yAudit

November 24, 2025

Prepared for
WORM

Audited by
qpzm
nullity

WORM (Proof of Burn)

ZK Circuit Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

WORM is a privacy-preserving, cryptographically scarce ERC-20 token protocol that enables users to convert ETH into a new asset through irreversible destruction (burning) at stealth addresses. The protocol leverages zero-knowledge proofs (zk-SNARKs) to prove ETH burns without revealing burn addresses or transaction details, implementing the EIP-7503 Private Proof-of-Burn standard. It uses a two-token model: BETH (1:1 burn receipt) and WORM (time-released scarce token minted at 50 WORM per 30-minute epoch).

1.2 Audit Scope

This security review covers the core Circom circuits. Solidity smart contracts are referenced for context. This audit covers 20 circuits totaling approximately 2,119 lines of code across 8 days of review. An additional one day was reserved to review a feature made post all fixes.

```
circuits/
├── proof_of_burn.circom
├── spend.circom
├── main_proof_of_burn.circom
├── main_spend.circom
└── utils/
    ├── rlp/
    │   ├── merkle_patricia_trie_leaf.circom
    │   ├── integer.circom
    │   └── empty_account.circom
    ├── substring_check.circom
    ├── burn_address.circom
    ├── proof_of_work.circom
    ├── public_commitment.circom
    ├── keccak.circom
    ├── convert.circom
    ├── concat.circom
    ├── constants.circom
    ├── assert.circom
    ├── array.circom
    ├── selector.circom
    ├── shift.circom
    └── divide.circom
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	3
Low	3
Informational	8

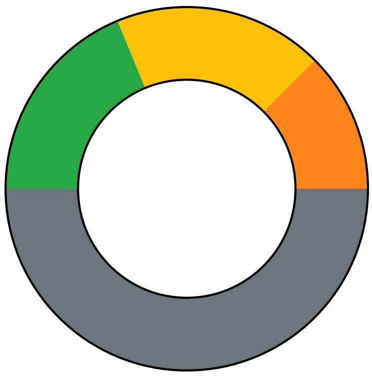

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The protocol demonstrates solid architectural design with effective use of zero-knowledge proofs for burn verification. One high-severity issue remains: privacy leakage through coin lineage tracking (deferred for future work). The codebase makes reasonable security assumptions - 10 ETH balance caps and 16-layer MPT depths provide adequate protection margins - but these assumptions lack runtime validation. Test coverage is insufficient for a protocol handling potentially significant value. While collision attacks are infeasible with current technology, this may change over time. Overall the code demonstrates strong cryptographic foundations with well-chosen primitives and sound zero-knowledge proof implementation, but exhibits gaps in defensive programming, privacy architecture, and test coverage.

2 Audit Overview

2.1 Project Information

Protocol Name: WORM
Repository: https://github.com/worm-privacy/proof-of-burn
Commit Hashes:

- 0e5237480fbac83aa4291a9e06618b4318da3585
- a39690248ed5ba70b9c1e14b543bd693ce416ccf

Commit URLs:

- https://github.com/worm-privacy/proof-of-burn/tree/0e5237480fbac83aa4291a9e06618b4318da3585
- https://github.com/worm-privacy/proof-of-burn/tree/a39690248ed5ba70b9c1e14b543bd693ce416ccf

2.2 Audit Team

qpzm, nullity

2.3 Audit Resources

Code repositories and documentation

2.4 Protocol Overview

2.4.1 Burn Address Generation

The WORM protocol begins with the user generating a secret `burnKey`. Using the Poseidon hash function, the user derives a burn address by hashing the `burnKey` (along with other parameters) and truncating the result to a 160-bit Ethereum address. The user then sends ETH to this burn address, effectively destroying the ETH since no private key exists for the address.

2.4.2 Zero-Knowledge Proof Construction

After burning ETH, the user constructs a zero-knowledge proof (zk-SNARK). This proof demonstrates that ETH exists at the burn address (verified via a Merkle Patricia Trie proof), that the user knows the `burnKey` used to generate the address, and that the burn satisfies all protocol rules. The proof is privacy-preserving and does not reveal which specific burn address was used or the amount of ETH involved.

2.4.3 Minting WORM Tokens

To claim WORM tokens, the user submits the proof to the smart contract through the `mintCoin()` function. The contract verifies the proof and allows the user to specify the amount to withdraw immediately (`revealedAmount`), a fee for the prover or broadcaster, and the remaining encrypted balance (`remainingCoin`) if the full amount is not claimed. The contract mints WORM tokens up to a maximum of 10 ETH per proof and uses a nullifier to prevent double-claims from the same burn.

2.4.4 Spending Remaining Balance

If the user did not claim the entire balance initially, they can later spend the remaining balance by creating a lightweight spend proof. This proof demonstrates knowledge of the original `burnKey` , verifies that the remaining balance is sufficient to cover the withdrawal and fee, and produces a new `remainingCoin` with the updated balance. The user submits this proof to the `spendCoin()` function, which verifies the proof, invalidates the old coin, mints additional tokens, and stores the new `remainingCoin` . This process can be repeated until the balance is fully withdrawn.

2.5 Circuit Overview

2.5.1 Main Circuits

`main_proof_of_burn.circom`

- Proves ETH was burned to a derived address on Ethereum
- Verifies MPT inclusion proof showing balance exists in state tree
- Checks PoW constraint (16-bit), balance limits (less than 10 ETH), and burn address derivation
- Outputs: nullifier, remainingCoin, commitment for minting tokens

`main_spend.circom`

- Proves ownership of an existing coin and ability to spend part of it
- Verifies arithmetic: `balance >= withdrawnBalance + fee`
- Outputs: old coin, new remainingCoin with updated balance

2.5.2 Core Utility Circuits

`proof_of_work.circom` - ProofOfWorkChecker()

- Verifies `Keccak256(burnKey || revealAmount || burnExtraCommitment || "EIP-7503")` has N leading zero bytes
- Creates 31-byte mask and checks first N bytes of hash are zero

`burn_address.circom` - BurnAddress()

- Derives Ethereum address:

```
 uint160(
Poseidon4(
   POSEIDON_BURN_ADDRESS_PREFIX, burnKey, revealAmount, burnExtraCommitment
 ))
```

- Truncates 254-bit Poseidon hash to 160-bit Ethereum address

`merkle_patricia_trie_leaf.circom` - RlpMerklePatriciaTrieLeaf()

- Decodes RLP-encoded MPT leaf node containing account data
- Extracts balance, verifies address hash matches, checks minimum nibble requirements

`substring_check.circom` - SubstringCheck()

- Verifies one byte array appears as substring in another
- Used to check each MPT layer's hash appears in parent layer's RLP data

`keccak.circom` - KeccakBytes()

- Computes Keccak-256 hash of variable-length byte array
- Used for MPT node hashing and block root computation

`public_commitment.circom` - PublicCommitment()

- Computes

```
 Keccak256(
blockRoot,
nullifier,
remainingCoin,
revealAmount,
burnExtraCommitment,
extraCommitment)[:31]
```

- Single public output that commits to all proof parameters

2.5.3 Helper Circuits

Basic utilities: `convert.circom` (field - bits - bytes conversions), `assert.circom` (range checks and inequalities), `concat.circom` (array concatenation), `selector.circom` (array element selection), `array.circom` (filtering and array manipulation), `rlp/integer.circom` (RLP integer encoding/decoding), `rlp/empty_account.circom` (Ethereum account structure encoding).

2.6 Assumptions and Invariants

2.6.1 Protocol Parameters

- **Maximum Balance**: `balance ≤ 10 ETH` — Limits economic incentive for collision attacks (max gain ~$30k vs attack cost ~$5B).
- **Proof-of-Work**: First 2 bytes of `Keccak256(burnKey || receiver || fee || "EIP-7503")` must be zero (16-bit PoW). On average, ~65,536 hash attempts needed per valid burnKey.
- **Minimum Leaf Nibbles**: At least 50 nibbles (200 bits) of the address hash must appear in the MPT leaf node. Users may relax this by up to 25 bytes, but must add 8 bits of PoW per relaxed byte.
- **MPT Depth Limit**: Maximum 16 layers. Empirical data: min 8, max 10, avg 8.69 (from 100 richest addresses).

2.6.2 Ethereum State Assumptions

- **EVM Chain Compatibility**: Assumes all EVM-compatible chains use the same block header structure and MPT implementation as Ethereum mainnet. Some L2s and EVM chains (e.g., Monad) differ. **Full end-to-end testing is required before deploying on any L2 or EVM-compatible chain.**
- **StateRoot Position**: Assumes state root is at byte 91 of the block header (with a 3-byte RLP prefix). This breaks if the header exceeds 64KB (current max ~1KB, so 60× safety margin).
- **MPT Node Size**: Maximum 544 bytes (4 blocks × 136 bytes). Largest observed: 532 bytes (branch node with 16 children). Future Ethereum upgrades may require circuit updates.

2.6.3 Field Arithmetic Constraints and Cryptographic Assumptions

- **Integer Size Limit**: All numeric values must be less than 31 bytes (248 bits) to avoid bn254 field overflow (field size $\sim 2^{254}$). Enforced via `assert(N <= 31)` in conversion circuits.
- **Burn Address Security**: `burnAddress` = `uint160(Poseidon4(PREFIX, burnKey, revealAmount, burnExtraCommitment))` — No known private key exists; ETH sent to burn addresses is permanently locked.
- **Nullifier Uniqueness**: `nullifier = Poseidon2(PREFIX, burnKey)` — Each burnKey yields a unique nullifier. Reusing a burnKey for multiple burns causes a collision (second claim fails).

- **Hash Collision Resistance**: MPT verification relies on Keccak-256 collision resistance. Finding a collision would require $\sim 2^{128}$ operations (cryptographically infeasible).

2.7 Critical Findings

None.

2.8 High Findings

2.8.1 Missing leaf node validation allows forged Merkle proofs via contract account injection

Technical Details

The `ProofOfBurn` circuit verifies Merkle-Patricia Trie structure by checking that the last layer is a valid account leaf with an empty code hash (EOA), and that each layer's keccak hash appears as a substring in its parent layer. However, the circuit does not validate that intermediate layers are NOT account leaf nodes themselves.
In Ethereum's state trie, there are two types of account leaves:

1. **EOA (Externally Owned Account)**: `RLP([nonce, balance, EMPTY_STORAGE_HASH, EMPTY_CODE_HASH])`
2. **Contract Account**: `RLP([nonce, balance, storage_root, code_hash])` where `code_hash` and `storage_root` are 32-byte hashes

The vulnerability arises because:

1. The circuit only validates that **the last layer** (`layers[numLayers - 1]`) is a properly formed EOA leaf:

```
signal (leaf[maxLeafLen], leafLen) <== RlpMerklePatriciaTrieLeaf(32, amountBytes)(
    addressHashNibbles, numLeafAddressNibbles, balance
);
for(var i = 0; i < maxLeafLen; i++) {
    leaf[i] === lastLayer[i];
}
```

proof*of*burn.circom#L192-L199

2. Intermediate layers (`layers[0]` through `layers[numLayers - 2]`) are only verified via substring checks:

```
substringCheckers[i - 1] <== SubstringCheck(maxNodeBlocks * 136, 31)(
    subInput <== reducedLayerKeccaks[i],
    mainLen <== layerLens[i - 1],
    mainInput <== layers[i - 1]
);
```

proof*of*burn.circom#L171-L175

3. No check exists to ensure intermediate layers are branch/extension nodes rather than account leaves:

This means an attacker can use a CONTRACT ACCOUNT leaf as an intermediate layer. Since contract account leaves contain 32-byte fields (`storage_root` and `code_hash`), the attacker can craft scenarios where these hash fields match the keccak hash of a fake child layer.
Attack scenario (bytecode case):
The attacker exploits the fact that the circuit doesn't validate that layers are actual MPT nodes. The attacker can provide raw bytecode as a "layer" even though bytecode is not part of the state trie structure.
Step 1: Craft the attack from bottom-up:
`ATTACKER_LEAF` =
`RLP([fake_address_hash_nibbles, RLP([0, 999 ETH, EMPTY_STORAGE_HASH, EMPTY_CODE_HASH])])`
where `fake_address_hash_nibbles` = nibbles of `keccak256(fake_burn_address)` and `ATTACKER_LEAF` = fake burn address leaf claiming arbitrary balance
Step 2: Create bytecode containing the hash of `ATTACKER_LEAF`:
`bytecode = 0x60<32_bytes_of_keccak(ATTACKER_LEAF)>...`
Step 3: Deploy contract with crafted bytecode:
Deploy contract at some address, which creates:

```
  contract_leaf = RLP([
  contract_address_hash_nibbles,
  RLP([nonce, balance, storage_root, code_hash])
])
```

where `contract_address_hash_nibbles` = nibbles of `keccak256(contract_address)` and `code_hash` = `keccak256(bytecode)`
Step 4: Craft fake proof path with 4 layers:

- `layers[0]` = `branch_node` (legitimate, contains `keccak(contract_leaf)`)
- `layers[1]` = `contract_leaf` (`CONTRACT_ACCOUNT` leaf - pretending to be intermediate node!)
- `layers[2]` = `bytecode` (NOT an MPT node! Just raw bytecode bytes containing `keccak(ATTACKER_LEAF)`)
- `layers[3]` = `ATTACKER_LEAF` (fake burn address leaf)

Step 5: Circuit verification (all checks PASS incorrectly):

- Check 1 (i=1): Is `keccak(contract_leaf)[0:31]` substring of `branch_node`? Yes, `contract_leaf` is a real account in the state trie
- Check 2 (i=2): Is `keccak(bytecode)[0:31]` substring of `contract_leaf`? Yes, The `contract_leaf` contains `code_hash = keccak256(bytecode)`
- Check 3 (i=3): Is `keccak(ATTACKER_LEAF)[0:31]` substring of bytecode? Yes, The bytecode physically contains `keccak256(ATTACKER_LEAF)` as embedded data.
- Check 4: Is `ATTACKER_LEAF` a valid EOA leaf? Yes.
- Result: Circuit accepts `ATTACKER_LEAF` as existing in state root!

Key insight:
The circuit doesn't validate that layers represent actual MPT nodes. The attacker exploits this by:

1. Using a contract account leaf at Layer N (contains `code_hash` field)
2. Providing raw bytecode as Layer N+1 (NOT a real MPT node!)
3. The bytecode contains `keccak(ATTACKER_LEAF)` as embedded data
4. Using the fake `ATTACKER_LEAF` as Layer N+2

Visual diagram (`bytecode` variant):

```
Legitimate path (what circuit expects):
  State Root -> Branch -> Branch -> EOA Leaf
                (layers[0])  (layers[1])  (layers[2])
                             ^
                             All intermediate layers should be branch/extension nodes

Malicious path (bytecode attack):
  State Root -> Branch -> Contract Leaf -> Bytecode -> Fake EOA Leaf
                (layers[0]) (layers[1])    (layers[2])  (layers[3])
                              ^                ^
                              |                |
                    CONTRACT ACCOUNT leaf    NOT an MPT node!
                    (circuit accepts this     Just raw bytecode with
                     as intermediate layer!)  keccak(layers[3]) embedded
                    Contains code_hash
```

Key vulnerabilities:

1. No validation that `layers[1]` is a branch/extension node (not a leaf)
2. No validation that `layers[2]` is an actual state MPT node (not arbitrary data)

Alternative attack (`storage_root` variant):
The same attack works with `storage_root` instead of `code_hash`. The attacker can:

1. Deploy a contract and use SSTORE to write the complete `ATTACKER_LEAF` structure as a storage value: `sstore(slot, ATTACKER_LEAF)` where `ATTACKER_LEAF = RLP([fake_address_hash_nibbles, RLP([0, 999 ETH, ...])])` and `fake_address_hash_nibbles = nibbles of keccak256(fake_burn_address)`
2. The storage trie will have a leaf: `RLP([storage_key_hash_nibbles, RLP(ATTACKER_LEAF)])` where `storage_key_hash_nibbles = nibbles of keccak256(storage_slot)`
3. Provide the storage trie path (storage branch/extension nodes -> storage leaf with `ATTACKER_LEAF`)
4. The contract's `storage_root` equals the root of this storage trie, creating a valid link
5. The substring check finds `ATTACKER_LEAF` (the raw bytes) within the storage leaf's RLP encoding

Visual diagram (`storage_root` variant):

```
Malicious path (storage_root attack):
  State Root -> Branch -> Contract Leaf -> Storage Branch -> Storage Leaf
                (layers[0]) (layers[1])    (layers[2])      (layers[3])
                              ^                                  ^
```

```
                    |                                    |
          CONTRACT ACCOUNT leaf                Storage leaf = Fake EOA Leaf
          (circuit accepts this                RLP([key, RLP(ATTACKER_LEAF)])
           as intermediate layer!)             The value IS the ATTACKER_LEAF
          Contains storage_root
```

Attack setup:

1. Deploy contract and execute: `sstore(slot_x, ATTACKER_LEAF)`
2. Storage trie now has leaf containing `ATTACKER_LEAF` as the stored value
3. Craft fake proof path with 4 layers:

- `layers[0]` = `branch_node (legitimate, contains keccak(contract_leaf))`
- `layers[1]` = `contract_leaf` (CONTRACT ACCOUNT leaf - pretending to be intermediate node!)
- `layers[2]` = `storage_branch` (storage trie branch node)
- `layers[3]` = `storage_leaf = RLP([key, RLP(ATTACKER_LEAF)])` (This is the fake EOA leaf!)

4. Circuit verification:

- Substring check finds `ATTACKER_LEAF` (raw bytes) within `storage_leaf`'s RLP encoding
- Final layer (`storage_leaf`) is validated as the `ATTACKER_LEAF` claiming 999 ETH

Key insight:
`SSTORE` lets attacker write arbitrary values, so they can store the entire `ATTACKER_LEAF` structure as a storage value. The storage leaf containing this value becomes the final validated layer.

Impact

High - Complete bypass of Merkle proof validation allowing theft of arbitrary funds.
An attacker can:

1. Deploy a contract with specially crafted bytecode containing `keccak(fake_leaf)` as embedded data
2. Submit a proof that claims any arbitrary burn address exists with any balance amount
3. Mint unlimited BETH tokens backed by non-existent burns
4. The attack cost is just the gas to deploy a contract (~0.01-0.1 ETH), while the profit can be unlimited

Recommendation

The issue was reported and fixed in PR#15 by the WORM team during the audit

Developer Response

Found by @keyvank and fixed through a `LeafDetector` component that counts the number of leaf-looking layers and enforces it to be 1.

2.8.2 Burn Transaction Traceability Enables Source Address Identification

Technical Details

The BETH.sol contract maintains coin lineage to enforce the 10 ETH mint cap:

```
mapping(uint256 => uint256) public coins;  // coin -> rootCoin
mapping(uint256 => uint256) public revealed; // Total revealed per rootCoin
```

```
uint256 rootCoin = coins[_coin];
// ...
coins[_remainingCoin] = rootCoin;  // Links new coin to SAME rootCoin
revealed[rootCoin] += _amount + _fee;  // Accumulates on SAME rootCoin
require(revealed[rootCoin] <= MINT_CAP, "Mint is capped!");
```

This creates a permanent on-chain transaction graph where every spend is linked back to the original burn. Any observer can reconstruct the complete spending history by finding all `mintCoin` events, tracing all `spendCoin` events where `coins[coin] == rootCoin`, calculating total revealed amounts, and correlating with mainnet burns. The protocol leaks the original burn amount (bounded by `revealed[rootCoin]`), complete spending history (all amounts, receivers, timing), and all recipient addresses.

Impact

High. Privacy is achieved for burn -> mint unlinkability via `BETH.mintCoin`, but compromised for BETH circulation.
The rootCoin linkage enables:

- Full traceability of all spending history from the same root (anonymity set = 1)
- Amount correlation via `revealed[rootCoin]` accumulator
- Chain-wide deanonymization from any single identified address

This creates a permanent on-chain transaction graph for all post-mint BETH activity.

Recommendation

Replace coin lineage tracking with a mixing tree to break linkability. Remove `coins` and `revealed` mappings and implement a commitment tree with root history buffer. Update circuits to use separate nullifiers per spend with commitments computed as `Poseidon(burnKey, nullifier, balance)`. Track `nullifierHashes` for double-spend prevention. Reference Privacy Pools architecture for implementation patterns.

Developer Response

I want to decide on this later. Let's just assume it's ok for partial revealed amounts to be linked with each other and figure something out later. And do a separate audit for that. Also, part of the reason we added Spend functionality is to allow users to "deny" their relationship with ETH transfers with same amounts by being able to claim that there is some remaining balance that can still be minted (Even when there is 0 balance left)

2.9 Medium Findings

2.9.1 BurnKey Reuse Causes Permanent Loss of User Funds

Technical Details

The circuit computes nullifiers based solely on the `burnKey` parameter:

```
signal nullifier <== Poseidon(2)([POSEIDON_NULLIFIER_PREFIX(), burnKey]);
```

The nullifier depends only on `burnKey` and does not include the burn address, receiver, fee, or balance. Meanwhile, the burn address is derived from multiple inputs:

```
signal hash <== Poseidon(4)([POSEIDON_BURN_ADDRESS_PREFIX(), burnKey,
receiverAddress, fee]);
addressBytes <== Fit(32, 20)(hashBytes);  // Truncate to 160 bits
```

This creates a one-to-many relationship where one `burnKey` generates multiple distinct burn addresses (by varying `receiverAddress` or `fee`), but all produce the same nullifier. The smart contract enforces nullifier uniqueness via `require(!nullifiers[_nullifier], "Nullifier already consumed!")`, meaning only the first proof using a given `burnKey` can be accepted.
A user burns 3 ETH to address `A` using `(burnKey=999, receiver=Alice, fee=0)` and successfully mints 3 ETH of tokens. Later, the user reuses `burnKey=999` for a second burn of 2 ETH to address `B` with `(burnKey=999, receiver=Bob, fee=1)`. The two burns create different on-chain addresses (`burnAddress_A ≠ burnAddress_B`), but both produce `nullifier = Poseidon2(NULLIFIER_PREFIX, 999)`. When attempting to submit the second proof, the transaction reverts with "Nullifier already consumed!" and the 2 ETH in address `B` becomes permanently unclaimable.

Impact

Medium. Users who accidentally reuse `burnKeys` across multiple burns suffer permanent loss of funds from all subsequent burns. The funds are provably burned on-chain but become irretrievable due to nullifier collision. There is no recovery mechanism, and nothing prevents users from making this mistake. This also forces users to claim their entire balance in a single proof rather than splitting claims across multiple transactions, eliminating flexibility and increasing transaction risk.

Recommendation

Modify the nullifier to include the burn address hash, making each burn address independently claimable:

```
signal addressHashNibbles[64] <== BurnAddressHash()(burnKey, receiverAddress,
fee);
signal addressHashNum <== Nibbles2Num(64)(addressHashNibbles);
signal nullifier <== Poseidon(2)([POSEIDON_NULLIFIER_PREFIX(), addressHashNum]);
```

This ensures different burn addresses produce different nullifiers, allowing safe `burnKey` reuse. Alternatively, document the single-use requirement prominently in contracts and user interfaces with explicit warnings that reusing `burnKeys` results in permanent fund loss.

Developer Response

Burn-key should never be reused. That's part of the protocol. My choice is to document this and warn devs not to use same burn-key multiple times.

2.9.2 MPT Depth Limit May Reject Valid Proofs

Technical Details

The circuit sets `maxNumLayers = 16` based on observational data showing a current maximum of 10 nodes among 100 sampled addresses:

```
// 16 -> maxNumLayers (Maximum number of Merkle-Patricia-Trie proof nodes supported)
//       Number of MPT nodes in account proofs of 100 richest addresses as of July 2nd 2025:
//       Min: 8 Max: 10 Avg: 8.69
component main = ProofOfBurn(16, 4, 8, 50, 31, 2, 10 ** 19);
```

However, Ethereum's Merkle-Patricia-Trie can theoretically have a maximum depth of 64 nodes (one per nibble of the 32-byte address hash). Extension nodes provide opportunistic compression but are not guaranteed by the protocol—they only form when consecutive nibbles happen to share paths in the random hash distribution. As Ethereum's state grows denser over time, extension nodes will become less common and proof depths will naturally increase toward the theoretical maximum. The circuit assumes at least 75% compression (64→16) will always occur, which has no protocol guarantee.

Impact

Medium. As Ethereum's state tree grows, naturally occurring address hashes will create deeper proof paths exceeding 16 nodes. A malicious actor could also mine address hashes designed to create pathological MPT paths with minimal extension node compression, then burn legitimate ETH to such an address. The account proof will exceed 16 nodes, and the circuit cannot prove this valid burn because `numLayers > maxNumLayers`. This creates a denial-of-service where valid burns become unprovable and user funds are effectively locked—they burned ETH but cannot mint the corresponding tokens.

Recommendation

Increase `maxNumLayers` to account for worst-case scenarios. Consider increasing from 16 to 32 layers, which provides 2x margin over the current maximum while remaining well below the theoretical maximum of 64. This balances security against circuit complexity and accounts for future state tree growth. Alternatively, implement dynamic layer support or document the 16-layer limitation prominently so users understand the risk.

Developer Response

I think that the average number of layers is going to be `log16(num unique eth addresses)`, which right now is `log16(350000000)~7.09` So the trie depth will reach 16 when the number of unique eth addresses reach `16^16 = 1.8446744e+19` which is WAY WAY more than the current numbers.
32 layers will double the size of zkey parameters and I believe reaching 16 layers is already going to take hundreds of years (If not thousands/millions/...?). So, wont fix.

2.9.3 Burn Address Front-Running Causes Proof Failure and Fund Loss

Technical Details

The MPT leaf node encoding includes the balance field:

```
signal (leaf[maxLeafLen], leafLen) <== RlpMerklePatriciaTrieLeaf(32, amountBytes)(
    addressHashNibbles, numLeafAddressNibbles, balance  // Balance is part of leaf encoding
);
```

Any change to the balance changes the entire leaf hash, making it a different leaf in the Merkle tree. An attacker can monitor Ethereum for ETH transfers to newly created addresses, identify burn transactions, and front-run by sending any amount of ETH to the same address before the next block. This causes the victim's Merkle proof to fail by changing the leaf encoding. In Scenario A (`dust ≤ 10 ETH`), the victim must regenerate their proof with the updated balance, facing time pressure from the 51-minute blockhash expiry window. In Scenario B (dust > 10 ETH cap), the circuit constraint `balance ≤ 10 ETH` fails, making proof generation impossible and permanently locking the victim's funds.

Impact

Medium. Low likelihood as the attacker must actively monitor transfers and burn ETH, but high severity potential. In Scenario A, victims lose the attacker's dust amount and face proof regeneration delays. In Scenario B, victims suffer permanent loss of all burned ETH when the attacker sends enough dust to exceed the 10 ETH cap. Attack cost ranges from 0.001 ETH to 1+ ETH (burned forever), while victim loss ranges from minor inconvenience to complete fund loss.

Recommendation

Implement balance clamping in the circuit to prevent permanent loss. Modify the circuit to use `min(balance, maxBalance)` in the commitment computation, allowing users to recover up to the cap even when dust pushes the balance above it. Additionally, document that users should wait for one block confirmation and verify their balance matches the sent amount before generating proofs to detect and avoid dust attacks.

Developer Response

Fixed at PR#14

2.10 Low Findings

2.10.1 User must remember remaining balance to spend coins

Technical Details

The `remainingCoin` commitment is computed as `Poseidon(burnKey, intendedBalance - revealAmount)` in the circuit, but this remaining balance value is not stored anywhere on-chain. Users must manually track and remember the exact remaining balance for each coin to spend it later.

```
signal remainingCoin <== Poseidon(3)([
    POSEIDON_COIN_PREFIX(),
    burnKey,
    intendedBalance - revealAmount  // This value is NOT stored on-chain
]);
```

proof*of*burn.circom#L113

```
coins[_remainingCoin] = _remainingCoin; // Only stores the hash
revealed[_remainingCoin] = _proverFee + _broadcasterFee + _revealedAmount;
```

BETH.sol#L79-L80

What's stored on-chain:

- `_remainingCoin` - the Poseidon hash (commitment)
- `revealed[_remainingCoin]` - total amount minted so far

What's NOT stored:

- `intendedBalance - revealAmount` - the actual remaining balance
- `burnKey` - the secret key

What user needs to spend the coin: Both `burnKey` AND `intendedBalance - revealAmount` must be known to recompute the correct `remainingCoin` commitment.

2.10.2 Impact

Low - Usability issue that can lead to permanent fund loss through user error, but not a protocol vulnerability.

Recommendation

The current commitment method provides flexibility for fee handling in the contract. When adding new fee types beyond `_proverFee + _broadcasterFee`, the circuit does not need to be updated since fees are handled entirely at the contract level.
Document clearly in user-facing documentation that when calling `BETH.spendCoin`, the `_coin` parameter must be calculated as:

```
signal remainingCoin <== Poseidon(3)([POSEIDON_COIN_PREFIX(), burnKey,
intendedBalance - revealAmount]);
```

proof*of*burn.circom#L113

Developer Response

A user needs to remember the remaining balance. Our current Rust client does save the coin info in a file

2.10.3 Balance collision allows forging Merkle proof layers via SubstringCheck bypass

Technical Details

The `ProofOfBurn` circuit verifies Merkle-Patricia Trie structure by checking that each layer's Keccak hash appears as a substring in its parent layer:

```
if(i > 0) {
    substringCheckers[i - 1] <== SubstringCheck(maxNodeBlocks * 136, 31)(
        subInput <== reducedLayerKeccaks[i],      // 31 bytes of keccak(child layer)
        mainLen <== layerLens[i - 1],
        mainInput <== layers[i - 1]                // parent layer bytes
    );
}
```

The circuit does not distinguish between a child node hash appearing in the parent node (legitimate) versus account balance bytes appearing in a parent layer (attack). An attacker can exploit this by:

1. Crafting a `fake_child` node they want to inject
2. Computing `target_hash = keccak(fake_child)[0:31]`
3. Sending ETH to an uncle address (sibling of the real parent in the MPT) to set its balance = `target_hash`
4. Submitting a proof using the uncle's leaf node as the fake parent layer
5. The `SubstringCheck` finds the balance bytes in the uncle's leaf and incorrectly validates it

The uncle's leaf contains the target hash in its balance field, which the circuit cannot distinguish from a legitimate hash reference pointing to a child node.

Impact

Low. While the vulnerability exists in principle, it is not practically exploitable with current parameters.
The protocol sets `maxBalance = 10 ETH`, which effectively mitigates this attack. For a random `fake_child`, the required balance is approximately 2^248 wei. To satisfy the 10 ETH constraint, the attacker must grind to find a `fake_child` where `keccak(fake_child)[0:31] ≤ 10^19 wei`, requiring at least 184 leading zero bits. This requires 2^184 hash attempts, which exceeds SHA-256 collision resistance and is computationally infeasible with current technology.

Recommendation

The current `maxBalance = 10 ETH` parameter provides adequate protection. For defense-in-depth, document the security assumption that `maxBalance` must remain less than 2^80 to prevent balance collision attacks. For a long-term structural fix, modify the MPT verification to distinguish between hash references and balance fields by adding explicit node type checking.

Developer Response

Fixed at commit ebc72a7. During the audit, the WORM team raised and emphasised on investigating this issue.

2.10.4 Missing Block Number Validation Causes Valid Proof Rejection

Technical Details

The contract does not validate that `_blockNumber` is within the valid range before calling `blockhash()`:

```
bytes32 blockRoot = blockhash(_blockNumber);
uint256 commitment = uint256(
    keccak256(
        abi.encodePacked(
            blockRoot,
            _nullifier,
            _remainingCoin,
            // ...
        )
    )
) >> 8;
require(proofOfBurnVerifier.verifyProof(_pA, _pB, _pC, [commitment]), "Invalid proof!");
```

If the current block is `N = block.number`, then `blockhash(N)` returns `0x0` (cannot get current block hash), `blockhash(N-1)` through `blockhash(N-256)` return valid blockhashes, and `blockhash(N-257)` or older returns `0x0`. When `_blockNumber` is older than 256 blocks or equals the current block, `blockhash(_blockNumber)` returns `0x0`, which becomes part of the commitment used for proof verification.

Impact

Low. While `blockhash()` returns `0x0` for invalid block numbers, the attacker cannot exploit this to mint unbacked tokens. Creating a valid MPT proof that results in a state root of `0x00...00` requires finding a preimage `x` such that `keccak256(x) = 0x00...00`, which is computationally infeasible (requires 2^256 hash attempts). The actual impact is that valid proofs will be rejected if submitted after the 256-block window (~51 minutes), causing user inconvenience and potential fund loss if users miss the submission deadline.

Recommendation

Add validation to ensure `_blockNumber` is within the valid range and reject proofs that use invalid block numbers:

```
require(_blockNumber < block.number && _blockNumber >= block.number - 256, "Invalid block number");
bytes32 blockRoot = blockhash(_blockNumber);
require(blockRoot != bytes32(0), "Block root unavailable");
```

This provides clear error messages when proofs are submitted outside the valid time window.

Developer Response

Fixed at commit a91a64a

2.11 Gas Savings Findings

2.11.1 Optimize SubstringCheck Constraint Usage with Sliding Window Approach

Technical Details

The `SubstringCheck` template in `substring_check.circom` uses a cumulative accumulation approach that creates more constraints than necessary:

```
for (var i = 0; i < maxMainLen; i++) {
    M[i + 1] <== mainInput[i] * (256 ** i) + M[i];
}
```

When `maxMainLen = maxNodeBlocks * 136 = 4 * 136 = 544`, the calculation involves `256^543`, which vastly exceeds the BN254 scalar field modulus ($\sim 2^{254}$), causing field arithmetic overflow. Despite the overflow, the substring check remains cryptographically sound due to the following constraints:

1. **Range constraints:** `AssertByteString` enforces `0 ≤ mainInput[i], subInput[i] < 256`
2. **Length limit:** `subLen ≤ 31` ensures `subInputNum < 256^31 < 2^248 < p`
3. **Valid field division:** The equality check `(256^i) * subInputNum ≡ M[i + subLen] - M[i] (mod p)` can be divided by `256^i mod p` (which is never zero) to yield `subInputNum ≡ (byte window representation) (mod p)`

Impact

Gas Optimization - The original implementation is secure but SubstringCheck2 achieves identical functionality with better performance. The performance was measured via the command `circom -l circuits .tmp_circuits/substringcheck544_31.circom`.

- Original `SubstringCheck`: **3734 linear constraints**, 11728 wires
- Optimized `SubstringCheck2`: **3190 linear constraints**, 11184 wires
- Improvement: **544 fewer constraints (14.5% reduction), 544 fewer wires (4.6% reduction)**

Recommendation

Replace the accumulation approach with a sliding window implementation that limits exponents to `subLen - 1` (maximum 30). Use `W[i+1] = (W[i] - mainInput[i] * 256^(subLen-1)) * 256 + mainInput[i+subLen]` to compute windows of exactly `subLen` bytes, ensuring all intermediate values remain under 2^248 and avoiding field overflow.

