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

Erik Rybakken[1], Leona Hioki[1], Mario Yaksetig[2], Denisa Diaconescu[3], František Silváši[3], and Julian Sutherland[3]

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

Abstract. We present a blockchain scaling solution called Intmax2, which is a Zero-Knowledge rollup (ZK-rollup) protocol with stateless and permissionless 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. We give a proof of the main security property of the protocol, which has been formally verified by the Nethermind Formal Verification Team in the Lean theorem prover.

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 (L2) 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:

- Uses less onchain data than any existing rollup, giving an upper limit of 7500 transaction batches per second on Ethereum, where each transaction batch can transfer an unlimited number of tokens to an unlimited number of recipients.
- Offers permissionless block production.
- Provides stronger privacy properties than traditional ZK-rollups.

1.3 Formal verification of the security proof

We give a pen-and-paper proof of the security of the protocol in Theorem 1. This proof has been formally verified in the Lean theorem prover[16] by the Nethermind Formal Verification team [2]. All mathematical definitions and statements that are needed for the security proof contain a hyperlink (like this: ↗) to the formalized version.

2 Simplified design description

In this section we describe a simplified version of the design which doesn't use ZK-proofs. The simplified design achieves low onchain data consumption (4-5 bytes per transaction sender), but is otherwise inefficient and not private. In Section 4 we add ZK-proofs to the design in order to achieve efficiency and privacy.

2.1 Overview

The simplified design works roughly as follows. At the heart of the design is a rollup contract deployed on a programmable blockchain (such as Ethereum). To deposit funds to the rollup, a user simply sends the funds, together with the L2 address of the recipient, to the rollup contract which then records the deposit in its contract storage. To transfer funds on the rollup, a subset of L2 accounts will first send their transactions to a single aggregator, which then inserts the transactions at the leaves of a merkle tree. Then the aggregator sends to each sender the merkle root and the merkle proof for that sender's transaction. Each sender then signs the merkle root with their public BLS key and sends this signature back to the aggregator. The aggregator then aggregates the signatures into a single aggregated signature, and sends the merkle root, the aggregated signature and the list of public keys of the senders that was included in the aggregated signature to the rollup contract. The rollup contract then verifies the signature and adds the root, signature and sender list to its storage. Each sender is then responsible for sending the merkle proof of the transaction to each transaction recipient offchain, together with earlier merkle proofs that together prove that the sender had sufficient balance for the transaction. To prove their own balance, each user needs to keep track of all merkle proofs they have received from aggregators and other users. This collection of merkle proofs, called a balance proof, is sent to the rollup contract when a user wants to withdraw their funds to L1.
We now describe the simplified design in more details.

2.2 Notation

If X and Y are sets, we will write Y^X for the set of all functions from X to Y. We will often call a function $f \in Y^X$ a *mapping* from (elements of) X to (elements of) Y.

2.3 Setup

The design depends on an authenticated dictionary scheme[4] AD, a signature aggregation scheme[5] SA, and a collision-resistant hash function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^n$. Given a security parameter $\lambda \in \mathbb{N}$ we set up the authenticated dictionary scheme

$$\mathsf{AD}.(\mathcal{K}, \mathcal{M}, \mathcal{C}, \Pi, \mathsf{Commit}, \mathsf{Verify})$$

and the signature aggregation scheme

$$\mathsf{SA}.(\mathcal{K}_p, \mathcal{K}_s, \Sigma, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Aggregate}, \mathsf{Verify}).$$

We use the alias $\mathcal{K}_2 := \mathsf{SA}.\mathcal{K}_p$ and call this the set of *L2 accounts*. We also depend on a set $\mathcal{K}_1$ of L1 accounts[6], and a lattice-ordered abelian group $\mathcal{V}$ which is used as the set of transaction values and account balances.[7] We denote by $\mathcal{V}_+ \subseteq \mathcal{V}$ the subset of positive values, defined as the values $v \in \mathcal{V}$ where $v \geq 0$.

2.4 Rollup contract

The rollup contract is a smart contract deployed on a programmable blockchain (e.g. Ethereum), which is responsible for keeping track of the rollup state and for managing deposits, transfers, and withdrawals. The internal state of the rollup contract consists of the list of all blocks that have been added to the rollup so far. There are three types of blocks in our design, namely deposit blocks, transfer blocks and withdrawal blocks, denoted by $\mathcal{B}_{deposit}$, $\mathcal{B}_{transfer}$ and $\mathcal{B}_{withdrawal}$ respectively. We formally define these sets below where we describe the respective protocols. Letting $\mathcal{B} := \mathcal{B}_{deposit} \amalg \mathcal{B}_{transfer} \amalg \mathcal{B}_{withdrawal}$ be the set of all blocks , the contract state is formally defined as

$$\mathcal{S}_{contract} := \mathcal{B}^*.$$

When the rollup contract is deployed to the blockchain, it is initialized with the state () consisting of the empty list .

2.5 Depositing

To deposit funds from L1 to L2, a L1 user will simply send the funds to the rollup contract along with the L2 address of the recipient. The rollup contract

[4] See Appendix A.2.

[5] See Appendix A.3.

[6] For instance, in Ethereum, accounts are represented by 20 bytes, so in this case we have $\mathcal{K}_1 := \{0,1\}^{20 \cdot 8}$.

[7] See Appendix A.5 for the definition of a lattice-ordered abelian group. This generality allows us to easily support transfers of multiple value types (e.g. NFTs, ERC20 tokens, etc.) by letting $\mathcal{V}$ be the set of mappings from a set of token names to the set $\mathbb{Z}$ of integers, which naturally gets the structure of a lattice-ordered abelian group.

then constructs a *deposit block* which consists of the specified recipient and the deposited amount. Formally, we define the set of deposit blocks as

$$\mathcal{B}_{deposit} := \mathcal{K}_2 \times \mathcal{V}_+.$$

The contract then adds this deposit block to the list of blocks in its storage.

2.6 Transferring

We now describe the protocol for transferring funds on the rollup (illustrated in Figure 1). To transfer funds from an L2 account, the account owner will first construct a *transaction batch*, which is a mapping that maps each transaction recipient to the amount the sender wants to send to that recipient. A transaction recipient is either an L2 account or an L1 account (used when withdrawing to L1, as described in Section 2.7). Formally, letting $\mathcal{K} := \mathcal{K}_1 \amalg \mathcal{K}_2$, a transaction batch is an element of $\mathcal{V}_+^{\mathcal{K}}$, i.e. a mapping from $\mathcal{K}$ to V_+. Suppose we have a set of senders $S \subseteq \mathcal{K}_2$ where each sender $s \in S$ has a secret key sk_s and a transaction batch $t_s \in \mathcal{V}_+^{\mathcal{K}}$ they want to send. The transfer protocol consists of two phases. In the first phase, the senders collaborate with a single aggregator to produce a *transfer block* which is added to the rollup contract. In the second phase, after the transfer block has been added to the rollup contract, each transaction sender s will send (offchain) to each recipient (i.e. accounts $r \in \mathcal{K}$ where $t_s(r) \neq 0$) the data needed to prove that the sender sent the specified amount to the recipient in the transfer block. We now describe the two phases of the transfer protocol in more details.

Phase 1: Constructing and adding a transfer block To send the transaction batches, the senders will first select a single aggregator[8] and agree upon a common bitstring $extradata \in \{0,1\}^*$. This bitstring can be used to implement protections against replay attacks and delayed block publication (see Section 2.8). Then the senders and aggregator interacts in the following protocol.

1. First, each sender s chooses a random salt $salt_s$, hashes their transaction batch with the salt
$$h_s \leftarrow H(t_s, salt_s),$$
and sends h_s to the aggregator.[9]
2. The aggregator collects all the transaction batch hashes from the senders. Let $S' \subseteq S$ be the subset of senders who sent a transaction batch hash to the aggregator. The aggregator then constructs the dictionary[10] (S', h)

[8] The protocol allows anyone to be an aggregator for a transfer block, enabling maximum censorship resistance.

