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 blockchain scaling solution called Intmax2, which is a Zero-Knowledge rollup (ZK-rollup) protocol featuring stateless and decentralized block production, while minimizing the usage of data and computation on the underlying blockchain. Our architecture distinctly diverges from existing ZK-rollups since essentially all of the data and computational costs are shifted to the client-side as opposed to imposing heavy requirements on the block producers or the underlying Layer 1 blockchain. The only job for block producers is to periodically generate a commitment to a set of transactions, distribute inclusion proofs to each sender, and collect and aggregate signatures by the senders. This design allows permissionless and stateless block production, and is highly scalable with the number of users.

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

1 Introduction

As the blockchain ecosystem continually evolves, so does the urgency for blockchain scaling 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 onchain. Existing ZK-rollups, while managing to move computation costs away from the underlying Layer 1 (L1) blockchain, are still limited by the fact that all necessary data for verifying users' balances have to be posted on L1. This data, in a typical scenario, includes the transaction sender, the index of the token, the amount, and the recipient for each transaction, thus limiting the number of transactions per second that can be supported by the rollup.

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 achieve data availability by leveraging the data availability of the underlying blockchain and require that all transaction data is posted to L1 (e.g. using calldata or blob data on Ethereum). Because this data needs to be replicated among a large set of nodes, there is a limit on how much data can be made available, which limits the number of transactions per second that the blockchain or the rollup 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 the set of senders who have signed the commitment, confirming that they have received inclusion proofs of their transactions. Users can then generate Zero-Knowledge proofs (ZK-proofs) of their own balances by combining the inclusion proofs of their sent transactions with the inclusion proofs and ZK-proofs of sufficient balance of each received transaction, which is provided by the transaction sender offchain. Our rollup design uses this method to achieve increased throughput compared to existing alternatives. In addition, the design allows permissionless block building that can happen in parallel, without needing any leader election or any coordination between the block builders. Since the block builders do not verify the validity of the transactions, they can be fully stateless, allowing a very simple and censorship resistant rollup design.

1.2 Our Contributions

Intmax2 is an efficient and stateless rollup design that:

- Only posts the index[3] of each transaction sender, as well as a merkle tree root and an aggregated signature of it for each batch on L1.
- Allows senders to transfer an unlimited number of tokens to an unlimited number of recipients while using a fixed 4-6 bytes data consumption.
- Shifts the computational requirements from the aggregator to the client, making it highly scalable with the number of users.
- Offers permissionless block production.
- Provides stronger privacy properties than traditional ZK-rollups.

[3]This can be as low as 4-6 bytes without compression, depending on the number of users in the system.

1.3 Our Design

We divide the description of our design into four parts.

First, in Section 3 we define a simple blockchain where all transaction data is public, and where everyone can verify the balance of every account.

Second, in Section 4 we show how users can prove their balances in a smaller blockchain whose blocks consists of commitments to a set of transactions (e.g. merkle roots) together with the set of senders who have signed the commitment in the block. For simplicity, we first show how users can construct explicit balance proofs of their balances, and then replace the explicit proofs with recursive ZK-proofs [12,13,17], which results in a much more efficient approach.

Third, in Section 5 we describe how we leverage BLS signatures to verify that each sender has signed the commitment in each block. This involves defining three block types (i.e., transfer, deposit, and registration). We will also define the set of rollup accounts which includes special accounts used for withdrawing funds to the underlying blockchain.

Finally, in Section 6 we turn our design into a rollup by deploying a rollup contract on an underlying blockchain. We describe the rollup contract and its functions, i.e. how to add blocks, how to deposit funds, and how to perform withdrawals. 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.

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].

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 Transaction map sequences

In this section we will describe transaction maps, which are collections of transactions, and sequences of such transaction maps, in which anyone can compute the balances of any account. We let $(Amounts, +, \leq)$ be a partially ordered

abelian group of amounts.[4] We also have a set *Accounts* of accounts, with a designated $source \in Accounts$ which will be the only account allowed to have negative balances. In the final rollup design, the source account will be used to deposit funds from the underlying blockchain to the rollup.

Definition 1 (Transaction) *A* transaction[5] *is a function*

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

which maps a finite set $Recipients(t) \subset Accounts$ *of* recipients *to the amount received by each recipient. Here,* $Amounts^+ \subset Amounts$ *is the set of all amount* $\in Amounts$ *where amount* ≥ 0.

A transaction does not include a sender. Instead, we will collect transactions from different senders together in *transaction maps*.

Definition 2 (Transaction map) *A* transaction map *is a function*

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

which maps a finite set $Senders(T) \subset Accounts$ *of* senders *to the transaction sent by each sender.* [6]

We get the total amount sent and received by an account in a transaction map as follows.