```
pragma circom 2.2.2;

include "../circomlib/circuits/comparators.circom";
include "./convert.circom";
include "./assert.circom";

// Alternative substring check using sliding window approach to avoid overflow issues.
//
// The original implementation accumulates: M[i] = mainInput[0] + mainInput[1]*256 + ... + mainInput[i-1]*256^(i-1)
// This causes overflow when i is large (maxMainLen = 544 means 256^544 >> 2^254).
//
// This implementation uses sliding windows of exactly subLen bytes:
//   W[i] = mainInput[i] + mainInput[i+1]*256 + ... + mainInput[i+subLen-1]*256^(subLen-1)
//
// Since subLen <= 31:
//   - Each window: W[i] < 256^31 < 2^248 < 2^254 (no overflow!)
//   - Each element: mainInput[i] * 256^j where j < 31, so mainInput[i] * 256^30 < 256 * 2^240 = 2^248 < 2^254
//
// Example:
//   mainInput: [1, 2, 3, 4, 5]
//   subLen:    3
//   W[0] = 1 + 2*256 + 3*256^2
//   W[1] = 2 + 3*256 + 4*256^2
//   W[2] = 3 + 4*256 + 5*256^2
//
template SubstringCheck2(maxMainLen, subLen) {
    signal input mainInput[maxMainLen];
    signal input mainLen;
    signal input subInput[subLen];
    signal output out;

    assert(subLen <= 31); // So that subInput fits in a field element without overflow

    // Substring-checker works with byte-string inputs
    AssertByteString(subLen)(subInput);
    AssertByteString(maxMainLen)(mainInput);

    AssertLessEqThan(16)(mainLen, maxMainLen);
    AssertLessEqThan(16)(subLen, mainLen);

    // Convert the sub-input into a field-element (BIG-endian for cleaner sliding window)
    signal subInputNum <== BigEndianBytes2Num(subLen)(subInput);
```

```
    // W[i] = Big-endian number representation of mainInput[i..i+subLen)
    // W[i] = mainInput[i]*256^(subLen-1) + mainInput[i+1]*256^(subLen-2) + ... + mainInput[i+subLen-1]*256^0
    //
    // Example with subLen = 3:
    //   mainInput: [1, 2, 3, 4, 5]
    //   W[0] = 1*256^2 + 2*256^1 + 3*256^0
    //   W[1] = 2*256^2 + 3*256^1 + 4*256^0
    //   W[2] = 3*256^2 + 4*256^1 + 5*256^0
    //
    // Key insight: Since subLen <= 31, we have:
    //   W[i] < 256^31 = 2^248 < 2^254 (fits in field element!)
    //
    // Efficient sliding window recurrence:
    //   W[i+1] = (W[i] - mainInput[i] * 256^(subLen-1)) * 256 + mainInput[i+subLen]
    //
    // Derivation:
    //   W[i]   = mainInput[i]*256^(subLen-1)   + mainInput[i+1]*256^(subLen-2) + ... + mainInput[i+subLen-1]
    //   W[i+1] = mainInput[i+1]*256^(subLen-1) + mainInput[i+2]*256^(subLen-2) + ... + mainInput[i+subLen]
    //
    //   Remove leftmost byte:  W[i] - mainInput[i]*256^(subLen-1)
    //                          = mainInput[i+1]*256^(subLen-2) + ... + mainInput[i+subLen-1]
    //   Shift left by 256:    (W[i] - mainInput[i]*256^(subLen-1)) * 256
    //                          = mainInput[i+1]*256^(subLen-1) + ... + mainInput[i+subLen-1]*256
    //   Add new byte:          + mainInput[i+subLen]
    //                          = mainInput[i+1]*256^(subLen-1) + ... + mainInput[i+subLen]
    //                          = W[i+1]
    //
    // This requires only 2 constraints per window (1 subtraction, 1 multiply-add)!
    //
    signal W[maxMainLen - subLen + 1];

    // Compute first window directly (big-endian)
    var firstWindowSum = 0;
    for (var j = 0; j < subLen; j++) {
        firstWindowSum += mainInput[j] * (256 ** (subLen - 1 - j));
    }
    W[0] <== firstWindowSum;

    // For subsequent windows, use efficient recurrence relation
    signal diff[maxMainLen - subLen];

    for (var i = 0; i < maxMainLen - subLen; i++) {
        // Step 1: Remove the leftmost byte (which has weight 256^(subLen-1))
        diff[i] <== W[i] - mainInput[i] * (256 ** (subLen - 1));

        // Step 2: Shift left by 256 and add new rightmost byte
        W[i + 1] <== diff[i] * 256 + mainInput[i + subLen];
    }
```

```
    // Substring-ness Equation: Substring exists if there is `i` where:
    // W[i] == subInputNum

    // Existence flags. When exists[i] is 1 it means that:
    // mainInput[i..i + subLen] == subInput
    signal exists[maxMainLen - subLen + 1];

    // Used for creating an `allowed` filter: [1, 1, ..., 1, 1, 0, 0, ..., 0, 0]
    // Where the first `mainLen - subLen + 1` elements are 1, indicating the existence
    // flags that should be considered.
    signal isLastIndex[maxMainLen - subLen + 1];
    signal allowed[maxMainLen - subLen + 2];
    allowed[0] <== 1;

    // For summing up all the *allowed* existence flags.
    signal sums[maxMainLen - subLen + 2];
    sums[0] <== 0;

    for (var i = 0; i < maxMainLen - subLen + 1; i++) {
        // Building the `allowed` filter
        isLastIndex[i] <== IsEqual()([i, mainLen - subLen + 1]);
        // prev index is allowed && not last index
        allowed[i + 1] <== allowed[i] * (1 - isLastIndex[i]);

        // Existence check: does W[i] equal subInputNum?
        exists[i] <== IsEqual()([W[i], subInputNum]);

        // Existence flag is accumulated in the sum only when we are in the allowed region
        sums[i + 1] <== sums[i] + allowed[i + 1] * exists[i];
    }

    // Substring exists only when there has been a 1 while summing up the existence flags
    signal doesNotExist <== IsZero()(sums[maxMainLen - subLen + 1]);
    out <== 1 - doesNotExist;
}
```

Developer Response

Won't fix since it's informational.

2.12 Informational Findings

2.12.1 Nullifier Stored On-Chain and Derived From Single Secret

Technical Details

The nullifier is computed as `Poseidon2(POSEIDON_NULLIFIER_PREFIX, burnKey)` where `burnKey` is the only source of entropy. It is stored in publicly on-chain.

