a future where zero-knowledge proofs become more accessible and seamlessly integrated into various domains, empowering users with enhanced security and privacy guarantees. This development holds immense potential for bringing zero-knowledge proofs to the masses and unlocking their benefits for various applications.

B Informal Security Notes

In this section, we briefly discuss the security aspects of the proposed construction, focusing on liveness, safety, and user assumptions.

B.1 Liveness

One of the key features of the proposed construction is its liveness, which allows any participant to become an aggregator. This decentralized approach ensures that transaction processing and updates can continue even in the absence or unavailability of a specific aggregator. The ability for users to readily assume the role of the aggregator promotes a distributed and collaborative environment, enhancing the system's resilience and adaptability.

B.2 Safety

Our construction also emphasizes strong safety properties, particularly in preventing unauthorized fund access. The system ensures that funds cannot be stolen by unauthorized parties, as users must provide valid proofs of balance to authorize transactions. Moreover, the completeness property guarantees that users can always withdraw their funds to the underlying blockchain.

B.3 Malicious Users

Users can choose to not sign the Merkle root of the tree of transactions. Failure to do so results in a situation where the user's transaction is effectively voided, preventing them from proving its existence in the corresponding zero-knowledge proofs used for withdrawals. Similarly, if the aggregator fails to send the Merkle proof to a specific user, the user's transaction will not be counted as included in that set. As a consequence, the user will not be able to prove the transaction's validity in zero-knowledge, preventing them from claiming any funds associated with that (voided) transaction.
Alternatively, a user may attempt to spam the network with a very high number of dummy (invalid) transactions to attempt to increase the size of the Merkle proofs that are sent to each user in an attempt to bloat the local storage of individual users. This attack, however, requires exponential effort from the attacker as the Merkle proof size is logarithmic in the number of leaves.

C Security Proof

We assume an adversary attempting to subvert the security of our construction. Therefore, $\mathcal{A}$ may attempt to explore different attack vectors. For example, $\mathcal{A}$ may attempt to forge a proof of inclusion for the used Merkle tree, produce a zero knowledge proof forgery, randomly go offline in an attempt to disrupt the liveness of the system, or even even censor specific transactions from users. These represent different attack vectors that we model in this section.

C.1 Safety

To break the safety of the rollup system, $\mathcal{A}$ may target the soundness of the used zero-knowledge scheme to prove ownership of funds. This assumptions stems from the fact that the soundness of the zero-knowledge scheme guarantees with very high probability that any attempt to forge or modify a valid state will be detected, thus preserving the security of the system.

Zero-Knowledge Proof Forgery

Theorem 5. *Given a zero-knowledge proof π, a statement x, and a set of public parameters pp generated to provide a security parameter λ, the adversary $\mathcal{A}$ has a negligible probability of producing a zero-knowledge proof forgery, assuming the soundness property of the zero-knowledge scheme.*

Proof. (Sketch.) We consider the soundness of the zero-knowledge scheme a critical property for ensuring the security of the proof. The soundness property guarantees that an adversary $\mathcal{A}$ cannot produce a valid zero-knowledge proof unless they possess the correct witness. To break the soundness property, $\mathcal{A}$ must find a witness w' that makes the verifier accept an invalid proof π' generated from $\mathsf{Prove}(pp, x, w')$. However, the soundness property ensures that the probability of $\mathcal{A}$ successfully executing this attack is negligible, typically bounded by 2^{-k} where k represents the knowledge error.
Therefore, as long as the zero-knowledge scheme is instantiated with appropriate parameters and exhibits the soundness property, the probability of an adversary producing a zero-knowledge proof forgery is negligible.
Thus, based on the assumption of soundness and the negligible probability of forging a zero-knowledge proof, we can conclude that the zero-knowledge scheme provides the desired security against proof forgery attempts.

Commitment Scheme

To break the safety of the rollup system, $\mathcal{A}$ may target the security properties of the used commitment scheme, which ensures the integrity of each new state.

