PUBLIC

Code Assessment

of the Intmax 2 ZKP Circuits

May 05, 2025

Produced for

INTMAX

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Intmax with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed circuits of Intmax 2 ZKP according to Scope to support you in forming an opinion on their security risks.

Intmax implements a zk-rollup for private transfers using zk proofs to prove state transitions and balances. Smart contracts on Scroll manage the rollup, while smart contracts on Ethereum handle liquidity on- and offboarding.

The most critical subjects covered in our audit of the circuits are soundness, completeness and zero-knowledge. Several critical issues have been uncovered including:

- Missing constraint on pubkey allows for double spending
- Incorrect exclusion proof circuit allows for double spending
- Multiple Valid Account Trees / Public States after applying the same block

The latest iteration covers refinements in the core circuits, refactored withdrawal circuits and the new claim circuits.

The reviewed code is well-structured and properly documented. Although testing has been continuously enhanced, a thorough stress test on a public testnet is recommended before mainnet launch due to the project's cutting-edge nature. The circuits are part of Intmax 2, a system which consists of multiple interacting parts. The rollup is managed by a set of smart contracts which have been audited in the ChainSecurity Intmax 2 Smart Contract Audit Report.

In summary, we find that the current codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	7
• Code Corrected	7
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	3
• Code Corrected	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Intmax 2 ZKP repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	27 Aug 2024	4f43690a9cd005686f1283746204845f13dcea8b	Initial Version
2	16 Dec 2024	43ebf80e36abf55a1289626a05f732fe96f001ae	After Intermediate Report
3	20 Dec 2024	3befdfa05002e99722010c5a25ad70216029598e	Fix for 5.2
4	20 Dec 2024	aca65d0c44b3f736224c8364f265ba41d431ab64	Fix for 5.3
5	23 Dec 2024	9037139f2f61c523e942340541d91e0eabe9183d	Final fixes for first version
6	14 Apr 2025	8e413bd556e44f9a4b353f8d489a3833e584b88d	Claim feature added
7	05 May 2025	d55e9cc4db422d197abf167d0726791499966855	Final commit

A forked version of Plonky2 at commit `4813d563` is used for the zero knowledge proofs.

The scope of this review is the completeness and soundness of the ZKP circuits in the following files:

```
circuits/
    balance/
        balance_circuit.rs
        balance_pis.rs

        receive/
            receive_deposit_circuit.rs
            receive_transfer_circuit.rs
            update_circuit.rs

            receive_targets/
                private_state_transition.rs
                transfer_inclusion.rs

        send/
            sender_circuit.rs
            spent_circuit.rs
            tx_inclusion_circuit.rs
        transition/
            transition_circuit.rs

    fraud/
        fraud_circuit.rs

    validity/
        validity_circuit.rs
        validity_pis.rs

        block_validation/
            account_exclusion.rs
            account_inclusion.rs
            aggregation.rs
            format_validation.rs
            main_validation.rs
            utils.rs

        transition/
            account_registration.rs
            account_transition_pis.rs
            account_update.rs
            transition.rs
            wrapper.rs

    withdrawal/
        withdrawal_circuit.rs
        withdrawal_inner_circuit.rs
        withdrawal_wrapper_circuit.rs

common/
    block.rs
    deposit.rs
    generic_address.rs
    insufficient_flags.rs
    private_state.rs
    public_state.rs
    salt.rs
    transfer.rs
    tx.rs
    withdrawal.rs

    signature/
        flatten.rs
        format_validation.rs
        key_set.rs
        mod.rs
        sign.rs
        utils.rs
        verify.rs

    trees/
        account_tree.rs
        asset_tree.rs
        block_hash_tree.rs
        deposit_tree.rs
        nullifier_tree.rs
        sender_tree.rs
        transfer_tree.rs
        tx_tree.rs

ethereum_types/
    account_id_packed.rs
    address.rs
    bytes16.rs
    bytes32.rs
    u256.rs
    u32limb_trait.rs

utils/
    cyclic.rs
    dummy.rs
    leafable_hasher.rs
    leafable.rs
    logic.rs
    poseidon_hash_out.rs
    recursively_verifiable.rs
    wrapper.rs

    trees/
        get_root.rs
        incremental_merkle_tree.rs
        merkle_tree.rs
        sparse_merkle_tree.rs

        indexed_merkle_tree/
            insertion.rs
            leaf.rs
            membership.rs
            mod.rs
            update.rs
```

The libraries `plonky2_keccak` and `plonky2_bn254` are part of a separate review: https://www.chainsecurity.com/security-audit/plonky2-bn254-keccak256.