[9] Sending the transaction hash instead of the transaction itself gives privacy from the aggregator.

[10] See Appendix A.1 for the definition of a dictionary.

where h_s is the transaction batch hash by s for all $s \in S'$, and constructs a dictionary commitment and lookup proofs:

$$(C, (S', \pi)) \leftarrow \mathsf{AD.Commit}(S', h).$$

The aggregator sends to each sender $s \in S'$ the dictionary commitment C and the lookup proof π_s for the sender's transaction batch hash.

3. Upon receiving the dictionary commitment and lookup proof, each sender s checks if the lookup proof is valid with the commitment:

$$\mathsf{AD.Verify}(\pi_s, s, h_s, C) \stackrel{?}{=} True.$$

If the lookup proof is valid, the sender generates the signature

$$\sigma_s \leftarrow \mathsf{SA.Sign}(sk_s, (C, aggregator, extradata))$$

and sends this signature to the aggregator.

4. The aggregator collects the signatures from the senders and verifies them. Let $S'' \subseteq S'$ be the subset of senders who sent a valid signature. The aggregator then constructs the aggregated signature

$$\sigma \leftarrow \mathsf{SA.Aggregate}((s, \sigma_s)_{s \in S''}),$$

and constructs the tuple $(aggregator, extradata, C, S'', \sigma)$, called a *transfer block*. Formally, we define the set of transfer blocks as

$$\mathcal{B}_{transfer} = \mathcal{K}_1 \times \{0,1\}^* \times \mathsf{AD}.\mathcal{C} \times \mathcal{P}(\mathcal{K}_2) \times \mathsf{SA}.\Sigma.$$

The aggregator sends this transfer block to the rollup contract using their L1 account.

5. Upon receiving the transfer block, the rollup contract verifies the aggregated signature:

$$\mathsf{SA.Verify}(S'', (C, aggregator, extradata), \sigma) \stackrel{?}{=} True$$

and also verifies that the transaction is indeed coming from the account $aggregator$. If these checks are valid, the contract adds the transfer block to the list of blocks in its storage. If not, the transaction is reverted.

Phase 2: Maintaining and distributing balance proofs To be able to prove the balance of their account, each user needs to maintain a *balance proof*, which is the collection of transaction batches with corresponding salts and lookup proofs that they have received from an aggregator (when sending transactions) and from other users (when receiving transactions). We formally define the set of balance proofs as

$$\mathit{\Pi} := Dict(\mathsf{AD}.\mathcal{C} \times \mathcal{K}_2, (\mathsf{AD}.\mathit{\Pi} \times \{0,1\}^*) \times \mathcal{V}_+^{\mathcal{K}}).$$

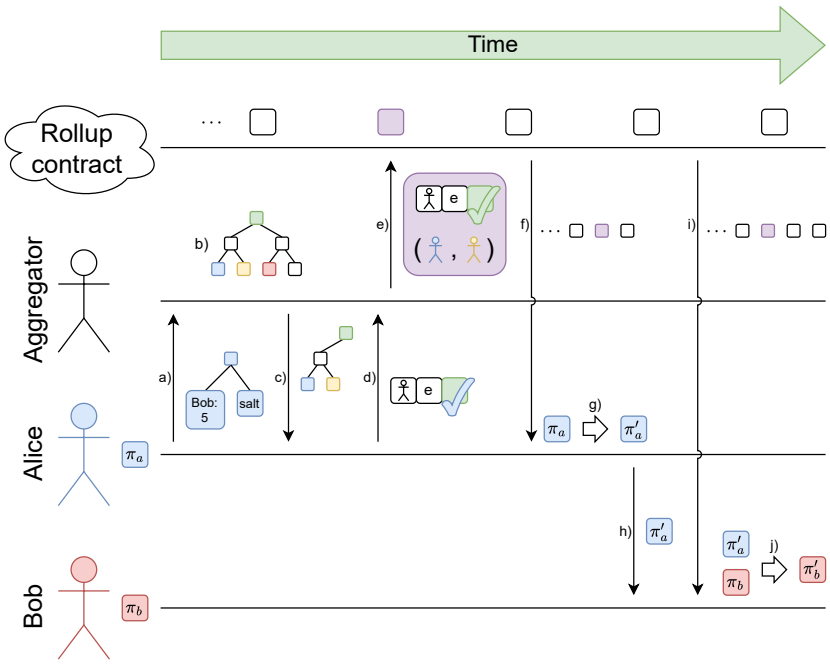

Fig. 1. The transfer protocol. In this example, Alice wants to send 5 coins to Bob. a) Alice starts by sending the hash of the the transaction batch, consisting of a single transaction of 5 coins to Bob, and a random salt to an aggregator. b) The aggregator then constructs a merkle tree consisting of Alice's transaction batch hash and the transaction batch hashes of other senders. c) The aggregator sends the merkle proof of Alice's transaction batch to Alice. d) Alice verifies the merkle proof and signs the merkle root together with the pre-determined extradata e. This signature is sent back to the aggregator. e) The aggregator collects the signatures from all senders, constructs the transfer block, and sends it to the rollup contract. f) Alice watches the blocks that are added to the rollup contract until the block containing her transaction is published. g) Alice updates her balance proof by adding her transaction batch, the salt and the merkle proof. h) Alice sends to Bob her updated balance proof. h) Bob updates his view of the rollup blocks. i) Bob updates his balance proof by merging it with the balance proof he received from Alice.

A balance proof is valid if the following algorithm returns $True$.

$$\begin{aligned}&\mathsf{Verify}\colon \mathit{\Pi} \to \{True, False\}\\ &(K, D) \mapsto \bigwedge_{\substack{(C,s)\in K\\((\pi,salt),t)=D(C,s)}} \mathsf{AD.Verify}(\pi, s, \mathsf{H}(t, salt), C)\end{aligned}$$

In other words, a valid balance proof is a dictionary which maps commitment-sender pairs (C, s) to tuples $((\pi, salt), t)$ where $t \in \mathcal{V}_+^{\mathcal{K}}$ is a transaction batch, $salt$ is a random salt and $\pi \in \mathsf{AD}.\mathit{\Pi}$ is a valid lookup proof that $H(t, salt)$ is the value at index s in an authenticated dictionary with commitment C.
Each user will maintain a balance proof, which is initialized as the empty dictionary. In the second phase of the transfer protocol, each transaction sender will add their transaction batch with the corresponding lookup proof they received from the aggregator (if they did receive one) to their own balance proof. Then, they will send this new balance proof to each recipient of the transaction batch. Upon receiving the balance proof, each recipient will then merge it with their own. In details, if $\pi_b \in \mathit{\Pi}$ is the current balance proof of a recipient named Bob and $\pi_a \in \mathit{\Pi}$ is the balance proof they received from a sender named Alice, then Bob performs the following algorithm:

- Verify the received balance proof:

$$\mathsf{Verify}(\pi_a) \stackrel{?}{=} True.$$

  If valid, continue to the next step, otherwise terminate.
- Update their balance proof π_b by merging it with π_a:

$$\pi_b \leftarrow \mathsf{Merge}(\pi_a, \pi_b),$$

  where Merge is the dictionary merging algorithm defined in Appendix A.1.

To compute the balance of their accounts, users will use the balance function

$$\mathsf{Bal}\colon \mathit{\Pi} \times \mathcal{B}^* \to \mathcal{S},$$

defined in Appendix B. Here $\mathcal{S}$ is the set of states, where a state is an assignment of a balance to each account. Formally, let $\overline{\mathcal{K}} := \mathcal{K}_1 \amalg \mathcal{K}_2 \amalg \{\text{Source}\}$, where Source is a special account used to represent deposits and withdrawals. Then the set of states $\mathcal{S}$ is defined as

$$\mathcal{S} := \{b \in \mathcal{V}^{\overline{\mathcal{K}}}, \text{ such that } b_k \geq 0, \forall k \in \overline{K} \backslash \{\text{Source}\}\}.$$

The balance function takes a balance proof $\pi \in \mathit{\Pi}$ and the current state of the rollup contract $(B_*) \in \mathcal{B}^*$, and returns the balance of each account in the rollup that can be proven by the balance proof.