Definition 3 (Amount sent and received in a transaction map) *Let* T *be a transaction map. For each* $a \in Accounts$ *we define*

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

and

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

[4] 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 $f(t) \leq g(t), \forall t \in Tokens$.

[5] 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.

[6] Notice that this definition enforces that each sender can only send one transaction in each transaction map, but this is not a limitation since each transaction can have an unlimited number of recipients.

The balance of an account in a sequence of transaction maps is defined as the total amount received minus the total amount sent in the sequence.

Definition 4 (Balance) *Let $T_* = (T_i)_{i=1}^N$ be a sequence of transaction maps. For all $0 \leq i \leq N$ and $a \in Accounts$ we define the* balance of a in T_* at index i *to be*

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

For simplicity, when $i = N$ we will omit the index and write $Bal(T_*, a) = Bal_N(T_*, a)$, which we simply call the balance of a in T_*.

To prevent accounts from overspending, we will require that every non-source account has positive balances.

Definition 5 (Account with positive balances) *An account $a \in Accounts$ has* positive balances in a transaction map sequence T_* *if $Bal_i(T_*, a) \geq 0$ for all i. If every non-source account has positive balances in T_*, we will say that T_* is* balance-positive.

Proposition 1 *An account $a \in Accounts$ has positive balances in a transaction map sequence T_* iff $Bal_i(T_*, a) \geq 0$ for all i where $a \in Senders(T_i)$.*

Proof. This follows because an account balance can only decrease and become negative at the transaction maps where the account sent a transaction.

4 Proving balances from partial transaction data

If a transaction map sequence T_* is publicly known, every user can compute their balances in it. Now, the question is, can we reduce the amount of data that needs to be public? For instance, can we have only a *commitment*, e.g. a merkle root, of each transaction map in T_* be publicly known, and then having each user prove that they have a certain balance by providing a transaction map sequence $T'_* = (T'_i)_{i=1}^N$ together with inclusion proofs that prove that each T'_i is a restriction of T_i? It is clear that the balance of an account in such a sequence T'_* can only be a lower bound on the account's balance in T_* if all transactions previously sent by the account in T_* are included in T'_* (otherwise, a user could just omit their sent transactions to get an artificially high balance). To be able to verify that T'_* includes all transactions previously sent by the account in T_*, the set of senders in each transaction map in T_* must also be publicly known. In summary, we will replace a transaction map sequence with a sequence of transaction map commitments and associated sender sets. To define this, we will first define transaction map commitments.

4.1 Transaction map commitments

For committing to a transaction map, we will need a commitment scheme that takes a transaction map T and returns a commitment $C \in \mathcal{C}$ in a set $\mathcal{C}$ of possible commitments, and inclusion proofs $\pi(s) \in \mathit{\Pi}$ in a set $\mathit{\Pi}$ of possible inclusion proofs for every sender $s \in \mathit{Senders}(T)$. The commitment scheme has a verifying function, which takes a transaction, a sender, a commitment and an inclusion proof, and returns *true* if the inclusion proof is valid and *false* otherwise. The commitment scheme must be *binding*, in the sense that it should be computationally infeasible to construct valid inclusion proofs of two different transactions by the same sender in a transaction map. We can realize such a commitment scheme by having the commitment for a transaction map T be the merkle root of a merkle tree which stores for each $s \in \mathit{Senders}(T)$ a hash of the transaction $T(s)$ at the leaf whose merkle path is determined by s.[7]

Definition 6 (Simple block) *A* simple block *is a tuple (S, C), where $S \subset \mathit{Accounts}$ is a finite set of accounts called the* senders *of the simple block, and $C \in \mathcal{C}$ is a transaction map commitment.*

In practice, the senders in a simple block must have signed the corresponding transaction map commitment, which we will explain in Section 5. Signing the transaction map commitment serves as a confirmation that the sender intended to send their transaction in the committed transaction map, and it also provides a way to guarantee that honest users will always have inclusion proofs for their transactions, since they can refuse to sign the commitment if they don't have an inclusion proof of their transaction.

4.2 Local views and explicit balance proofs

A *local view* of a simple block sequence is a transaction map sequence together with inclusion proofs that the transactions are in the simple block sequence.

Definition 7 (Local view of a simple block sequence) *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence. A* local view *of B_* is a tuple $(T_*, \pi_*(*))$ where $T_* = (T_i)_{i=1}^N$ is a transaction map sequence such that $\mathit{Senders}(T_i) \subset S_i$ for all i, and*

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

is an indexed family of inclusion proofs where each $\pi_i(s)$ is a valid inclusion proof of transaction $T_i(s)$ by sender s in the commitment C_i.

[7] Note that it is possible to construct an invalid merkle tree, where some of its leaves are not valid transactions, but this would not be an issue since the binding property still holds in this case. We also note that KZG commitments can be used instead of merkle trees.