After Version 6 the following files have been added/moved/removed from the scope:

Added:

```
circuits/
    claim/
        deposit_time.rs
        determine_lock_time.rs
        single_claim_circuit.rs
        utils.rs
    withdrawal/
        single_withdrawal_circuit.rs
common/
    claim.rs
ethereum_types/
    u64.rs
```

```
utils/hash_chain/
    chain_end_circuit.rs
    hash_inner_circuit.rs
    mod.rs
```

Moved:

```
common/signature/verify.rs -> common/signature_content/aggregation_validation.rs
common/signature/flatten.rs -> common/signature_content/flatten.rs
common/signature/format_validation.rs -> common/signature_content/format_validation.rs
common/signature/key_set.rs -> common/signature_content/key_set.rs
common/signature/mod.rs -> common/signature_content/mod.rs
common/signature/sign.rs -> common/signature_content/block_sign_payload.rs
common/signature/utils.rs -> common/signature_content/utils.rs

circuits/withdrawal/withdrawal_circuit.rs -> utils/hash_chain/cyclic_hash_chain_circuit.rs

ethereum_types/account_id_packed.rs -> ethereum_types/account_id.rs
```

Removed:

```
circuits/withdrawal/
    withdrawal_inner_circuit.rs
    withdrawal_wrapper_circuit.rs
```

2.1.1 *Excluded from scope*

- Plonky2 soundness and correctness properties
- Plonky2 implementation
- Plonky2 functions and libraries, including `plonky_u32`
- Witness generation / trace computation / proof generation
- Impact of proof generation time on the system
- Tests
- All the files not explicitly listed above, especially, the circuits in `proof_of_innocence/`
- Compilation of the circuits with features other than the empty `default = []` is out of the scope of this review

2.2 System Overview

This system overview describes the latest version (**Version 6**) as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Intmax 2 is a zk-rollup for private transfers. Liquidity is on/offboarded on Ethereum while the state of the rollup is managed on Scroll. The system is based on four zk proofs to prove state transition and balance.

The public state of the blockchain consists of:

- block tree: tree built from the blocks

- account tree: holds the BLS public keys of each account and records the last block each account transacted
- previous account tree: same as account tree but for the previous block
- next account id: incremental counter to assign IDs to new accounts
- deposit tree: incremental merkle tree that records all deposits
- block hash: hash of the last block
- timestamp: timestamp of the last block
- block number: number of the last block

Each account keeps track of a private state which consists of:

- asset tree: a merkle tree that tracks the account's balances for all tokens. Each leaf corresponds to a token index, with the stored data representing the token balance. An `is_insufficient` flag indicates whether a balance shortage has occurred (when a transfer attempts to spend more than the available balance), preventing further payments with that token.
- nullifier tree: records the nullifiers used when receiving a deposit or a transfer.
- nonce: transactions are ordered and must be executed sequentially, enforced by an incremental nonce.
- salt: a random number is used to obfuscate the content of the private state when revealing the private state commitment.

Based on its private state and the public state, each account generates balance proofs. These proofs ensure that the account's token balances are valid and in line with the shared public state.

- Blocks contain minimal information: the previous block hash, block number, block timestamp, deposit root, and signature hash. The signature hash encapsulates the signature content, which includes the transactions.
- The block hash tree is a merkle tree that records the hashes of all previous blocks.
- The account tree is an indexed merkle tree that tracks all accounts that have sent a transaction, i.e. its leaves store an account's BLS public key along with the last block (`last_block_number`) in which the account transacted. The index of a leaf is the account ID of the associated public key. The `last_block_number` of an account is updated every time the account submits and signs a transaction, this serves as a reference to ensure double-spending is prevented.
- The deposit tree is an incremental merkle tree that records all deposits and is updated by the smart contract. Each block includes the current root of the deposit tree.

Transactions work as follows: Users create a transfer tree consisting of up to 64 individual transfer elements, with each element containing the fields `recipient_public_key`, `token_index`, `amount`, and a `salt`. A transaction is then generated using the `transfer_tree_root` and the users's current `nonce`, which is submitted to a block builder. The block builder aggregates up to 128 transactions from multiple senders into a transaction tree (tx tree). A sender can sign at most one transaction per block, otherwise the block is invalid.

The block builder sends the block payload back to each sender for their signature and collects the signatures. The block payload to sign consists of a boolean indicating whether the block is a registrationblock, the `tx_tree_root` containing all the transactions of the block, an expiry, the block builder address, and the block builder nonce. Once the signatures are collected, the block builder creates a sender tree. The leafs contain the senders pubkey and a flag `is_valid` indicating whether the sender has signed the block payload. The index of the leave corresponds to the index of the sender's tx in the tx_tree.

Finally, the block builder aggregates the signature content which consists of:

- is_registration_block: Whether the block is for account updates (transfers sent by accounts already existing in the account tree) or account registrations (transfers sent by accounts not yet existing in the account tree).
- tx_tree_root: Root of the transaction tree.
- sender_flag: Indicates which senders signed the transaction tree.
- pubkey_hash or account_id_hash: Hashes corresponding to the sender's public keys or account IDs.
- agg_pubkey: The aggregated pubkey of all senders who signed. The aggregated pubkey is computed following https://eprint.iacr.org/2018/483
- agg_signature: The aggregated signature from all valid senders.
- message_point: The point corresponding to the hash of the block payload mapped to a point in the G2 group of the BN254 curve.

When a block is submitted to the rollup, the contract receives the signature content and adds the previous block hash, deposit tree root, block timestamp and block number. The block hash is then computed by hashing the previous block hash, block number, signature hash, and deposit root.

Anyone can act as a block builder. While the system can handle invalid or empty blocks, they must still be proved in sequence, which requires significant time and can impair system performance. The content of invalid blocks is simply ignored, as no proof with the `is_valid` flag set to true can be generated for them.

For withdrawals, users simply perform a regular transfer, but specify an Ethereum address (20 bytes) instead of a public key.