2.7 Withdrawing

When a user wants to withdraw funds from their L2 account to an L1 account, they must first transfer the funds to the L1 account using the transfer protocol described above. When the transfer block is added to the rollup contract, the contract does not automatically withdraw the funds to L1. Instead, to initiate the withdrawal, the owner of the L1 account must send a withdrawal request to the rollup contract which consists of the user's current balance proof $\pi \in \Pi$. Upon receiving the balance proof, the rollup contract performs the following steps:

- First, the balance proof is verified:

$$\mathsf{Verify}(\pi) \stackrel{?}{=} True.$$

- If the balance proof is valid, the rollup contract constructs a withdrawal block, which is simply the in-rollup balance of each L1 account computed from the balance proof and the current rollup state:

$$B \leftarrow \mathsf{Bal}(\pi, B_*)|_{K_1},$$

  where B_* is the current list of blocks in the rollup contract. Formally, the set of withdrawal blocks is defined as

$$\mathcal{B}_{withdrawal} = \mathcal{V}_+^{K_1}.$$

- The contract adds the withdrawal block B to the list of blocks in its storage:

$$B_* \leftarrow (B_*||(B))$$

- For each L1 account $k \in K_1$, the contract withdraws the amount B_k to the L1 account.

These steps ensure that a user cannot double-spend by withdrawing the same funds twice, since the amount to be withdrawn is computed by applying the balance function to the balance proof *and all previous blocks that have been added to the rollup*, which includes all previous withdrawal blocks. This is formalised in Theorem 1, which is stated and proven in Appendix C.

2.8 Protection against replay attacks and delayed block publication

There are a couple of attacks that a malicious aggregator can do that we need to protect against. One kind of attack is delayed block publication, where a malicious aggregator waits a long time before publishing the transfer block, causing a liveness issue. A second attack is replay attacks, where a malicious aggregator publishes the same transfer block multiple times, thereby draining the balances of the senders. Instead of adding protections against these attacks in-protocol, it can be done out-of-protocol as follows.

In order to make users trust them, an aggregator can self-impose restrictions that prohibits them from performing these attacks by deploying a *relayer contract* on L1. When a set of senders wants to create a transfer block with this aggregator, the aggregator will first pick a deadline for the transfer block not far into the future. The senders who accepts the deadline enters the transfer protocol using this deadline as *extradata*, and the relayer contract address as *aggregator*. After constructing the transfer block, the aggregator will send it to the relayer contract from an L1 address which is whitelisted by the contract (for front-running protection). Upon receiving the transfer block, the relayer contract verifies the sender and checks if the deadline in the *extradata* field is no later than the current time, before forwarding the transfer block to the rollup contract. This protects against delayed block publication. In addition, the relayer contract stores the timestamp of the last block that it has forwarded to the rollup contract, and verifies that each new transfer block has a timestamp strictly greater than the last forwarded block before forwarding it. This protects against replay attacks.

3 Data usage and compression

In this section we analyze the scalability of our design and describe how to add compression to achieve even more scalability. The main bottleneck for the scalability is the size of the transfer blocks, which is decomposed as follows:

- The aggregator's L1 address (20 bytes in Ethereum)
- A *extradata* string (32 bytes)
- An authenticated dictionary commitment (32 bytes if it is a merkle tree root)
- The subset of senders $S \subseteq \mathcal{K}_2$ in the block ($|S| \times 96$ bytes if encoded as a list of BLS public keys)
- An aggregated signature (48 bytes for BLS signatures)

This gives a transfer block size of $|S| \times 96 + 132$ bytes, where $|S|$ is the number of senders in the block. This is smaller than for traditional rollups where all transaction details (such as sender, recipient and transaction amount) are included in the blocks. Also, unlike traditional rollups, our block size only depends on the number of senders, and not the number of transactions. This means that a sender can send a transaction batch with an arbitrary number of recipients without affecting the size of the transfer block.
To further increase scalability, we can add block compression out-of-protocol using the relay contracts we introduced in Section 2.8, where an aggregator sends compressed transfer blocks to their relay contract, which will decompress the blocks before relaying them to the rollup contract. A simple compression algorithm works as follows. Users can register their public BLS key with the relay contract of an aggregator and receive a short incremental ID. The relay contract stores in its storage a dictionary which maps IDs to BLS public keys. Then, when the aggregator sends transfer blocks to the relay contract, they

will send the short IDs of the senders instead of their public keys. The relay contract looks up each ID in its dictionary and reconstructs the transfer block with the public keys before sending it to the rollup contract. The size of the IDs depends on the total number of IDs in the dictionary. As an example, in order to support 10 billion addresses (more than the current world population), each ID must be

$$\log_2(10^9) \approx 33 \text{ bits} \approx \ 4.15 \text{ bytes},$$

which gives a block size of about $|S| \times 4.15 + 132$ bytes. When implemented on Ethereum, which provides 0.375 MB of data per block[19], with blocks coming every 12 seconds[1], we get a theoretical limit of about

$$\frac{0.375 \times 10^6 - 132}{4.15} \approx 90000$$

senders per L1 block, or 7500 senders per second. This number will increase when Ethereum adds more scaling. According to [19], the goal is to achieve ≈ 16 MB per block, which would allow ≈ 320000 senders per second.

4 Adding privacy and efficiency

The simplified design described in Section 2 lacks privacy, because transaction recipients will gain information about other transactions not intended for them, and it lacks efficiency because the balance proofs are large and expensive to verify (especially onchain during withdrawals). In this section, we add privacy and efficiency using recursive ZK-proofs.

4.1 Changes to rollup contract state and the procedure of adding blocks

First, the rollup contract is modified so that instead of storing the list of all blocks added to the rollup, it stores a list of *history roots*, where each root is a hash digest in $\{0,1\}^n$, as well as a mapping which maps each L1 account to the total amount that has been withdrawn to the L1 account:

$$\mathcal{S}_{contract} := (\{0,1\}^n)^* \times \mathcal{V}_+^{\mathcal{K}_1}.$$

If $((root_i)_{i\in[N]}, withdrawn)$ is the current state of the contract, and $B \in \mathcal{B}$ is a new block to be added to the rollup, the contract adds the new block as follows. If the block is a deposit block or a transfer block, the contract computes a new history root by taking the hash $\mathsf{H}(root_N, B)$ of the most recent history root and the new block, and adds this new history root to its list of history roots. If the new block is a withdrawal block, the withdrawn amounts are added to the current map of withdrawn amounts:

$$withdrawn \leftarrow withdrawn + B.$$

4.2 Changes to the transfer protocol

Phase 1 of the transfer protocol regarding how to construct and add transfer blocks remains exactly as described in Section 2.6, but Phase 2 regarding how to maintain and distribute balance proofs is changed as follows. When a transaction sender s sends funds to a recipient r, instead of providing the recipient with the complete transaction history of the sender and recursively those of other users (as in the simplified design), they will only send the tuple $(root, s, r, v, \pi)$ where

- $root \in \{0,1\}^n$ is the history root of the rollup block containing the transaction,
- $s \in \mathcal{K}_2$ is the sender's L2 address,
- $r \in \mathcal{K}$ is the recipient's address,
- $v \in \mathcal{V}_+$ is the transaction amount,
- π is a transaction validity proof, which is a ZK-proof proving that the sender s did send a transaction with value v to the recipient r in the rollup block with history hash $root$, and that the sender had a sufficient balance for sending it.

This means that the recipient only learns about this transaction, and gets zero knowledge about anything else, such as the balance of the sender or other transactions.
To be able to construct transaction validity proofs, each user needs to maintain the data consisting of

- All transaction batches they have sent (that have been included in a transfer block onchain) together with their corresponding salts and lookup proofs
- All verified transactions $(hash, s, r, v, \pi)$ they have received from other users

Given this data, as well as the list of blocks added to the rollup[11], each user can generate validity proofs for their transactions.