When the context is clear, we will abuse the terminology and identify a local view $(T_*, \pi_*(*))$ with its transaction map sequence T_*. For instance, we will say that an account has *positive balances* in the local view and that a local view is *balance-positive* if the same things can be said about its transaction map sequence. As previously mentioned, a requirement for being able to prove a lower bound on the balance of an account in a simple block sequence B_* from a local view, is that it contains every transaction previously sent by the account in B_*.

Definition 8 (Valid balance) *Let $(T_*, \pi_*(*))$ be a local view of a simple block sequence $B_* = ((S_i, C_i))_{i=1}^N$, let $a \in Accounts$ and $0 \leq i \leq N$. The account a is said to have a* valid balance at index i in the local view *if we have $a \notin S_j \backslash Senders(T_j)$ for all $j \leq i$. If the account has valid balance at every index i where $a \in Senders(T_i)$, the account is said to have a* valid transaction history in the local view. *If every account has a valid transaction history in the local view, we will say that* the local view has valid transaction histories.

If a local view of a simple block sequence is both balance-positive and has valid transaction histories, it can be used to prove the balance of an account in the simple block sequence. These proofs will be called *explicit balance proofs* (which will later be replaced by ZK balance proofs).

Definition 9 (Explicit balance proof) *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence. A local view $p = (T_*, \pi_*(*))$ is called an* explicit balance proof *if it has valid transaction histories and is balance-positive. If an account $a \in Accounts$ has a valid balance in T_* at index i, we will call p an* explicit proof *that the balance of a in B_* at index i is $Bal_i(T_*, a)$.*

We will now prove some important facts about local views, culminating in a soundness theorem for explicit balance proofs. The first fact is that we cannot increase the balance of an account in a local view by removing some of the transactions in the local view.

Proposition 2 *Let $B_* = ((S_i, C_i))_{i=1}^N$ be a simple block sequence and let $(T_*, \pi_*(*))$ and $(T'_*, \pi'_*(*))$ be local views of B_* where each T'_i is a restriction of T_i. Then, if $a \in Account$ has a valid balance at an index i in T'_*, the account also has a valid balance at i in T_*, and we have $Bal_i(T'_*, a) \leq Bal_i(T_*, a)$.*

Proof. We first show that the account has a valid balance at index i in T_*. This follows because 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. To prove the inequality, 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 we get

$$\begin{aligned} Bal_i(T'_*, a) &= \sum_{j=1}^{i} received(T'_j, a) - \sum_{j=1}^{i} sent(T'_j, a) \\ &\leq \sum_{j=1}^{i} received(T_j, a) - \sum_{j=1}^{i} sent(T_j, a) \\ &= Bal_i(T_*, a). \end{aligned}$$

Local views can be combined to form a greater local view.

Definition 10 (The combination of local views) *Let $(T^1_*, \pi^1_*(*))$ and $(T^2_*, \pi^2_*(*))$ be two known local views for a simple block sequence. Since the commitment scheme is binding, we have that for each i, the transaction maps T^1_i and T^2_i and the families of inclusion proofs $\pi^1_i(*)$ and $\pi^2_i(*)$ agree on their common intersections*[8]*. Then, the* combination of the two local views *is defined as the local view $(T_*, \pi_*(*))$, where for each i we let T_i and $\pi_i(*)$ be resp. the unique extensions of T^1_i and T^2_i, and of $\pi^1_i(*)$ and $\pi^2_i(*)$, to the union of their domains.*

The properties of local views are preserved when taking combinations.

Proposition 3 *Let $(T_*, \pi_*(*))$ be the combination of two local views $(T^1_*, \pi^1_*(*))$ and $(T^2_*, \pi^2_*(*))$ of a simple block sequence. Then we have the following.*

a) *If an account has valid transaction histories in T^1_* and T^2_*, then the account also has a valid transaction history in T_*.*
b) *If the account in addition has positive balances in T^1_* and T^2_*, then the account also has positive balances in T_*.*

Proof. Let $a \in Accounts$ be an account with valid transaction histories in both T^1_* and T^2_*, and let i be an index where $a \in Senders(T_i)$. Then we have either $a \in Senders(T^1_i)$ or $a \in Senders(T^2_i)$, which means that the account has a valid balance at index i in either T^1_* or T^2_*. Let T'_* be the one in which the account has valid balance at index i. It follows from Proposition 2, using T'_* and T_*, that the account has valid balance in T_* at index i. For part b) the assumption gives us that a has positive balance at index i in T'_*, so Proposition 2 also gives us $Bal_i(T_*, a) \geq Bal_i(T'_*, a) \geq 0$. By Proposition 1, we only needed to check for positive balances for the indices where the account sent a transaction, which we have now done.