After the block containing the transaction has been included, the sender can update their private state and balance proof. The recipient then requires the sender's updated balance proof to update their own private state and balance proof or create a withdrawal proof.

State transitions in Intmax 2 are based on four zk proofs.

2.2.1 Validity Proof

The validity proof proves the correctness of an Intmax 2 block, both its format and the resulting transition of the public state. It is a cyclic proof, meaning each proof is based on the previous one, with the first proof generated from the genesis state. The validity proof can be built by anyone observing the posted blocks, along with calldata on the rollup smart contract (see ChainSecurity's Intmax 2 Smart Contract Audit Report), however, users are expected to query it from dedicated nodes. The validity proof is pivotal in the Intmax 2 protocol, it serves as a base to build the validity proof of the next block, and users need it to update their balances.

The public inputs for the validity proof are:

- PublicState
- tx_tree_root
- sender_tree_root
- is_valid_block

The validity circuit takes a transition proof and the previous validity proof as private inputs, and generates a new validity proof.

The transition proof is generated by the transition wrapper circuit, which validates:

1. A main validation proof (validates the block format).
2. A `ValidityTransitionTarget` that ensures the public state transitions correctly.

The main validation proof ensures the block format is correct. It verifies:

1. Conditionally, an account exclusion (for registration blocks) or account inclusion proof (for non-registration blocks). For non-registration blocks, all accounts must exist in the account tree. A valid merkle proof must be provided as witness for each account or no proof can be generated. For registration blocks, none of the accounts must already exist in the account tree. For each account, a membership proof with `is_included = false` must be provided. This includes a merkle proof of the preceding leaf, linked to the next existing leaf, demonstrating that the account is not part of the indexed merkle tree. The main validation proof connects the data of the individual proofs to ensure they operate on the same data.
2. A format validation proof, and if the result is considered valid up to this point, an aggregation proof. Format validation ensures that the signatures associated with the public keys are of valid format. Aggregation constrains the correctness of the aggregated public key based on the individual public keys.

The `ValidityTransitionTarget` ensures that the public state (account tree, block tree) evolves correctly. It verifies an account registration or update proof and a block hash Merkle proofs lead to the correct resulting state.

For account registration, the proof iterates through all new accounts, verifying that the resulting account tree root is correct. Prior to version `4`, every public key of the `tx_tree` was registered in the account tree, the pubkeys related to a sent-but-unsigned transaction were registered with their `block_number` equal to `0`. In version `4` this was changed, public keys of accounts that haven't signed the `tx_tree_root` are skipped and no longer added to the account tree. For account updates (transfers), it ensures the account tree root reflects the updated `last_block_number` for all signers in the block.

2.2.2 Balance Proof:

The balance circuit is a cyclic proof, a new balance proof is generated based on the previous balance proof and a BalanceTransitionProof. This proof refers to a public state and validates events such as receiving or sending transfers, processing deposits, or confirming that no transaction has been sent from the account (update proof). The balance proof ensures that the account's private state has been updated correctly in regard to the associated public state. The balance proof of the sender is also used by the recipient to update their own balance in order to receive transfers.

The public inputs for the balance proof are:

- pubkey
- private_commitment
- last_tx_hash
- last_tx_insufficient_flags
- public state

The BalanceTransitionCircuit conditionally verifies different proofs based on the type of action being performed:

- receive_transfer_circuit
- receive_deposit_circuit
- update_circuit
- sender_circuit

1. Receive Transfer Circuit:

The receive transfer circuit constrains the private state update when receiving a transaction. The recipient must first update his balance proof to a public state including the transaction to be received. It validates that the block containing the transfer is valid by checking its inclusion against the public state, using a merkle proof. The balance proof of the sender, where the sender circuit is proven for this transaction, must be available to the recipient and is verified. It is ensured that the balance proof corresponds to this transaction which includes the transfer and that the sender had sufficient funds for the transfer that is received. Further it ensures that the recipient's nullifier tree was updated properly and includes the nullifier that corresponds to the transfer commitment. This update guarantees that the same transaction cannot be used to update the balance again. Finally, the circuit verifies that the private state transition, which updates the asset tree, correctly reflects the `token_index` and amount associated with the received transfer.

2. Receive Deposit Circuit:

The receive deposit circuit constrains the processing of deposits. It enforces that the deposit being processed is part of the deposit tree by verifying its inclusion in the merkle tree. The recipient must first update his balance proof to a public state including the deposit to be received. The circuit checks that the pubkey salt hash of the deposit matches the hash calculated from the user's public key and salt, which proves the user's ownership of the deposit. The deposit hash is treated as a nullifier and added to the account's nullifier tree to prevent reuse. The circuit ensures that the asset tree is updated for the correct token index and deposit amount, ensuring the private state transition was done correctly.

Receive Transfer and Receive Deposit proofs can be created at any time after the transaction or deposit has been included in a block. These proofs update the account's asset tree, adding the respective funds to the account's balance, which then allows the balance to be accessed and used.

3. Update Circuit:

The update circuit allows an account to update its balance proof to a new public state. It verifies that no transaction has been sent by this account since the last balance proof by checking that the `last_block_number` in the account tree has not increased since the previous public state. This ensures that no outgoing transaction has occurred from the account during this period. Note that incoming transactions are not observed and do not matter.