4.3 Changes to the withdrawal protocol

When the owner of an L1 account wants to withdraw their in-rollup balance to L1, they will send a withdrawal request to the rollup contract which consists of an L1 address $address \in \mathcal{K}_1$, a value $v \in \mathcal{V}_+$, a history root $root \in \{0,1\}^n$ and a ZK-proof that $address$ has received at least v in the rollup at history root $root$. Upon receiving the withdrawal request, the rollup contract will verify that the ZK-proof is valid and that $root$ is in the list of history roots in its storage. If these checks are valid, the contract will compute the in-rollup balance of the L1 account by subtracting the previously withdrawn amount of the address from v. Then, the contract withdraws the computed balance to L1 and updates the total amount withdrawn in its storage.

[11] This can be obtained by monitoring all L1 transactions sent to the rollup contract.

5 Conclusion

We presented Intmax2, a novel ZK-rollup approach that completely shifts away from traditional ZK-rollup approaches. In contrast with previous approaches, our solution does not require the posting of all transaction data on the underlying L1, which enables unprecedented scalability. By leveraging the fact that aggregators do not need to perform computationally intensive zero-knowledge proofs, and instead moving the computation on the side of the users in the system, our design provides a novel, practical, and resilient solution to L2 scaling. On a final note, by making the aggregator role completely permissionless, our design allows for a much more censorship-resistant solution, thus addressing one of the main existing problems in the rollup space.

References

1. Blocks: Block time. Ethereum Development Documentation, https://ethereum.org/en/developers/docs/blocks/#block-time, accessed: January 29, 2025
2. Nethermind formal verification team. https://www.nethermind.io/formal-verification
3. 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
4. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: Peyrin, T., Galbraith, S. (eds.) Advances in Cryptology – ASIACRYPT 2018. pp. 435–464. Springer International Publishing, Cham (2018)
5. 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
6. 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
7. 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
8. Dahlberg, R., Pulls, T., Peeters, R.: Efficient sparse merkle trees. In: Brumley, B.B., Röning, J. (eds.) Secure IT Systems. pp. 199–215. Springer International Publishing, Cham (2016)
9. Dompeldorius, A.: Springrollup. https://github.com/adompeldorius/springrollup (2021), accessed: January 29, 2025
10. 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
11. 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
12. 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
13. 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
14. 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: January 29, 2025
15. 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

16. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: 2021 Conference on Automated Deduction. pp. 625–635. Springer, Cham (July 2021), https://www.microsoft.com/en-us/research/publication/the-lean-4-theorem-prover-and-programming-language/
17. 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
18. team, B.F.: Plasma prime design proposal. https://ethresear.ch/t/plasma-prime-design-proposal/4222, accessed: January 29, 2025
19. (@vbuterin), V.B., (@dankrad), D.F., (@protolambda), D.L., (@asn d6), G.K., (@lightclient), M.G., (@Inphi), M.T., (@adietrichs), A.D.: Eip-4844: Shard blob transactions [draft]. Ethereum Improvement Proposals, no. 4844 (2022), https://eips.ethereum.org/EIPS/eip-4844

A Background

A.1 Dictionaries

Definition 1 *Let X be a set. We define*

$$Maybe(X) := X \amalg \{\bot\}.$$

Definition 2 *Given two sets X and Y, we define*

$$Dict(X, Y) := Maybe(Y)^X.$$

Elements of $Dict(X, Y)$ are called dictionaries over (X, Y).

Remark 1 *A dictionary over (X, Y) is also often called a* partial function *from X to Y.*

Dictionaries can be combined as follows.

Definition 3 *Let X be a set. We define*

$$\begin{aligned} \mathsf{First}\colon Maybe(X) \times Maybe(X) &\to Maybe(X) \\ (x_1, x_2) &\mapsto \begin{cases} x_1, & \text{if } x_1 \neq \bot \\ x_2, & \text{otherwise.} \end{cases} \end{aligned}$$

Definition 4 *Let X and Y be sets. We define*

$$\begin{aligned} \mathsf{Merge}\colon Dict(X, Y) \times Dict(X, Y) &\to Dict(X, Y) \\ (D_1, D_2) &\mapsto D, \\ &\text{where } D(x) := First(D_1(x), D_2(x)),\ \forall x \in X. \end{aligned}$$

A.2 Authenticated dictionaries

Definition 5 (Authenticated dictionary scheme) *An authenticated dictionary scheme over a key set $\mathcal{K}$ and value set $\mathcal{M}$ consists of sets*

- *$\mathcal{C}$ of commitments*
- *Π of lookup proofs*

and algorithms

- $\mathsf{Commit} : Dict(\mathcal{K}, \mathcal{M}) \to \mathcal{C} \times Dict(\mathcal{K}, \Pi)$
- $\mathsf{Verify} : \Pi \times \mathcal{K} \times \mathcal{M} \times \mathcal{C} \to \{True, False\}$

parameterized over a security parameter $\lambda \in \mathbb{N}$.

An authenticated dictionary scheme should satisfy correctness and binding, defined as follows.

Definition 6 (Correctness) *An authenticated dictionary scheme is* correct *if for all dictionaries $D \in Dict(\mathcal{K}, \mathcal{M})$ we get*

$$(C, \pi) \leftarrow \mathsf{Commit}(D)$$

such that

$$\mathsf{Verify}(\pi_k, k, D_k, C) = True,\ \forall k \in \mathcal{K},\ D_k \neq \perp.$$

Definition 7 (Binding) *An authenticated dictionary scheme is* binding *if it is computationally infeasible to find a commitment $C \in \mathcal{C}$, a key $k \in \mathcal{K}$, values $m_1, m_2 \in \mathcal{M}$ and lookup proofs $\pi_1, \pi_2 \in \Pi$ such that*

$$\begin{aligned} &\mathsf{Verify}(\pi_1, k, m_1, C) = True \\ \wedge\, &\mathsf{Verify}(\pi_2, k, m_2, C) = True \\ \wedge\, &m_1 \neq m_2. \end{aligned}$$

Implementation A common implementation of an authenticated dictionary scheme is sparse merkle trees [8], where the binding property is achieved by using a collision-resistant hash function.

A.3 Signature aggregation

A signature aggregation scheme consists of sets

- $\mathcal{K}_p$ of public keys
- $\mathcal{K}_s$ of secret keys
- Σ of signatures

and algorithms

- $\mathsf{KeyGen} : 1 \to \mathcal{K}_p \times \mathcal{K}_s$
- $\mathsf{Sign} : \mathcal{K}_s \times \mathcal{M} \to \Sigma$
- $\mathsf{Aggregate} : (\mathcal{K}_p \times \Sigma)^* \to \Sigma$
- $\mathsf{Verify} : \mathcal{K}_p^* \times \mathcal{M} \times \Sigma \to \{True, False\}$

parameterized over a security parameter $\lambda \in \mathbb{N}$.
A signature aggregation scheme should satisfy correctness and unforgeability, defined as follows.

Definition 8 *A signature aggregation scheme is* correct *if whenever we have a list of key-pairs $(pk_i, sk_i)_{i\in[n]}$ generated by the* KeyGen *algorithm, and a message $m \in \mathcal{M}$, we have*

$$\mathsf{Verify}((pk_i)_{i\in[n]}, m, \mathsf{Aggregate}((pk_i, \mathsf{Sign}(sk_i, m))_{i\in[n]})) = True.$$

Definition 9 *A signature aggregation scheme is* unforgeable *if it is computationally infeasible for an adversary to output a list $(pk_i)_{i\in[n]}$ of public keys, a message $m \in \mathcal{M}$ and a signature $\sigma \in \Sigma$ such that*

$$\mathsf{Verify}((pk_i)_{i\in[n]}, m, \sigma) = True,$$

and where one of the public keys $(pk_i)_{i\in[n]}$ belongs to an honest user who didn't sign the message m with their secret key.