Corollary 1 *The combination of explicit balance proofs is again an explicit balance proof.*

[8]Technically speaking, for this to be true, we must either have that the chosen commitment scheme has at most one inclusion proof for a transaction in a transaction map, or we can instead consider *equivalence classes* of inclusion proofs, where two valid inclusion proofs of the same transaction by the same sender are equivalent.

We will now show that is infeasible to construct an explicit proof of a balance which is greater than the true balance of an account.

Theorem 1 (Soundness of explicit balance proofs). *Let B_* be a simple block sequence and let $(T_*, \pi_*(*))$ be the explicit balance proof for B_* obtained by combining all explicit balance proofs for B_* that are collectively known (i.e. known by someone). Then, if a user has an explicit balance proof $(T'_*, \pi'_*(*))$ for B_*, we have $Bal_i(T'_*, a) \leq Bal_i(T_*, a)$ for every $a \in Accounts$ and every i where the account has a valid balance in T'_*. In other words, the balance of an account cannot be proven to be larger than its balance in T_*, which we consider to be the* true balance *of the account.*

Proof. We clearly have that each T'_i is a restriction of T_i, since if an explicit balance proof is known by one user, it is also known collectively. Then, the result follows from Proposition 2.

4.3 Constructing balance proofs and completing transactions

In order for a transaction recipient to be able to add the received amount to their balance, the sender must complete the transaction by sending an inclusion proof and a validity proof, i.e. a proof that the sender had sufficient balance for sending the transaction, to the recipient. The validity proof can simply be an explicit proof of the sender's balance after sending the transaction.

Definition 11 (Transaction validity proof) *An* explicit validity proof *of the transaction in a simple block sequence $B_* = ((S_i, C_i))_{i=1}^N$ by a sender $s \in Senders(T_i)$ is an explicit balance proof of the sender's balance at index i.*

In practice, however, we will replace explicit balance and transaction validity proofs with corresponding *ZK-proofs*, which we will define later. We will now talk about balance and validity proofs in general, before we specialize to explicit proofs and ZK-proofs.

Definition 12 (Completed transaction) *A transaction is said to be (explicitly or ZK)* completed *if every recipient has received an inclusion proof and a (explicit or ZK) validity proof of it.*

If a user knows inclusion proofs for every transaction they have sent, as well as enough completed transactions received to get sufficient balance for every sent transaction, they can construct balance proofs for their account, which can then be sent (together with inclusion proofs) to the recipients of each sent transaction in order to complete them. In details, the user needs the following data to prove their balances.

Definition 13 (Data set for constructing balance proofs for an account) *A* data set for constructing (explicit or ZK) balance proofs for an account $a \in$

Accounts in a simple block sequence $B_* = (B_i)_{i=1}^N$ *is a tuple* $(T_*, \pi_*(*), p_*(*))$, *where* $(T_*, \pi_*(*))$ *is a local view of* B_* *and where*

$$p_*(*) = \big(p_i(s)\big)_{1 \le i \le N,\ s \in Senders(T_i) \setminus \{a\}}$$

is an indexed family of (explicit or ZK) transaction validity proofs where each $p_i(s)$ *is a validity proof of the transaction by* s *at index* i *in* T_*. *The data set is said to be* complete *if* a *has positive balances in* T_*, *and if* T_* *contains every transaction sent by* a *in* B_*.

Each user will know a data set for their account which consists of every completed transaction received by the account (whose inclusion and validity proofs are known to the user by definition), as well as every transaction sent by the account whose inclusion proofs are known by the user. In order for this data set to be complete, the user should obey the following rule.

Definition 14 (Signing rule) *Let* B_* *be a simple block sequence. A user* obeys the signing rule *if they never sign a transaction map commitment unless the following conditions are met.*

1. *The user knows an inclusion proof of a transaction sent by their account in the committed transaction map*
2. *The user has no pending transactions, meaning that* B_* *contains all transaction map commitments that the sender has signed.*[9]
3. *The user's account has sufficient balance for the new transaction according to the data set for the account known by the user.*

Theorem 2. *If the owner of an account obeys the signing rule,*[10] *the data set for the account known by the owner is complete.*

Proof. Let B_* be the simple block sequence and $(T_*, \pi_*(*), p_*(*))$ be the data set for an account $a \in Accounts$ known by its owner at the time of signing a transaction map commitment. We observe that there will be no transactions sent by a in B_* from the time when the commitment was signed by the sender, to the time when the new commitment is included in B_*. This means that the balance $Bal(T_*, a)$ cannot decrease while the new transaction is waiting to be included. Since this balance was sufficient for the new transaction at the time the commitment was signed, it will still be sufficient by the time the new transaction is included in B_*.

We will now show how to construct explicit balance and transaction validity proofs from a complete data set.

[9] This condition can be relaxed to requiring that the account has sufficient balance for *all* of its pending transactions including the new transaction.

[10] 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 the simple block sequence are unique. If nonces are implemented, we do not need this requirement.