4. Sender Circuit:

The sender circuit is used when a user sends a transaction. After the transaction is included in a block, the sender updates their balance proof and shares it with the recipient, enabling the recipient to access the funds. The sender circuit verifies two proofs: the spent proof, which proves that the sender's asset tree has been correctly updated by reducing the balance of the `token_index` corresponding to the transfers of the transaction and the tx inclusion proof, which proves that the transaction was indeed included in a valid block. These two proofs ensure that the transaction is legitimate and that the balance reduction has been correctly executed.

Important: Once an account attempts to transfer more than its balance for a specific token (identified by the `token_index`), the insufficient flags are set. This flag prevents the account from using the balance of that token again, even after additional funds have been added.

2.3 Withdrawal Proof:

Withdrawals are initiated by transferring funds to an Ethereum address. A single withdrawal proof is generated to verify the transaction's inclusion in an Intmax 2 rollup block. The withdrawal is included in a chain of multiple withdrawals from potentially different senders, with a proof for this chain generated using the `CyclicChainCircuit`. This is done by a withdrawal aggregator, who wraps the final proof in the `ChainEndCircuit`, including the aggregator's address to enable reward collection. The withdrawal aggregator posts the final proof and the chained withdrawal information on Scroll. After successful verification, this triggers the release of funds on Ethereum (the smart contracts are outside the scope of this report; see ChainSecurity’s Intmax 2 Smart Contract Audit Report)

2.4 Claim Proof:

The system rewards users doing privacy mining by allowing them to claim rewards in the form of IntMax tokens. For a deposit to be eligible for a claim, the deposited token must be ETH (token index `0`), and the deposit address must be permitted by the eligibility permitter contract see ChainSecurity's Intmax 2 Smart Contract Audit Report for details). A single claim proof is generated to verify the deposit's inclusion in an Intmax 2 rollup block, and to ensure the maturity of the claim. A claim is considered mature after a pseudo random lock time, computed as `2 days + (PoseidonHash(block_hash||deposit_salt) % 3 days)` (max 5 days lock), where `block_hash` is the hash of the block where the deposit was included in the `deposit_tree`. On top of that, no transfer or withdrawal must be done from the deposit's recipient address between the block where the deposit entered the deposit tree in the public state and the block used in the validity proof for the claim. The claim is included in a chain of multiple claims from potentially different depositors, with a proof for this chain generated using the `CyclicChainCircuit`. This is done by a claim aggregator, who wraps the final proof in the `ChainEndCircuit`, including the aggregator's address to enable further reward collection. The claim aggregator posts the final proof and the chained claim information on Scroll. After successful verification, this triggers the release of reward token on Ethereum.

2.5 Roles and Trust Model

The system operates in a trustless way, except for liquidity onboarding. All actors are untrusted.

Block Builders Anyone can act as a block builder and submit a new block to the rollup smart contract.

Users

Senders: Submit a transaction containing transfers to a block builder. Validates and signs the transaction tree root created by the block builder. After the block has been included, creates an updated balance proof and provides it to the recipients. If a sender attempts to transfer more than his balance his balance for this token will be locked and the funds are stuck irrecoverably.

Recipients: Require the sender's balance proof to access and use the transferred funds.

The `salt` in every user's private state is expected to be random and required to be unique for each transfer and withdrawal. If a `salt` value is reused, the funds associated with it may be lost.

Withdrawal Aggregator

Aggregates withdrawal proofs and posts them on Scroll enabling withdrawal on L1 Ethereum. Anyone can act as withdrawal aggregator.

Withdrawal Contract

The withdrawal smart contract is responsible for verifying the withdrawal proofs and ensuring that the withdrawals transactions are included in an actual block submitted to the rollup contract.

Claim Aggregator

Aggregates claim proofs and posts them on Scroll enabling rewards distribution on L1 Ethereum. Anyone can act as claim aggregator.

Claim Contract

The claim smart contract is responsible for verifying the claim proofs and ensuring that the claims are associated with deposits existing in an actual block submitted to the rollup contract.

Rollup Smart Contract

Source of truth operating on Scroll. Assumed to be always available and never rollback.

While any underlying asset can be mapped to a token id and hence be used within Intmax 2; note that Intmax 2 facilitates raw balance transfer only. Any other token functionality is not supported. Special token (e.g. rebasing tokens etc.) are not supported.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Soundness: The verifier can be convinced of a false statement

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	7

- TransitionWrapper Misses Constraint on Transition_Target.Prev_Next_Account_Id Code Corrected
- Incorrect Exclusion Proof Circuit Allows for Double Spending Code Corrected
- Missing Constraint on pubkey Allows for Double Spending Code Corrected
- Multiple Valid Account Trees / Public States After Applying the Same Block Code Corrected
- Transition Wrapper Underconstrains Main Validation PIs Code Corrected
- Vulnerabilities in BN254 Library Allow for Double Spending and Arbitrary Block Invalidation Code Corrected
- Wrong Genesis State Prevents First Validity Proof Code Corrected

High-Severity Findings	1

- Public State Connection Is Missing a Constraint Code Corrected

Medium-Severity Findings	1

- Uniformity of Hash Function Into Scalar Field Code Corrected

Low-Severity Findings	3

- Genesis Can Reuse Initialize Function Code Corrected
- Redundant Constraints in Single Claim Circuit Code Corrected
- Skipped Range Check on Hash Target Conversion Code Corrected