Implementation We will use the modified BLS signature scheme introduced in [4], which is defined as follows.
Given a security parameter $\lambda \in \mathbb{N}$, we setup a bilinear pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ of groups of prime order q, and two hash functions $\mathcal{H}_0 : \mathcal{M} \to \mathbb{G}_0$ and $\mathcal{H}_1 : \mathcal{M} \to \mathbb{Z}_q$. We then let

- $\mathcal{K}_p := \mathbb{G}_1$
- $\mathcal{K}_s := \mathbb{Z}_q$
- $\Sigma := \mathbb{G}_0$

and

- $\mathsf{KeyGen}() \to (sk, pk)$, where the secret key is a random value $\mathsf{sk} \xleftarrow{R} \mathbb{Z}_q$ and the public key is $pk \leftarrow g_1^{\mathsf{sk}} \in \mathbb{G}_1$
- $\mathsf{Sign}(sk, m) \to \sigma$, where $\sigma \leftarrow \mathcal{H}_0(m)^{\mathsf{sk}} \in \mathbb{G}_0$.
- $\mathsf{Aggregate}((pk_1, \sigma_1), \ldots, (pk_n, \sigma_n)) \to \sigma$, where

$$\sigma \leftarrow \prod_{i \in [n]} \sigma_i^{t_i},$$

  and where $t_i \leftarrow \mathcal{H}_1(pk_i, \{pk_1, \ldots, pk_n\})$ for all $i \in [n]$.
- $\mathsf{Verify}((pk_1, \ldots, pk_n), m, \sigma)$ is computed by first computing the aggregated public key as

$$pk \leftarrow \prod_{i \in [n]} pk_i^{t_i},$$

  where $t_i \leftarrow \mathcal{H}_1(pk_i, (pk_j)_{j \in [n]})$ for all $i \in [n]$. Then, output $True$ if

$$e(g_1, \sigma) = e(pk, \mathcal{H}_0(m))$$

  and output $False$ otherwise.

A.4 Zero-knowledge proofs

Zero-knowledge proofs, introduced in [12], 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:

- $\mathsf{ZK.Setup}(\lambda) \to pp$. For a certain security parameter λ, the setup algorithm outputs pp, the public parameters of the system.
- $\mathsf{ZK.Prove}(pp, x, w) \to P$. Given the system's public parameters pp, a statement x, and a witness w, issue a proof P.
- $\mathsf{ZK.Verify}(pp, x, P) \to \{True, False\}$. Upon receiving the public parameters pp, the public statement x and the proof P, the verifier $\mathcal{V}$ either accepts (returns $True$) or rejects (returns $False$) 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 [5,6,10,13,15]. Quantum-secure similar constructions exist, as in [3,7].

A.5 Order theory

We here restate some common definitions and results from order theory, which are used in our protocol description and security proof.

Prosets, posets and setoids

Definition 10 *Let X be a set. A* preorder on X *is a binary relation $\leq$ on X such that*

- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ *for all $a, b, c \in X$ (transitivity),*
- $a \leq a$ *for all $a \in X$ (reflexivity).*

A preordered set*, or* proset*, is a tuple $(X, \leq)$ where X is a set and $\leq$ is a preorder on X.*

Remark 2 *If $(X, \leq)$ is a proset, we will denote by $\geq$ the opposite relation:*

$$a \geq b \Leftrightarrow b \leq a.$$

Definition 11 *Let $(X, \leq)$ be a proset. We say that two elements $a, b \in X$ are* isomorphic*, written $a \simeq b$, if $a \leq b \wedge a \geq b$.*

Definition 12 *Let $(X, \leq)$ be a proset. We say that the relation $\leq$ is*

- *a partial order if $a \simeq b \Leftrightarrow a = b$ for all $a, b \in X$, and*
- *an equivalence relation if $a \leq b \Leftrightarrow a \simeq b$ for all $a, b \in X$.*

We call $(X, \leq)$ a

- partially ordered set*, or* poset*, if $\leq$ is a partial order, and a*
- setoid *if $\leq$ is an equivalence relation.*

Examples of preorders We now define various preorders used in the paper.

Definition 13 *Let X be a set. The* trivial preorder on X *is the preorder $\leq$ where $a \leq b$ for all $a, b \in X$, i.e. every pair of elements of X are related.*

Definition 14 *Let X be a set. The equality relation $=$ on X is a preorder, called the* discrete preorder on X*. This is in fact the only relation on X that is both a partial order and an equivalence relation.*

Definition 15 *Let $(X, \leq_X)$ be a proset. We define the induced preorder $\leq$ on $Maybe(X)$ where for all $x, y \in Maybe(X)$ we have*

$$x \leq y \Leftrightarrow x = \bot \vee (x, y \in X \wedge x \leq_X y).$$

Definition 16 *Let X be a set and let $(Y, \leq_Y)$ be a proset. We define the induced preorder $\leq$ on the set Y^X of functions from X to Y where for all $f, g \in Y^X$ we have*

$$f \leq g \Leftrightarrow f(x) \leq_Y g(x) \, \forall x \in X.$$

Definition 17 *Let X be a set and let $(Y, \leq_Y)$ be a proset. We define the induced preorder on $Dict(X, Y) = Maybe(Y)^X$ by combining Definition 15 and Definition 16 above.*

Definition 18 *Let $(Y, \leq_Y)$ be a proset and let $X \subseteq Y$. We define the* induced subset preorder $\leq_X$ *on X where for all $x, y \in X$ we have*

$$x \leq_X y \Leftrightarrow x \leq_Y y.$$

Definition 19 *Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be prosets. We define the* induced product preorder $\leq$ *on $X \times Y$ where for all $x, x' \in X$ and $y, y' \in Y$ we have*

$$(x, x') \leq (y, y') \Leftrightarrow x \leq_X x' \wedge y \leq_Y y'.$$

Joins and meets

Definition 20 *Let $(X, \leq)$ be a proset, let $(x_i)_{i \in I}$ be an indexed family of elements of X and let $x \in X$. We say that x is a* join *of $(x_i)_{i \in I}$ if the following two properties hold:*

- *$x_i \leq x$ for all $i \in I$*
- *if $x' \in X$ is an element such that $x_i \leq x'$ for all $i \in I$, then we have $x \leq x'$.*

Dually, we say that x is a meet *of $(x_i)_{i \in I}$ if the following two properties hold:*

- *$x_i \geq x$ for all $i \in I$*
- *if $x' \in X$ is an element such that $x_i \geq x'$ for all $i \in I$, then we have $x \geq x'$.*

We have that meets and joins are unique up to isomorphism, stated as follows.

Proposition 1 *Let $(X, \leq)$ be a proset, let $(x_i)_{i \in I}$ be an indexed family of elements of X and let $x, y \in X$. If x and y are both joins (or both meets) of $(x_i)_{i \in I}$, then we have $x \simeq y$. If $(X, \leq)$ is also a poset, we have $x = y$.*

Definition 21 *Let $(X, \leq)$ be a proset, let $(x_i)_{i \in I}$ be an indexed family of elements of X and let $x \in X$ be a join (resp. meet) of $(x_i)_{i \in I}$. Then we write $x \simeq \bigvee_{i \in I} x_i$ (resp. $x \simeq \bigwedge_{i \in I} x_i$). If $(X, \leq)$ is also a poset, we have that isomorphisms imply equality, so we can write $x = \bigvee_{i \in I} x_i$ (resp. $x = \bigwedge_{i \in I} x_i$). If $I = \{1, 2\}$ we can instead write $x_1 \vee x_2$ for the join and $x_1 \wedge x_2$ for the meet.*

Examples of joins and meets We now identify the joins and meets in the prosets we constructed in Appendix A.5.