Theorem 3. *Let B_* be a simple block sequence, and suppose we know a complete data set $(T_*, \pi_*(*), p_*(*))$ for generating the explicit balance proofs for an account $a \in Accounts$. Then we will be able to generate an explicit balance proof that the balance of a at each index i in B_* is at least $Bal_i(T_*, a)$. We will also be able to explicitly complete all transactions sent by a in B_*.*

Proof. We will construct the local view $p' = (T'_*, \pi'_*(*))$ of B_* by combining the explicit validity proofs $p_*(*)$ with the local view $(T_*, \pi_*(*))$. We will show that this combined local view is an explicit balance proof, meaning that it has valid transaction histories and is balance-positive. We first show that T'_* has valid transaction histories. To do this, we first observe that p' can also be constructed by taking the combination of the local views in $\pi_*(*)$ and the local view $(T^{sent}_*, \pi^{sent}_*(*))$ where we have restricted p to just the transactions sent by a. Then, since every local view in $\pi_*(*)$ as well as $(T^{sent}_*, \pi^{sent}_*(*))$ have valid transaction histories, the combination also has valid transaction histories by Proposition 3. We then show that every non-source account has positive balances in T'_*. Let $a' \in Accounts \backslash \{source\}$. Then, if $a' = a$ we have that a' has positive balances in every local view in $\pi_*(*)$, as well as in $(T_*, \pi_*(*))$, so this follows from Proposition 3. If $a' \neq a$, we have that a' has positive balances in every local view in $\pi_*(*)$, as well as in $(T^{sent}_*, \pi^{sent}_*(*))$, so this also follows from Proposition 3. We have now shown that the combined local view p' is both valid and balance-positive, and since it contains every transaction sent by a in B_* it is an explicit balance proof that the balance of a at index i in B_* is at least $Bal_i(T_*, a)$ for every i.

4.4 ZK balance proofs

Explicit balance and transaction validity proofs have several drawbacks. They use a lot of space, are expensive to verify and they leak private transaction data of the senders to the recipients. These drawbacks are solved by replacing explicit proofs with corresponding ZK-proofs. In order to be able to recursively construct these ZK-proofs and to efficiently verify them, we will not use the whole simple block sequence as public input, but instead use the *hash* of the simple block sequence, which we define as follows.

Definition 15 (Hash of a simple block sequence) *Given a hash function $Hash : \{0,1\}^* \to \{0,1\}^n$ and an encoding of simple blocks $enc_{simple} : \mathcal{B}_{simple} \to \{0,1\}^*$, we define the* hash of a simple block sequence $B_* = (B_i)_{i=1}^N$ at index i*, written $H_i(B_*)$ inductively as follows. We define $H_0(B_*) = Hash([0])$, and for $1 \leq i \leq N$ we define $H_i(B_*) = Hash(H_{i-1}(B_*) \| Hash(enc_{simple}(B_i)))$. For simplicity, we will write $H(B_*) = H_N(B_*)$, which we call the* hash of B_*.

We will use Circuit 1 to prove an account's balance in a simple block sequence.[11]

[11] Note that the circuit only verifies that the balance of an account is *at least* the purported balance, allowing for greater privacy, since a user can generate ZK balance proofs without revealing their whole balance.

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

Public input:

a hash $h \in \{0,1\}^n$
$account \in Accounts$
$balance \in Amounts$

Private input:

a hash $prev\_h \in \{0,1\}^n$
$prev\_balance \in Amounts$
a ZK-proof $P_{prev\_balance}$ that the balance of $account$ at $prev\_h$ is at least $prev\_balance$
a simple block $B = (S, C)$
a transaction map T
inclusion proofs $\pi(*) = (\pi(s))_{s \in Senders(T)}$ that the transactions in T are in B.
ZK-proofs $P(*) = (P(s))_{s \in Senders(T) \setminus \{account, source\}}$ where each $P(s)$ is a ZK balance proof that s has a balance of at least 0 in the simple block sequence with hash h.

1: **if** $h = Hash([0])$ **then**
2: Verify $balance = 0$
3: **else**
4: Verify the ZK-proofs $P_{prev\_balance}$ and $P(*)$, and the inclusion proofs $\pi(*)$
5: Verify $h = Hash(prev\_h \parallel Hash(enc_{simple}(B))$
6: Verify that $account \notin S \setminus Senders(T)$
7: Verify that $balance \leq prev\_balance + received(T, account) - sent(T, account)$
8: Verify $balance \geq 0$ if $account \neq source$
9: **end if**

Definition 16 (ZK balance and transaction validity proof) *A* ZK-proof *that an account $a \in Accounts$ has a balance of at least $b \in Amounts$ at index i in a simple block sequence B_* is a ZK-proof that Circuit 1 can be satisfied with the public inputs $h = H_i(B_*)$, $account = a$ and $balance = b$. A* ZK validity proof *of the transaction of a sender $s \in S_i$ at index i in B_* is a ZK-proof that the balance of the sender at index i in B_* is at least 0.*

Theorem 4 (Soundness of ZK balance proofs). *Assuming soundness of the ZK scheme and collision resistance of the hash function, it is infeasible to generate a ZK-proof that the balance of an account is greater than the true balance of the account.*

Proof (Idea). The idea is to prove that the only way for a single prover to generate a ZK balance proof is to know a corresponding explicit balance proof, which were proven to be sound in Theorem 1.