Informational Findings	6

- Clarifying "Current Block" in Single Claim Circuit Specification Changed
- Unnecessary Vector Conversion Code Corrected
- No Range Checks Code Corrected
- Prev_Block_Number Refers to Current Block Number Code Corrected
- Reuse of Tx_Tree_Root Code Corrected
- Wrong Comments Code Corrected

6.1 TransitionWrapper Misses Constraint on Transition_Target.Prev_Next_Account_Id

Soundness Critical Version 4 Code Corrected

CS-INTMAX2-ZKP-014

As fix for Multiple Valid Account Trees / Public States after applying the same block, an additional field `next_account_id` has been added to the `PublicState`. While the transition_target, using the account_registration or account_update circuit ensures a proper transition of the value, the wrapper circuit does not connect the `transition_target.prev_next_account_id` to the `next_account_id` of the previous public state. Hence the `transition_target.prev_next_account_id` can be chosen arbitrarily. This allows to start inserting accounts at arbitrary higher indexes instead of the subsequent one, leading to multiple valid public states with similar consequences as in the original issue.

Code corrected:

`transition_target.prev_next_account_id` is now connected to the `next_account_id` of the previous public state.

```
builder.connect(
    prev_pis.public_state.next_account_id,
    transition_target.prev_next_account_id,
);
```

6.2 Incorrect Exclusion Proof Circuit Allows for Double Spending

Soundness **Security** **Critical** **Version 1** **Code Corrected**

CS-INTMAX2-ZKP-001

The exclusion proof implemented in the `MembershipProofTarget` is missing the constraint enforcing the key we want to prove exclusion from the tree to be strictly smaller than the lower index leaf's next key. This allows to prove exclusion of a leaf of the tree, even though the leaf is part of the tree.

This can be used in the balance's `UpdateCircuit` to

Example:

state `S`: updated up to block `B`, nonce is `N`

1. Send and sign tx in block `B+1` with nonce `N`
2. Get a validity proof for block `C >= (B+1)`
3. Use the normal sender circuit to send the transfer to the recipient
4. Rollback to previous state `S`
5. Use the broken exclusion proof in the `update_circuit` to convince the system we have no tx until block `C`, we will abuse the membership proof to show we are not in the tree and thus last_block_number=0
6. We made some transfers that are not reflected in our asset_tree

---

Code corrected:

`MembershipProofTarget::verify` was updated to enforce that in the case of an exclusion proof, the target `key` is constrained to be `leaf.key < key` and `key < next.key OR next.key==0`

6.3 Missing Constraint on `pubkey` Allows for Double Spending

Soundness Security Critical Version 1 Code Corrected

CS-INTMAX2-ZKP-002

In the `BalanceTransitionTarget` of `BalanceTransitionCircuit`, in the case of a simple update (`circuit_flags[2]`), the pubkey of the previous balance proof (`prev_balance_pis.pubkey`) is not constrained to be equal to the pubkey used as public input of the update proof (`pis.pubkey`). This allows an attacker to use an update proof for a pubkey that does not have a transaction and skip the update of its own asset tree.

Example:

state `S`: updated up to block `B`, nonce is `N`

1. Send and sign TX in block `B+1` with nonce `N`
2. Get a validity proof for block `C >= (B+1)`
3. Use the normal sender circuit to send the transfers to the recipients
4. Rollback to previous state `S`
5. Use update proof of an arbitrary pubkey that doesn't have a TX to update our balance proof. This works because the pubkey of our balance proof is not connected to the pubkey used in the update proof.
6. We made some transfers that are not reflected in our asset_tree and can spend that value again

In Version 2, the `pubkey` used in the update proof is used as the `pubkey` for the new public inputs of the balance proof, but only the `public_state` of the previous balance proof and the update proof are required to be the same, not the `pubkey`. This allows to duplicate the state of a pubkey `X` to a pubkey `Y`, i.e. `Y` replicates the private state of `X`.

Example:

1. `X` is funded with a deposit and `Y` doesn't have assets. Both are synced at block `B` so they have the same public state.
2. generate an update proof for `Y` for block `B+1`
3. use this update proof in the balance proof of `X`
4. the new balance proof has `Y` for public key, but the private state of `X`

Code corrected:

The `pubkey` used in the update proof is now constrained to be equal to the `pubkey` of the previous balance proof.

6.4 Multiple Valid Account Trees / Public States After Applying the Same Block

Correctness Critical Version 1 Code Corrected

CS-INTMAX2-ZKP-003

The AccountRegistrationCircuit ensures correctness of the `new_account_tree_root` based on the provided vector of `account_registration_proofs`. The expectation that new accounts are added to the account tree at leaf nodes with increasing indexes is not enforced by the account_registration_circuit, any free leaf node can be used to add an account. Hence multiple different but valid account trees with different `account_tree_roots` can be constructed depending on the chosen leaf nodes when adding accounts. Leaf nodes are identified by an index, which for non-empty nodes is used as the account ID.

Users relying on validity proofs may receive a valid proof but, without knowing the correct leaf nodes for the new accounts, cannot reconstruct the tree structure or update their account tree using on-chain data from the Rollup contract to match the resulting root.