Proposition 2 *Let $(X, \simeq)$ be a setoid, and let $x, y \in X$. Then we have that x and y have a join in X iff $x \simeq y$, in which case we have $x \simeq y \simeq x \vee y$.*

Proposition 3 *Let X be a proset and consider the induced proset $Maybe(X)$. For all $x \in Maybe(x)$ we have $x \vee \bot \simeq x$. Also, for all $x, y \in X$, we have that x and y have a join in $Maybe(X)$ iff they have a join in X, in which case the two joins are isomorphic.*

Proposition 4 *Let $(X, \simeq)$ be a setoid, and let $x, y \in Maybe(X)$. Then, x and y have a join in $Maybe(X)$ iff*

$$x \neq \bot \wedge y \neq \bot \Rightarrow x \simeq y.$$

If this is the case, we have $x \vee y \simeq \mathsf{First}(x, y)$.

Proposition 5 *Let X be a set, let $(Y, \leq_Y)$ be a proset and let $f, g \in Y^X$. We have that f and g have a join in Y^X iff $f(x)$ and $g(x)$ have a join $f(x) \vee g(x)$ in Y for all $x \in X$. In this case, we have $f \vee g \simeq h$, where h is a map where $h(x) \simeq f(x) \vee g(x)$ for all $x \in X$.*

Proposition 6 *Let X be a set, let $(Y, \simeq)$ be a setoid and let $D_1, D_2 \in Dict(X, Y)$ be two dictionaries. Then, we have that D_1 and D_2 have a join in $Dict(X, Y)$ iff for all $x \in X$ we have*

$$D_1(x) \neq \bot \wedge D_2(x) \neq \bot \Rightarrow D_1(x) \simeq D_2(x).$$

If this is the case, then we have $D_1 \vee D_2 \simeq \mathsf{Merge}(D_1, D_2)$.

Proof. This follows from Proposition 4 and Proposition 5.

Monotone functions

Definition 22 *Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be prosets, and let $f : X \to Y$ be a function. We say that f is* monotone *(also often called order-preserving) if for all $a, b \in X$ we have*

$$a \leq_X b \Rightarrow f(a) \leq_Y f(b).$$

Proposition 7 *If $(X, \leq_X)$, $(Y, \leq_Y)$ and $(Z, \leq_Z)$ are prosets, and if $f : X \to Y$ and $g : Y \to Z$ are monotone functions, then the composite function*

$$\begin{aligned} g \circ f \colon X &\to Z \\ x &\mapsto g(f(x)) \end{aligned}$$

is also monotone.

Lattice-ordered abelian groups We now define lattice-ordered abelian groups, which is the structure we require from the set $\mathcal{V}$ of transaction values (and account balances) in our design.

Definition 23 *A* lattice *is a poset in which every finite indexed family of elements has both a join and a meet.*

Definition 24 *A* lattice-ordered abelian group *is a tuple $(X, \leq, +, 0)$ where X is a set, $\leq$ is a binary relation on X, $+$ is a binary operator on X and $0 \in X$, such that*

- *$(X, +, 0)$ is an abelian group*
- *$(X, \leq)$ is a lattice*
- *For all $a, b, x \in X$ we have*

$$a \leq b \Rightarrow a + x \leq b + x.$$

Definition 25 *Given a lattice-ordered abelian group $(X, \leq, +, 0)$, we say that an element $x \in X$ is* positive *if $0 \leq x$.*

B Computing balances

In this section we define the function

$$\mathsf{Bal}\colon \Pi \times \mathcal{B}^* \to \mathcal{V}^{\overline{\mathcal{K}}}$$

which is used in the simplified design to compute account balances from a balance proof and rollup contract state. This function is used by users to compute their own balance, as well as by the rollup contract when processing a withdrawal request. Given a balance proof $\pi \in \Pi$ and the current rollup state $B_* \in \mathcal{B}_*$, the account balances are computed in two steps. First, we extract a list of partial transactions from π and B_*, where a partial transaction consists of a sender, a recipient and a (possibly unknown) transaction amount. Then, we compute the balances of every account by applying a state transition function on the list of partial transactions. We now describe the steps in more details.

B.1 Step 1: Extracting a list of partial transactions

The first step of calculating balances is to extract a list of *partial transactions* from a balance proof π and the current list of blocks in the rollup B_*. The set of partial transactions is defined as

$$\mathcal{T} := Maybe(\mathcal{V})^{\overline{\mathcal{K}}}.$$

The process of extracting the list of partial transactions is described by a function

$$\mathsf{TransactionsInBlocks}\colon \Pi \times B^* \to \mathcal{T}^*$$

which we will now define.

$$\begin{aligned} \mathsf{TransactionsInBlock}_{explicit}\colon \mathcal{B}_{explicit} &\to \mathcal{T}^* \\ B &\mapsto B \end{aligned}$$

For each $s \in \overline{\mathcal{K}}$ we define $\bot(s) \in \mathcal{T}$ as

$$\bot(s)_k := \begin{cases} \bot, & \text{if } k = s \\ 0, & \text{otherwise.} \end{cases}$$

We then define the function

$$\mathsf{TransactionsInBlock}_{transfer}\colon \Pi \times \mathcal{B}_{transfer} \to \mathcal{T}^*$$

for extracting a list of partial transactions from a balance proof and a transfer block as follows. Given a balance proof $\pi \in \Pi$ and a transfer block $(aggregator, extradata, C, S, \sigma) \in$

$B_{transfer}$, we take, for each sender $s \in \mathcal{K}_2$ in order, the partial transaction $v \in \mathcal{T}$ where

$$v = \begin{cases} t, \text{ where } (\_, t) = \pi(C, s), & \text{if } s \in S \text{ and } \pi(C, s) \neq \bot \\ \bot(s), & \text{if } s \in S \text{ and } \pi(C, s) = \bot \\ 0, & \text{if } s \notin S. \end{cases}$$

We combine these functions into one function for extracting partial transactions from a balance proof and a block :

$$\mathsf{TransactionsInBlock}\colon \Pi \times \mathcal{B} \to \mathcal{T}^*$$
$$(\pi, B) \mapsto \begin{cases} \mathsf{TransactionsInBlock}_{explicit}(B), & \text{if } B \in \mathcal{B}_{explicit} \\ \mathsf{TransactionsInBlock}_{transfer}(\pi, B), & \text{if } B \in \mathcal{B}_{transfer} \end{cases}.$$

Finally, to extract a list of partial transactions from a balance proof and a list of blocks, we extract the transactions from each block and concatenate the lists of partial transactions :

$$\mathsf{TransactionsInBlocks}\colon \Pi \times B^* \to \mathcal{T}^*$$
$$(\pi, (B_i)_{i\in[n]}) \mapsto \mathsf{Concatenate}((\mathsf{TransactionsInBlock}(\pi, B_i))_{i\in[n]}).$$

B.2 Step 2: Computing balances from a list of partial transactions

The second step in computing balances is to apply a *transition function* to the list of partial transactions obtained in step 1, starting from the state where all account balances are zero.

Definition 26 *A* transition function[12] *is a function on the form $f : \mathcal{T} \times \mathcal{S} \to \mathcal{S}$, where $\mathcal{T}$ is called the set of transactions and $\mathcal{S}$ is called the set of states.*

In our case, a state is an assignment of a balance to each account, where every non-source account has a positive balance :

$$\mathcal{S} := \{b \in \mathcal{V}^{\overline{\mathcal{K}}}, \text{ such that } b_k \geq 0, \forall k \in \overline{K} \backslash \{\text{Source}\}\},$$

and the set of transactions is the set $\mathcal{T}$ of partial transactions defined in Step 1 above. In order to define the transition function f, we will first define a different transition function $f_c : \mathcal{T}_c \times \mathcal{S} \to \mathcal{S}$, where the set of transactions is the subset $\mathcal{T}_c \subseteq \mathcal{T}$, called the *complete transactions*, consisting of the transactions $v \in \mathcal{T}$ where $v_k \neq \bot \, \forall k \in \overline{\mathcal{K}}$, and where

$$\sum_{k \in \overline{\mathcal{K}}} v_k \leq 0.$$

[12] Sometimes called a semiautomation in the literature.