We will now show how to construct ZK balance and transaction validity proofs from a complete data set.

Theorem 5. *Given a complete data set $(T_*, \pi_*(*), P_*(*))$ for generating the ZK balance proofs for an account $a \in Accounts$ in a simple block sequence B_*, we can generate ZK balance proofs $(P_i)_{i=0}^N$, where P_i is a ZK-proof that the*

balance of a at index i in B_* is at least $Bal_i(T_*, a)$. We will also be able to ZK complete all transactions sent by a in T_*.

Proof. The user can construct the ZK balance proofs inductively as follows.

Base case: $i = 0$ The user can trivially generate the ZK proof P_0 that circuit 1 is satisfied by the public inputs $h = Hash([0])$, $account = a$, and $balance = Bal_0(T_*, a) = 0$.

Induction step: $1 \leq i \leq N$ **and the user has constructed** P_{i-1} It is straightforward to verify that the circuit is satisfied by the public and private inputs $h = H_i(B_*)$, $account = a$, $balance = Bal_i(T_*, a)$, $B = B_i$, $prev\_h = H_{i-1}(B_*)$, $prev\_balance = Bal_{i-1}(T_*, a)$, $T = T_i$, $P(*) = P_i(*)$, $\pi(*) = \pi_i(*)$ and $P_{prev\_balance} = P_{i-1}$.

We can obtain ZK validity proofs for every transaction sent by a in B_* by replacing *balance* with 0 in the inputs used for each P_i, since the circuit is clearly still satisfied when setting the balance to zero.

Note that since only the last ZK balance proof and the ZK validity proofs of received transaction are needed to construct a ZK balance proof, users don't have to keep all previous balance proofs. However, when a user receives a ZK validity proof of an incoming transaction, they must regenerate all the ZK balance proofs after the received transaction. This means that users should keep the data set for their account, as well as some periodic ZK-proofs of their own balance to avoid having to recompute all the proofs from the beginning.

5 Signatures and rollup sequences

In this section we will add a signature mechanism for ensuring that the senders in a block have actually signed the corresponding transaction map commitment. In doing so, we will define three kinds of rollup blocks, namely *registration blocks*, used for registering a new user on the rollup, *transfer blocks*, used for transacting between rollup users, and *deposit blocks*, used for depositing funds to the rollup from an underlying blockchain.

Definition 17 (Rollup block and rollup sequence) *A rollup block is either a* registration block*, a* deposit block *or a* transfer block*, which are defined below. A* rollup sequence *is a finite sequence of rollup blocks.*

Given a rollup sequence, we will then derive a simple block sequence from it by taking only the blocks with valid signatures, as explained below. In order to define the rollup blocks, we will first define the set of accounts as

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

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 a set of L1 addresses which are used for withdrawing funds from the rollup to L1.

5.1 Registration blocks

Each user must register their public BLS key in a *registration block* before they can transact on the rollup. The user will then be assigned an L2 account, which is an integer which increments for each new account. When registering a new BLS public key, the user must prove that they know the corresponding private key to prevent the rogue key attack, which is done by including a signature of the BLS public key by the corresponding BLS private key in the registration block.

Definition 18 (Registration block) *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* σ *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 require every registration block to be valid, but will instead filter out the invalid blocks when deriving the simple block sequence.

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 $x \xleftarrow{R} \mathbb{Z}_q$
- 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, which is an integer that increments for each new account.

5.2 Transfer blocks

Transfer blocks are used for sending transactions on the rollup.

Definition 19 (Transfer block) *A* transfer block *is a tuple* $(senders, C, \sigma)$*, where* $senders \subset L2\_accounts$ *is a finite set of senders,* $C \in \mathcal{C}$ *is a transaction tree commitment, and* $\sigma \in \mathbb{G}_0$*.*

For a transfer block $(senders, C, \sigma)$ to be valid, σ must be a valid signature of C under the BLS public keys of the senders in $senders$, which are found in the registration blocks preceding the transfer block in the rollup sequence, as explained below. In the same way as for registration blocks, we will not require that σ is a valid BLS signature, and will instead just filter out the invalid transfer blocks when deriving the simple block sequence.