Most importantly, this can be used to exploit withdrawals:

1. Construct two distinct valid account trees, resulting in two public states: Public State A (expected state) and Public State B (a special state where our account is added at a different leaf node) after registration block N. Generate validity proofs for both.
2. Perform a transfer in block X and update Public State A as expected. Updating Public State B is not possible as our account is not at the expected index in its account tree, hence block X is invalid for Public State B.
3. In block Y (which is also submitted to the rollup), execute a withdrawal transaction using the expected account ID for Public State B. This block is invalid for the common Public State A (as the account doesn't exist at this index) but valid for our Public State B. Using Public State B, update the balance proof and generate a withdrawal proof. Submit this withdrawal proof on-chain. Since the transaction was part of an actual rollup block (this is verified by the smart contract), the withdrawal succeeds.
4. We still have access to the same funds in Public State A; the expected public state of the Rollup.

Code corrected:

An additional field `next_account_id` has been added to the `PublicState` allowing to track the index of the next new account to be added to the tree. This enables the `account_registration_circuit` to enforce that new accounts are added at consecutive, predetermined indexes. Consequently, the account tree and public state are uniquely determined after applying the same block.

6.5 Transition Wrapper Underconstrains Main Validation PIs

Security Soundness Critical Version 1 Code Corrected

CS-INTMAX2-ZKP-004

Both the `TransitionWrapperCircuit` and `ValidityTransitionTarget` define some `block_pis` representing the public inputs of a main validation proof that should be operated on. The ones in `TransitionWrapperCircuit` come directly from the public inputs of the main validation proof and are verified there, but they remain unconnected to those in `ValidityTransitionTarget`.

This allows to run the main validation and validity transition circuit with different values for `block_pis`, which should be the same. While this leads to different public inputs of the validity proof and hence the modification is visible, this proof can be used locally to do double spending.

Example:

After sending a transaction, an attacker can set an arbitrary `sender_tree_root` in the `block_pis` of `ValidityTransitionTarget` to choose which leaves of the account tree they want to update. Leading to an account tree where their transaction will not appear. They can use the "true" validity proof for the recipients so they can receive the value. Then, they can use the forged proof to skip updating their own balance, allowing them to reuse the value later.

Code corrected:

The function `TransitionWrapperCircuit::new()` has been updated to connect the targets from the `block_pis` of the main validation proof to the `block_pis` of the transition circuit.

6.6 Vulnerabilities in BN254 Library Allow for Double Spending and Arbitrary Block Invalidation

Soundness **Security** **Critical** **Version 1** **Code Corrected**

CS-INTMAX2-ZKP-005

Some of the vulnerabilities uncovered in `plonky2-bn254` library allow manipulation of different computations:

- the result of `is_recoverable_from_x()` (using a vulnerable STARK), used in `is_valid_format()` to check the validity of the X coordinate of a public key. By modifying this result, it is possible to create a fraud proof for every valid block.
- the result of `g1_msm()` (using a vulnerable STARK), used in `verify_aggregation()` to rebuild the aggregated public key. By modifying this result, it is possible to create a fraud proof for every valid block.
- the result of `hash_to_g2_circuit()` (using a vulnerable STARK), used in `is_valid_format()` to rebuild the message point. By modifying this result, it is possible to create a fraud proof for every valid block.

This can also be used to double spend:

1. Sign and send a transaction
2. Wait for the block (B) to be included, update and send the balance proof to the recipient
3. Rollback our local state before 1.
4. Create a validity proof for `B` with `is_valid=false` and update our balance with a spent proof
5. Wait for at least one block after `B` and use the validity proof to update the balance with a update proof
6. We made some transfers that are not reflected in our asset_tree and can spend that value again

Under certain circumstances, it may be possible to produce a valid proof for an invalid block. This could also lead to double spending.

Code corrected:

The `plonky2-bn254` library has been fixed and the latest commit is used as a dependency for Intmax 2 ZKP. The report for the `plonky2-bn254` library can be found at https://www.chainsecurity.com/security-audit/plonky2-bn254-keccak256.

6.7 Wrong Genesis State Prevents First Validity Proof

Correctness **Critical** **Version 1** **Code Corrected**

CS-INTMAX2-ZKP-006

The recursive validity proof proves the correctness of a block and the resulting transition of the public state. It ensures the transition begins from the public state resulting from the previous block; initially, this is the genesis public state.

```
pub fn genesis() -> Self {
    let mut block_tree = BlockHashTree::new(BLOCK_HASH_TREE_HEIGHT);
    block_tree.push(Block::genesis().hash());
    let account_tree = AccountTree::initialize();
    let deposit_tree_root = DepositTree::new(DEPOSIT_TREE_HEIGHT);
    let block_hash = Block::genesis().hash();
    let block_number = 0;
    Self {
        block_tree_root: block_tree.get_root(),
        prev_account_tree_root: PoseidonHashOut::default(),
        account_tree_root: account_tree.get_root(),
        deposit_tree_root: deposit_tree_root.get_root(),
        block_hash,
        block_number,
    }
}
```

The first valid block must be a registration block that adds accounts to the account tree. To validate this block, exclusion proofs are verified to ensure that each account isn't already in the tree.

With a genesis account tree root of `prev_account_tree_root: PoseidonHashOut::default()`, which is not the valid root of an initialized (only containing the dummy key at index `0`) account tree, no proofs can be generated. Hence no validity proof for the first block can be generated.