For all complete transactions $v \in \mathcal{T}_c$ and for all $b \in \mathcal{S}$ we define

$$f_c(v, b) := \begin{cases} b + v, & \text{if } b + v \in \mathcal{S} \\ b, & \text{otherwise.} \end{cases}$$

Before we can define the transition function $f : \mathcal{T} \times \mathcal{S} \to \mathcal{S}$, we will define a partial order on $\mathcal{T}$ and $\mathcal{S}$. In the following definitions, we apply the inductions from Appendix A.5. First, the partial order on $\mathcal{V}$ is the partial order coming from the fact that $\mathcal{V}$ is a lattice. This induces a partial order on $Maybe(\mathcal{V})$, where $\perp$ is the bottom element. We then get an induced partial order on $T = Maybe(\mathcal{V})$, of which $\mathcal{S}$ is a subset. We then give $\mathcal{S}$ the induced subset partial order.

Definition 27 *A* transition function for partial transactions *is a monotone function*

$$f : \mathcal{T} \times \mathcal{S} \to \mathcal{S}$$

such that

$$f(t, b) \leq f_c(t', b')$$

for all $(t', b') \in \mathcal{T}_c \times \mathcal{S}$ *where* $(t, b) \leq (t', b')$.

In the remainder of the protocol descripiton, let

$$f : \mathcal{T} \times \mathcal{S} \to \mathcal{S}$$

be a transition function for partial transactions (called just the *transition function*). The transition function $f : \mathcal{T} \times \mathcal{S} \to \mathcal{S}$ induces the function $f^* : \mathcal{T}^* \times \mathcal{S} \to \mathcal{S}$ which takes a list of transactions $T_* \in \mathcal{T}^*$ and an initial state $s_0 \in \mathcal{S}$ and returns the state obtained by applying the transition function f, in order, to every transaction in T_*, starting with the initial state s_0 . In our case, given the list of partial transactions T_* obtained in Step 1, we compute the balances as $f^*(T_*, 0)$, where $0 \in \mathcal{S}$ is the initial state where every account has a zero balance . Combining the two steps, we define the balance function as follows :

$$\begin{aligned} &\mathsf{Bal} \colon \Pi \times B^* \to \mathcal{S} \\ &\qquad (\pi, B_*) \mapsto f^*(\mathsf{TransactionsInBlocks}(\pi, B_*), 0). \end{aligned}$$

C Security

In this section we define and prove the security of the rollup contract. Informally speaking, we say that the rollup contract is secure if every withdrawal request succeeds, i.e. the rollup contract has sufficient balance for every withdrawal. This means that if a user has a balance proof which proves the in-rollup balance of one or more of their L1 accounts, they will be able to withdraw these balances to L1.

C.1 Formal description of the rollup contract

We formally define the rollup contract.

Definition 28 *The rollup contract state consists of the list of blocks that have been added to the rollup. Formally, we define the set of contract states as*

$$\mathcal{S}_{contract} := \mathcal{B}^*.$$

Definition 29 *We define the set of rollup requests as*

$$\mathcal{R} := \mathcal{R}_{deposit} \amalg \mathcal{R}_{transfer} \amalg \mathcal{R}_{withdrawal},$$

where

$$\begin{aligned} \mathcal{R}_{deposit} &:= \mathcal{B}_{deposit} \\ \mathcal{R}_{transfer} &:= \mathcal{B}_{transfer} \\ \mathcal{R}_{withrawal} &:= \mathit{\Pi}. \end{aligned}$$

Definition 30

$$\begin{aligned} \mathsf{ToBlock}_{deposit} \colon \mathcal{R}_{deposit} &\to \mathcal{B}_{deposit} \\ B &\mapsto B. \end{aligned}$$

Definition 31

$$\begin{aligned} \mathsf{ToBlock}_{transfer} \colon \mathcal{R}_{transfer} &\to \mathcal{B}_{transfer} \\ B &\mapsto B \end{aligned}$$

Definition 32

$$\begin{aligned} \mathsf{ToBlock}_{withdrawal} \colon \mathcal{R}_{withdrawal} \times \mathcal{B}^* &\to \mathcal{B}_{withdrawal} \\ (\pi, B_*) &\mapsto \mathsf{Bal}(\pi, B_*)|_{K_1} \end{aligned}$$

Definition 33

$$\mathsf{ToBlock} \colon \mathcal{R} \times \mathcal{B}^* \to \mathcal{B}$$

$$(R, B_*) \mapsto \begin{cases} \mathsf{ToBlock}_{deposit}(R), & \text{if } R \in \mathcal{R}_{deposit} \\ \mathsf{ToBlock}_{transfer}(R), & \text{if } R \in \mathcal{R}_{transfer} \\ \mathsf{ToBlock}_{withdrawal}(R, B_*), & \text{if } R \in \mathcal{R}_{withdrawal} \end{cases}$$

Definition 34 *A rollup contract transaction consists of a rollup request and a transaction sender. Formally, we define the set of rollup contract transactions as*

$$\mathcal{T}_{contract} := \mathcal{R} \times \mathcal{K}_1.$$

Definition 35

$$\mathsf{IsValid}_{\mathcal{T}_{contract}} : \mathcal{T}_{contract} \to \{True, False\}$$

$$(R, s) \mapsto \begin{cases} True, & \text{if } R \in \mathcal{R}_{deposit} \\ \mathsf{SA.Verify}(S, (C, agg, e), \sigma) & \\ \wedge\, s = agg, & \text{if } R = (agg, e, C, S, \sigma) \in \mathcal{R}_{transfer} \\ \mathsf{Verify}(\pi), & \text{if } R = \pi \in \mathcal{R}_{withdrawal} \end{cases}$$

Definition 36

$$f_{contract} : \mathcal{T}_{contract} \times \mathcal{S}_{contract} \to \mathcal{S}_{contract}$$

$$((R, s), B_*) \mapsto \begin{cases} (B_* || \mathsf{ToBlock}(R, B_*)), & \text{if } \mathsf{IsValid}_{\mathcal{T}_{contract}}(R, s) \\ B_*, & \text{otherwise.} \end{cases}$$

The balance of the rollup contract is not part of the contract state, but is computed from the blocks in the rollup as follows. The contract balance is defined to be initially zero, and then subsequently updated each time a new rollup block is added, using the following update function.

Definition 37

$$updateBalance : \mathcal{B} \times \mathcal{V} \to \mathcal{V}$$

$$(B, v) \mapsto \begin{cases} v + d, & \text{if } B = (recipient, d) \in \mathcal{B}_{deposit} \\ v, & \text{if } B \in \mathcal{B}_{transfer} \\ v - \sum_{k \in \mathcal{K}_1} w_k, & \text{if } B = w \in \mathcal{B}_{withdrawal} \end{cases}$$

C.2 Security definition

We formally define the security of the rollup contract with the following attack game.

Attack game 1 *The attack game is played between a PPT adversary and a challenger, where the challenger plays the role of the rollup contract. First, the challenger initializes the rollup contract with the state* $((), 0) \in \mathcal{S}_{contract}$*. Then, the adversary sends a sequence of contract transactions (elements of* $\mathcal{T}_{contract}$*) to the challenger. For each contract transaction, the challenger updates the rollup contract state using the transition function* $f_{contract}$*. The adversary wins the attack game if at the end of the interaction, the rollup contract has a state* $(B_*, balance)$ *where*

$$balance \ngeq 0.$$

Definition 38 *The rollup contract is* secure *if winning Attack game 1 is at least as hard as breaking either the binding property of the authenticated dictionary scheme, or finding a collision of the hash function* H.

C.3 Security proof

Before we can prove the security of the rollup contract, we will first prove some properties of the balance function.