5.3 Deposit blocks

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

Definition 20 (Deposit block) *A deposit block is a tuple $(recipient, amount)$, where $recipient \in L2\_accounts$ and $amount \in Amounts$.*

5.4 Signature verification

To verify the signatures in the transfer blocks in a rollup sequence, we need to keep track of the BLS public keys assigned to each L2 account.

Definition 21 (Sequence of registered BLS public keys) *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 registered BLS public keys in B_* at index i by taking the BLS public keys in each of the $K(i)$ valid registration blocks in the rollup sequence in order, up to and including B_i.*

Definition 22 (Valid transfer block) *A transfer block $B_i = (senders, C, \sigma)$ in a rollup sequence B_* is* valid *if σ is a valid BLS signature of C by the BLS public keys $(pk_i(s))_{s \in senders}$, where $pk_i(*)$ are the registered BLS public keys in B_* at index i.*

We will derive a simple block sequence from a rollup sequence by taking all deposit blocks and all valid transfer blocks and converting them to simple blocks. In order to be able to convert both deposit blocks and transfer blocks to a simple block, we must extend the commitment scheme to allow committing to a deposit block. In details, we will replace the set $\mathcal{C}$ of possible commitments and the set $\mathit{\Pi}$ of inclusion proofs in the original commitment scheme by the disjoint union $\mathcal{C}' = \mathcal{C} \sqcup \mathcal{B}_{deposit}$ and the disjoint union $\mathit{\Pi}' = \mathit{\Pi} \sqcup \{trivial\}$ respectively, where $\mathcal{B}_{deposit}$ is the set of deposit blocks and $\{trivial\}$ is a one-set element consisting of a trivial proof. Then, a valid inclusion proof is either a valid inclusion proof in $\mathit{\Pi}$ if the commitment is a regular transaction map commitment, or the trivial proof if the commitment is a deposit block.

Definition 23 (Derived simple block sequence) *The* simple block sequence derived from a rollup sequence B_* *is defined by taking all valid deposit and transfer blocks in B_* in order, and converting each deposit block $(recipient, amount)$ to the simple block $(\{source\}, (recipient, amount))$ and each transfer block $(senders, C, \sigma)$ to the simple block $(senders, C)$.*

5.5 Zero knowledge balance proofs for a rollup sequence

To prove the balance of an account in a rollup sequence, we will combine a ZK proof that a simple block sequence was correctly derived from the rollup sequence with a ZK balance proof for the account in the simple block sequence.

We will first define the hash of a rollup sequence.

Definition 24 (Hash of a rollup sequence) *Given an encoding of rollup blocks* $enc_{rollup} : \mathcal{B} \to \{0,1\}^*$, *we define the* hash of a rollup sequence $B_* = (B_i)_{i=1}^N$ at index i, *written* $H_i(B_*)$ *inductively as follows. We define* $H_0(B_*) = Hash([0])$, *and for* $1 \leq i \leq N$ *we define* $H_i(B_*) = Hash(H_{i-1}(B_*) \parallel Hash(enc_{rollup}(B_i)))$. *For simplicity, we will write* $H(B_*) = H_N(B_*)$, *called the* hash of B_*.

We will use Circuit 2 for verifying that a simple block sequence is derived from a rollup sequence, which will be combined with Circuit 1 to get Circuit 3, used for proving the balance of an account in a rollup sequence.[12]

[12] By 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, we gain performance benefits, since the ZK proof that the simple block sequence was derived correctly only needs to be generated once and distributed to all users.

Circuit 2 Verifying that a simple block sequence is derived from a rollup sequence

Public input:

hashes $h, h' \in \{0,1\}^n$

a sequence $pk(*) = (pk(i))_{i=1}^{M}$ of BLS public keys

Private input:

hashes $prev\_h, prev\_h' \in \{0,1\}^n$

a sequence $prev\_pk(*) = (prev\_pk(i))_{i=1}^{prev\_M}$ of BLS 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 BLS public keys $prev\_pk(*)$

1: Verify $h = Hash(prev\_h \parallel Hash(enc_{rollup}(B)))$
2: Verify $h' = Hash(prev\_h' \parallel Hash(enc_{simple}(B')))$
3: Verify that if B is a valid registration block (pk, σ), then $pk(*)$ is obtained by adding pk to $prev\_pk(*)$.
4: Verify the ZK-proof P
5: **if** B is a transfer block $(senders, C, \sigma)$ where σ is a valid BLS signature of C by the BLS public keys $(pk(s))_{s \in senders}$ **then**
6: Verify $B' = (senders, C)$
7: **else if** B is a deposit block $(recipient, amount)$ **then**
8: Verify $B' = (\{source\}, (recipient, amount))$
9: **else**
10: Verify $h' = prev\_h'$
11: **end if**