Code corrected:

The `prev_account_tree_root` of the genesis state has been updated to be the root of the initialized account tree.

6.8 Public State Connection Is Missing a Constraint

Security Soundness High Version 4 Code Corrected

CS-INTMAX2-ZKP-013

The `connect()` function of `PublicStateTarget` does not constrain `next_account_id` of the two public states to have the same value.

Code corrected:

The missing constraint to connect `prev_account_tree_root` of the two public states has been added.

```
self.prev_account_tree_root
    .connect(builder, other.prev_account_tree_root);
```

6.9 Uniformity of Hash Function Into Scalar Field

Design | Medium | Version 1 | Code Corrected

CS-INTMAX2-ZKP-012

Intmax 2 ZKP implements a slightly different version of BDN18 (https://eprint.iacr.org/2018/483). The implementation of the hash function H_1 has a minor, but non-negligible skew. The hash function takes the following steps:

1. Compute 8 challenges (Poseidon hashes) from $pk_i || keccak256(pk_0 || pk_1 || ... || pk_n)$
2. Build a 256 bits value from the 32 lower bits of the challenges computed at the previous point
3. Take the 256bits value modulo r, the size of the scalar field of BN254, which has 254 bits.

By taking a 256bits value modulo the 254bits r, because of the small difference of 2 bits, some residue classes in the scalar field will be hit more often than the others, leading to a skewed distribution.

Code corrected:

Instead of computing 8 challenges, the code has been updated to compute 16 challenges, leading to a 512bits number taken modulo r, which makes the distribution skew negligible.

6.10 Genesis Can Reuse Initialize Function

Design | Low | Version 6 | Code Corrected

CS-INTMAX2-ZKP-015

For consistency and to avoid code duplication, the `Block::genesis()` and `PublicState::genesis()` functions should use the `initialize()` function from `DepositTree` and `BlockHashTree`. This will ensure the trees to always be initialized in the same way across the codebase.

Code corrected:

The codebase has been updated to use the `initialize()` function of the concerned trees instead of using the `new()` function along with the size of the trees.

6.11 Redundant Constraints in Single Claim Circuit

Design | Low | Version 6 | Code Corrected

CS-INTMAX2-ZKP-016

The single claim circuit enforces that `last_block_number` is strictly less than `deposit_time_pis.block_number`:

```
// assert last_block_number < deposit_time_pis.block_number
let diff = builder.sub(deposit_time_pis.block_number, last_block_number);
builder.range_check(diff, 32);

let zero = builder.zero();
let is_diff_zero = builder.is_equal(diff, zero);
builder.assert_zero(is_diff_zero.target);
```

Since a deposit cannot be transferred in the same block it is first included, enforcing a non-strict inequality (<=) would be sufficient. The additional constraints that forbid equality are redundant.

Before applying a deposit update to a balance proof, the balance proof must first be updated to a public state which includes the deposit block. If a transaction for the account is included in the same block as the deposit, the account's leaf in the account tree (last_block_number) will have been updated to that block. Hence the balance proof update must be done using the sender proof. Thus, the order is guaranteed: any outgoing transaction must be processed before the deposit can be applied. A deposit cannot be spent in the same block.

Code corrected:

The redundant constraints have been removed, the circuit now enforces a non-strict inequality.

6.12 Skipped Range Check on Hash Target Conversion

Soundness **Design** **Low** **Version 1** **Code Corrected**

CS-INTMAX2-ZKP-007

The function `Bytes32Target::from_hash_out()` in `poseidon_hash_out.rs` takes a `PoseidonHashOutTarget` as input, and splits each element it into two chunks of `32bits`, referred as `high` and `low`. This function is mainly used in the context of nullifier and deposit trees. To achieve its goal, Plonky2's `split_low_high()` is used, `num_bits` should represent the bit size of the input to split. In the case of `Bytes32Target::from_hash_out()`, `num_bits` is set to `32bits`, even though it should be `64bits` as the input value is in the Goldilocks field.

For a Goldilock element `X` of `PoseidonHashOutTarget`, this has as an effect that the range check on the `high` part is disabled, allowing to set the witness to arbitrary values `H,L` satisfying `X = H * (1<<32) + L`, with only `L < (1<<32)`, instead of `H,L < (1<<32)`. `X` is fixed and cannot be chosen arbitrarily, heavily limiting any advantage an attacker could get by abusing the missing range check.

Code corrected:

The range check on `high` has been added. On top of this change, the constraint that if `hi == 2**32 - 1`, then `lo == 0`. This new constraint enforces the unique decomposition of an `x < 2**32`.

6.13 Clarifying "Current Block" in Single Claim Circuit

Informational **Version 6** **Specification Changed**

CS-INTMAX2-ZKP-017

The comments and terminology around "current" in `single_claim_circuit.rs` are imprecise and might be confusing.

The description of `SingleClaimTarget` reads `current block hash` and `current block number`.

Comments in `single_claim_circuit.rs` mention `no transfer occurred during the lock period` multiple times. The first comment is more precise and states: `Verifying no transfers occurred during the lock period (using account tree's last block number)`. This is what the circuit actually enforces: no transfer occurred between the deposit block and the current block selected for the proof.

The description could be a little more precise by clarifying that current block means a block after the deposit lock time has expired, but before this account sent any transaction. If there is a transfer of this account after the lock time, the proof must not use the latest block but instead a block after the lock time expired and before this first transfer.