Lemma 1. *For all balance proofs $\pi \in \Pi$ and block lists $B_* \in \mathcal{B}^*$ we have*

$$\mathsf{Bal}(\pi, B_*)_{Source} \leq 0.$$

Proof. We start by noticing that the transition function for complete transactions f_c decreases the sum of account balances, i.e.

$$\sum_{k \in \overline{\mathcal{K}}} f_c(T, b)_k \leq \sum_{k \in \overline{\mathcal{K}}} b_k, \quad \forall T \in \mathcal{T}_c,\, b \in \mathcal{S}.$$

This implies the following fact about the transition function for partial transactions f :

$$\sum_{k \in \overline{\mathcal{K}}} f(T, b)_k \leq \sum_{k \in \overline{\mathcal{K}}} b_k, \quad \forall T \in \mathcal{T},\, b \in \mathcal{S}.$$

Then, it follows by induction that we have

$$\sum_{k \in \overline{\mathcal{K}}} f_*(T_*, 0)_k \leq 0, \quad \forall T_* \in \mathcal{T}^*. \tag{1}$$

Finally, for all balance proofs $\pi \in \Pi$ and block lists $B_* \in \mathcal{B}^*$, letting $T_* = \mathsf{TransactionsInBlocks}(B_*)$ we have

$$\begin{aligned}
\mathsf{Bal}(\pi, B_*)_{\text{Source}} &= f^*(T_*, 0)_{\text{Source}} && \text{(by definition)} \\
&= \sum_{k \in \overline{\mathcal{K}}} f^*(T_*, 0)_k - \sum_{k \in \overline{\mathcal{K}} \setminus \{\text{Source}\}} f^*(T_*, 0)_k \\
&\leq - \sum_{k \in \overline{\mathcal{K}} \setminus \{\text{Source}\}} f^*(T_*, 0)_k && \text{(by Equation (1))} \\
&\leq 0 && \text{(non-source accounts have positive balances)}
\end{aligned}$$

which is the statement of the lemma.

The next lemma relies on a preorder structure on the set of balance proofs.

Definition 39 *We give*

$$\Pi = Dict(\mathsf{AD}.\mathcal{C} \times \mathcal{K}_2, (\mathsf{AD}.\Pi \times \{0,1\}^*) \times \mathcal{T})$$

a preorder as follows. First, we give $\mathsf{AD}.\Pi \times \{0,1\}^*$ the trivial preorder. Then, we give $(\mathsf{AD}.\Pi \times \{0,1\}^*) \times \mathcal{V}_+^{\mathcal{K}}$ the induced product preorder, and Π the induced dictionary preorder.

Proposition 8 *The function* TransactionsInBlocks *is monotone.*

Proposition 9 *The function f^* is monotone.*

Proof. This follows by induction.

Proposition 10 *The balance function*

$$\mathsf{Bal} : \Pi \times B^* \to \mathcal{S}$$

is monotone.

Proof. This follows from Proposition 8 and Proposition 9.

Lemma 2. *Let $(B_i)_{i\in[n]} \in \mathcal{B}^*$ be a list of blocks and let $\pi \in \Pi$ be a balance proof. Then we have*

$$\mathsf{Bal}(\pi, B_*)_{Source} = \sum_{\substack{i\in[n] \\ B_i\in\mathcal{B}_{withdrawal} \\ B_i=w \\ k\in\mathcal{K}_1}} \big(w_k \wedge \mathsf{Bal}(\pi, (B_j)_{j\in[i-1]})_k\big) - \sum_{\substack{i\in[n] \\ B_i\in\mathcal{B}_{deposit} \\ B_i=(r,v)}} v$$

Theorem 1 *The rollup contract is secure (by Definition 38).*

Proof. Suppose an adversary and a challenger have interacted in Attack game 1. We will show that either the resulting contract balance is positive (the adversary lost the game), or the adversary has been able to either break the binding property of the authenticated dictionary scheme or found a collision of the hash function H. Let $B_* = (B_i)_{i\in[n]}$ be the contract state after the attack game , let $I \subseteq [n]$ be the indices of the withdrawal blocks in B_* and let $(\pi_i)_{i\in I}$ be the balance proofs used in the withdrawal requests . The resulting contract balance can be computed by adding all deposited amounts and subtracting all withdrawn amounts :

$$contractBalance = v_{deposited} - v_{withdrawn},$$

where

$$v_{deposited} = \sum_{\substack{i\in[n] \\ B_i\in\mathcal{B}_{deposit} \\ B_i=(r,v)}} v$$

and

$$v_{withdrawn} = \sum_{\substack{i\in I \\ k\in\mathcal{K}_1}} \mathsf{Bal}(\pi_i, (B_j)_{j\in[i-1]})_k.$$

We now have two possibilities, either the balance proofs $(\pi_i)_{i\in I}$ have a join in Π or they don't . Suppose they have a join $\pi \in \Pi$. Then we have

$$
\begin{aligned}
0 &\leq -\mathsf{Bal}(\pi, B_*)_{\text{Source}} && \text{(lemma 1)}\\
&= v_{deposited} - \sum_{\substack{i\in I\\ k\in\mathcal{K}_1}} \mathsf{Bal}(\pi_i, (B_j)_{j\in[i-1]})_k \wedge \mathsf{Bal}(\pi, (B_j)_{j\in[i-1]})_k && \text{(lemma 2)}\\
&= v_{deposited} - \sum_{\substack{i\in I\\ k\in\mathcal{K}_1}} \mathsf{Bal}(\pi_i, (B_j)_{j\in[i-1]})_k && \text{(Follows from Proposition 10 since } \pi_i \leq \pi \text{)}\\
&= contractBalance
\end{aligned}
$$

which shows that the contract balance is positive . Now, suppose the balance proofs $(\pi_i)_{i\in I}$ do not have a join in Π. Let i_k be the $k'th$ index in I (so that $I = \{i_1, i_2, \ldots, i_m\}$, where $m = |I|$). Then, let $(\pi'_k)_{k\in[m]}$ be the balance proofs defined recursively as

$$
\pi_k = \begin{cases} \bot, & \text{if } k = 0 \\ \mathsf{Merge}(\pi'_{k-1}, \pi_{i_k}), & \text{if } k \geq 1. \end{cases}
$$

 . Clearly, these merged balance proofs are valid, since each of the original balance proofs are valid (otherwise they wouldn't be accepted by the rollup contract), and since the merge of two valid balance proofs is again valid. Now, we argue that there must be an index $k \in \{1, \ldots, m\}$ such that π'_k is *not* the join of π'_{k-1} and π_{i_k} in Π, since if not, the final merged balance proof π'_m would be a join of $(\pi_i)_{i\in I}$ (by Proposition 6), which we have assumed not to exist .

It then follows from Proposition 6 that there is a key $(C, s) \in \mathsf{AD}.\mathcal{C} \times \mathcal{K}_2$ such that $\pi'_{k-1}(C, s) \not\preceq \pi_{i_k}(C, s)$. Letting $((\pi, salt), t) = \pi'_{k-1}(C, s)$ and $((\pi', salt'), t') = \pi_{i_k}(C, s)$, this implies $t \neq t'$. Also, since both balance proofs are valid, as remarked earlier, we have

$$\mathsf{AD}.\mathsf{Verify}(\pi, s, \mathsf{H}(t, salt), C)$$

and

$$\mathsf{AD}.\mathsf{Verify}(\pi', s, \mathsf{H}(t', salt'), C).$$

It follows that that either $\mathsf{H}(t, salt) = \mathsf{H}(t', salt')$, meaning that we have found a hash collision , or $\mathsf{H}(t, salt) \neq \mathsf{H}(t', salt')$, which means we have broken the binding property of the authenticated dictionary scheme .

D Discussion

D.1 Tracing the Path to Intmax2

Plasma Prime [18] 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 [9] 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 [14] 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.

D.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., [17]).

D.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.

D.4 Delegating Zero-Knowledge Proof Generation

The emergence of new research on delegating the generation of zero-knowledge proofs [11], 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.