Circuit 3 Verify 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 BLS public keys

a hash $h' \in \{0,1\}^n$

a ZK-proof P_1 of Circuit 2 proving that the rollup with hash h generates the BLS public keys pk_* and the derived simple block sequence with hash h'

a ZK-proof P_2 of Circuit 1 proving that *account* owns at least *balance* in the simple block sequence with hash h'

1: Verify P_1 and P_2

6 Implementing the design as a rollup

We turn our design into a ZK-rollup by deploying a rollup contract on an underlying blockchain. The rollup contract fixes a rollup, which we will refer to as *the rollup*, by storing the hashes of the rollup in its storage. It also has functions for adding new blocks to the rollup and for depositing funds from or withdrawing funds to the underlying blockchain. When a new block is added to the rollup, the new hash is computed by the rollup contract and added to its storage. Anyone is allowed to add blocks to the rollup, which gives high censorship resistance. When withdrawing funds 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 the amount of funds that has previously been withdrawn to the L1 address (which is also stored in the contract). In details, we have the following rollup contract state.

Definition 25 (Rollup contract state) *The* rollup contract state *is a tuple* $(h_*, total\_withdrawn\_amount)$ *where* $h_* = (h_i)_{i=0}^N$ *are the hashes of the rollup 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.*[13]

6.1 Adding blocks to the rollup

Anyone can add blocks to the rollup by calling Rollup contract function 1. This function can also be used to deposit funds, as explained in Protocol 2.

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

Require:
A rollup block B
The amount $included\_amount \in Amounts$ included with the L1 transaction
The current sequence $h_* = (h_i)_{i=1}^N$ of hashes stored in the rollup contract

Ensure: h_* is the new sequence of hashes stored in the rollup contract.

1: **if** $B = (recipient, amount)$ is a deposit block and $amount = included\_amount$ or if B is a non-deposit block **then**
2: $h_* \leftarrow (h_0, h_1, \ldots, h_N, Hash(h_N \parallel enc_{rollup}(B)))$
3: **end if**

[13] We note that in order to save space, instead of storing all previous hashes of the rollup in the contract storage, we can store only the n newest hashes, where n may be increased by anyone willing to pay the storage costs. Also, if more than one rollup block is added in an L1 block, we only need to store the last hash. With these features, each hash is guaranteed to be stored in the contract storage for the duration of at least $t \cdot n$, where t is the minimum time between each L1 block, and n is the number of hashes in the storage.

Protocol 2 Deposit Protocol.

Depositing to rollup

In order to transact on the rollup, 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 and the amount of each token to be deposited.
- Alice submits the deposit block to the rollup smart contract together with the specified amount of each token.

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.

6.2 Withdrawing from the rollup

A user can withdraw from the rollup to an L1 address by calling Rollup contract function 2.[14] We refer the reader to Protocol 3, where we describe an overview of the withdrawal protocol.

Rollup contract function 2 Withdrawing from the rollup.

Require:
A hash $h \in \{0,1\}^n$.
An index $0 \leq i \leq N$
An address $a \in L1\_addresses$ which we will withdraw to.
An amount $balance \in Amounts$.
A ZK-proof P of Circuit 3 proving that a has a balance of at least $balance$ in the rollup sequence with hash h.
The current state of $total\_amount\_withdrawn$ in the rollup contract storage.

Ensure:
$total\_amount\_withdrawn$ is the new state of the total amount withdrawn.
$withdrawn\_amount \in Amounts$ is the amount to be withdrawn.

if $h = h_i$ and if P is a valid ZK-proof **then**
    $withdrawn\_amount \leftarrow balance - total\_amount\_withdrawn[a]$
else
    $withdrawn\_amount = 0$
end if
$total\_amount\_withdrawn[a] \leftarrow total\_amount\_withdrawn[a] + withdrawn\_amount$

[14] We mention that this can be made more efficient by batching many ZK balance proofs together into one proof used to withdraw to many L1 addresses.

Protocol 3 Withdrawal Protocol.

Withdrawing funds

To withdraw funds, user Alice performs the following steps:

- Alice sends in a transfer block the desired amount to be withdrawn from her L2 account to the rollup account representing her L1 account
- 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 rollup: $\mathsf{Prove}(pp, x, w) \to P$.
- 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: $\mathsf{Verify}(pp, x, P) \to \mathsf{Accept}$
- Checks the provided rollup hash is in the list of previous rollup hashes.
- Transfers to the L1 account (on L1) the difference between the proven balance of the rollup account representing her L1 address and the amount that has previously been withdrawn to the L1 address, and updates the total amount withdrawn to the L1 address accordingly in the contract storage.

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.

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 20, 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 20, 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 20, 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 20, 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 syn-

chronization 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 6. *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 7. *Given a commitment $\mathcal{C}$ and a transaction tx such that $\mathsf{Commit}(\mathsf{tx}) \to \mathcal{C}$, $\mathcal{A}$ 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 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 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 9. *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.