The term "current" may be misleading, even though the circuit logic is correct.

Specification changed:

The comments have been updated and now read "the time of claim" for more clarity.

6.14 Unnecessary Vector Conversion

Informational **Version 6** **Code Corrected**

CS-INTMAX2-ZKP-018

In the function `BlockSignPayloadTarget::message_point()`, `to_vec()` is called twice on `self`. The second `to_vec()` has no effect.

Code corrected:

The second `to_vec()` has been removed.

6.15 No Range Checks

Informational **Version 1** **Code Corrected**

CS-INTMAX2-ZKP-008

`LeafableHasher` for `KeccakLeafableHasher::hash_out_target()` instantiates a `Bytes32Target` with the range check being disabled:

```
fn hash_out_target<F: RichField + Extendable, const D: usize>(
builder: &mut CircuitBuilder<F, D>,
) -> Self::HashOutTarget {
Bytes32Target::new(builder, false)
}
```

In the following cases, range checks are not performed on certain fields when the `is_checked` flag is set to `true`:

`PublicStateTarget::new()` does not perform a range check on `block_number` to ensure it is within 32 bits, even when `is_checked` is `true`. Similarly, `TransferTarget::constant()` does not constrain `token_index` to be less than 32 bits.

In connection with the smart contracts, it is guaranteed that both `block_number` and `token_index` are 32 bits.

Code corrected:

Range check has been added in `hash_out_target()`.

6.16 Prev_Block_Number Refers to Current Block Number

Informational Version 1 Code Corrected

CS-INTMAX2-ZKP-009

In `ValidityTransitionTarget::new()` a target `prev_block_number` represents the current block number of the block being processed. This misleading name negatively affects the readability of the code.

Code corrected:

`prev_block_number` has been renamed `block_number`

6.17 Reuse of Tx_Tree_Root

Informational Version 1 Code Corrected

CS-INTMAX2-ZKP-010

The block validation circuit determines whether a posted block is valid based on signature verification, sender duplication and account tree lookup. For signature content of a non-registration blocks this validation passes for any block in the future. A malicious block builder can reuse a past `tx_tree_root` and the signatures of a previous non-registration block. Such a block and the resulting transition will be valid. The involved accounts must recognize the repeated transaction and update their balance proof using a sender proof for the invalid repeated transaction.

Code corrected:

In Version 6, users sign a block payload instead of only the `tx_tree_root`. That block payload contains the `tx_tree_root`, but also a specific block builder and its nonce, as well as an expiry. This makes it impossible for another malicious block builder to reuse the signatures, as the block builder in the signatures and the block builder of the actual block will not match, making the block invalid.

6.18 Wrong Comments

Informational Version 1 Code Corrected

CS-INTMAX2-ZKP-011

The comment `// assert last_block_number <= validity_pis.public_state.block_number` in `update_circuit.rs` does not express the constraint that is enforced in the circuit.

Code corrected:

The comment has been updated to `// assert last_block_number <= prev_public_state.block_number`

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Insufficient Amount for Transfer

Note Version 1

Users must be aware that if they submit and sign a transfer sending more than their balance in token `T`, the transfer cannot be received, and the sender's balance for that token will be frozen and the remaining tokens are locked.

7.2 Transfers and Deposits in the Same Block

Note Version 1

If a user has a balance `X` and deposits an amount `Y` that will be included in block `N`, the user must not expect to be able to transfer an amount `Z > X` in the block `N`, as the transfer will effectively be taken into account before the deposit. As specified in the System Overview, transferring an amount greater than the balance will lock the token for that user.

7.3 Validity Proof Generation Time

Note Version 1

Generating validity proofs for blocks is computationally expensive and takes time. The docs state that validity proof generation should take around 1 minute, with private discussions suggesting this could be reduced to 15 seconds. Since validity proofs are cyclic, generating the next proof depends on the previous one, though parts of the process could be parallelized.

Block builders can post new blocks without the validity proof. A rate limiter in the smart contracts enforces a penalty for block builders that post blocks too fast.

The contract keeps an exponential moving average of the interval between block produced which is computed as $emaInterval = \alpha * interval + (1 - \alpha) * emaInterval$. And the penalty paid to the system for posting a block too fast (if $targetInterval > emaInterval$) is computed as follows: $penalty = k * (targetInterval - emaInterval)^2$.

The effectiveness of this and the appropriate choice of parameters is out of scope of this review. Note that publishing invalid blocks is no longer punished by the Intmax 2 system. **It is assumed that block validity proving will be able to keep up with the block producing rate.**

7.4 Zero-Knowledge Property Depends on Usage

Note Version 1

Transaction privacy may be compromised, especially in cases of low utilization, where patterns in transfers can be inferred from deposits or withdrawals. For non-fungible tokens, these patterns are even more apparent.

Examples:

- If the utilization of the chain is low, it may be easier to link a deposit with a withdrawal.
- If an amount X of token is deposited and the same amount is withdrawn shortly after, one may suspect some correlation.

In later versions of the reviewed code (Version 6) `privacy mining` was introduced aiming to incentivize users to deposit the native asset for a pseudorandom lock period in exchange for rewards (enabled by the new claim feature). This increases network utilization and particularly the number of deposits and withdrawals of the native token on L1.