Theorem 6. *Given a commitment $\mathcal{C}$ and a transaction tx such that $\mathsf{Commit}(\mathsf{tx}) \rightarrow \mathcal{C}$, $\mathcal{A}$ has negligible probability of producing a $\mathsf{tx}' \neq \mathsf{tx}$ such that $\mathsf{Commit}(\mathsf{tx}') = \mathsf{Commit}(\mathsf{tx})$, if the used commitment scheme is binding.*

Proof. (Sketch.) We aim to prove that, assuming a binding property of the used commitment scheme, the probability of an adversary $\mathcal{A}$ producing a different value that matches the commitment value is negligible.
The binding property ensures that it is not computationally feasible to manipulate the opening phase and use a different value as it results in the commitment opening to a different message.
Consider the scenario where $\mathcal{A}$ attempts to produce a malicious value for a given commitment $\mathcal{C}$ to a transaction tx. To succeed, $\mathcal{A}$ must find a rogue transaction $\mathsf{tx}' \neq \mathsf{tx}$ such that $\mathsf{Commit}(\mathsf{tx}') = \mathsf{Commit}(\mathsf{tx})$. The binding property guarantees that the probability of finding such a transaction is negligible.

C.2 Liveness

To break the liveness property of the system, the adversary may attempt to go offline over extended periods of time or by censoring transactions from specific users. We now show that these attacks do not compromise the liveness property of the system.

Theorem 7. *In a rollup system with a designated aggregator responsible for submitting batch updates to the underlying layer 1, if a malicious aggregator attempts to disrupt liveness by going offline, the system can maintain liveness as long as there exists at least one honest participant in the system who can assume the role of the aggregator.*

Proof. (Sketch.) We aim to prove that in the given rollup system, liveness can be sustained even if a malicious aggregator goes offline, as long as there exists at least one honest participant in the system who can seamlessly transition to the role of the aggregator.
Let us consider a scenario where a malicious aggregator intentionally goes offline, disrupting the regular batch update process. Due to the decentralized nature of the rollup system, any honest participant can readily assume the role of the aggregator.
Since the rollup system does not depend on any specific entity as the aggregator, the ability to transition the role to an honest participant ensures the continuity of transaction processing and updates. The honest participant, upon assuming the aggregator role, can effectively submit batch updates to the underlying layer 1, thereby maintaining the liveness property of the system.
Thus, we can conclude that in the given rollup system, liveness can be maintained despite the malicious aggregator going offline, as long as there exists at least one honest participant who can assume the role of the aggregator.

Theorem 8. *In a rollup system with a designated aggregator responsible for submitting batch updates to the underlying layer 1, if a malicious aggregator attempts to censor transactions from users, the system can overcome censorship and maintain liveness if one or more honest party assumes the role of the aggregator.*

Proof. (Sketch.) We aim to prove that in a rollup system where an aggregator is responsible for submitting batch updates to the underlying layer 1, if a malicious aggregator attempts to censor transactions from users, the system can overcome censorship as long as each of these censored users can assume the role of the aggregator.
Consider a scenario where a malicious aggregator attempts to censor transactions from certain users by intentionally excluding their transactions from the batch updates. However, the decentralized design of the rollup system empowers users to become aggregators themselves.
In this case, if a user perceives censorship or exclusion of their transactions by the aggregator, they can opt to become an aggregator and directly submit batch updates to the underlying layer 1. By taking over the aggregator role, the user-turned-aggregator ensures that their transactions are included in the batch updates.
The ability of users to bypass the malicious aggregator and become aggregators themselves provides a mechanism to overcome censorship within the system, which ensures that transactions from users are not unduly suppressed or excluded, maintaining the desired liveness property.
Therefore, we can conclude that in such a rollup system, even if a malicious aggregator attempts to censor transactions from users, the system can overcome censorship and maintain liveness as long as there exists at least one honest participant who can assume the role of the aggregator.