```
mapping(uint256 => bool) public nullifiers;
```

BETH.sol#L13
This differs from Tornado Cash's design where the nullifier is computed from an independent random value.

```
signal nullifier <== Poseidon(2)([POSEIDON_NULLIFIER_PREFIX(), burnKey]);
```

proofofburn.circom#L118
Comparison with Tornado Cash:
Tornado Cash withdraw.circom#L18-L22:

```
// Two INDEPENDENT secrets
signal input nullifier;   // Random 248-bit value
signal input secret;      // Random 248-bit value

// commitment uses BOTH
commitment = PedersenHash(nullifier, secret)

// nullifierHash hash uses ONLY nullifier
nullifierHash = PedersenHash(nullifier)
```

BETH (current):

```
// One secret
signal input burnKey;  // Constrained by PoW

// Nullifier uses ONLY burnKey
nullifier = Poseidon2(PREFIX, burnKey)

// No independent nullifier secret
```

The difference:

- Tornado Cash: `nullifier` is an independent 248-bit random value, separate from `secret`
- BETH: `nullifier` is deterministically derived from `burnKey` (which is PoW-constrained)

Impact

Info - The nullifier's security is limited by the PoW difficulty, which filters the valid burnKey space.
PoW Effect on Valid BurnKey Space:
With `minimumZeroBytes = 2`:

```
Total burnKey space: 2^254
PoW Difficulty: 2^16 (only 1 in 65,536 burnKeys are valid)

Valid burnKeys in full field: 2^254 / 2^16 = 2^238

Rainbow table for full field:
  Must enumerate: 2^254 burnKeys
  Will find: 2^238 valid burnKeys
  Storage: 2^238 × 64 bytes ≈ 2.2e+72 bytes (impossible)
  Build time: 2^254 / (2×10^9 H/s) ≈ 4.6e+59 years (impossible)
```

Recommendation

Add Independent Nullifier Secret

```
template ProofOfBurn() {
    signal input burnKey;           // For burn address (PoW constrained)
    signal input nullifierSecret;   // For nullifier (unconstrained)

    // Nullifier uses independent secret
    signal nullifier <== Poseidon(3)([
        POSEIDON_NULLIFIER_PREFIX,
        burnKey,
        nullifierSecret  // Additional 254-bit entropy
    ]);

    // PoW still uses only burnKey (for burn address derivation)
    signal powHash <== Keccak(burnKey, revealAmount, burnExtraCommitment,
"EIP-7503");
}
```

Effect:

```
Rainbow table attack on nullifier:
  Search space: 2^254 (burnKey) × 2^254 (nullifierSecret) = 2^508
  Table size: Impossible

Security: 254-bit (independent of PoW difficulty)
```

Developer Response

Won't fix since it's informational

2.12.2 LeafDetector Accepts Invalid Compact Encoding Prefixes

Technical Details

The LeafDetector validates that `keyPrefix <= 0xb7` but does not explicitly reject invalid compact encoding values.

```
// circuits/utils/rlp/merkle_patricia_trie_leaf.circom:261
signal keyPrefixIsValid <== LessEqThan(16)([keyPrefix, 0xb7]);
```

https://github.com/worm-privacy/proof-of-burn/blob/ebc72a7b7cf509dc902140befb16e198568f8721/circuits/utils/rlp/merkle*patricia*trie_leaf.circom#L261
According to the Ethereum compact encoding specification, only 4 flag values are valid:

```
Flag 0 (0x00-0x0f): Extension, even nibbles
Flag 1 (0x10-0x1f): Extension, odd nibbles
Flag 2 (0x20):      Leaf, even nibbles (0 nibbles)
Flag 3 (0x30-0x3f): Leaf, odd nibbles (1 nibble)
```

For leaves, valid key prefixes are:

- `0x20` : Single-byte leaf, 0 nibbles
- `0x30-0x3f` : Single-byte leaf, 1 nibble
- `0x81-0xb7` : Multi-byte keys after RLP encoding (2-64 nibbles)

Invalid ranges currently accepted:

- `0x00-0x1f` : Extension nodes (not leaves)
- `0x40-0x80` : Invalid compact encoding flags (4-7 don't exist)

Impact

Informational - No security impact.
While the check accepts invalid compact encoding values, these cannot be easily exploitable.

Recommendation

Add explicit flag validation for specification compliance:

```
// Check if single-byte key with valid leaf flag (0x20 or 0x30-0x3f)
signal keyIs0x20 <== IsEqual()([keyPrefix, 0x20]);
signal keyIsInRange30to3f <== IsInRange(16)(0x30, keyPrefix, 0x3f);
signal keyIsSingleByteLeaf <== OR()(keyIs0x20, keyIsInRange30to3f);

// Check if multi-byte key (0x81-0xb7)
signal keyIsMultiByte <== IsInRange(16)(0x81, keyPrefix, 0xb7);

// Key must be either single-byte leaf OR multi-byte
signal keyPrefixIsValid <== OR()(keyIsSingleByteLeaf, keyIsMultiByte);
```

Developer Response

Acknowledged but won't fix. I think the ”not being able to generate proofs for some” is better than an exploit that allows someone to mint as much as he wants. The current implementation only accepts nodes with 2 rlp elements. Both branch and extension nodes do not obey this rule so we're fine.

2.12.3 Incorrect Comment About Filter Array Length

Technical Details

The comment on line 506 incorrectly states the number of elements set to 1 in the filter array:

```
// Create a 1, 1, ..., 1, 1, 0, 0, ..., 0, 0 filter
// Where the first `inLen` elements are 1  ← WRONG
signal filter[maxBytes + 1];
```

The filter construction logic creates indices 0 through `inLen` as 1, meaning `inLen + 1` elements total are set to 1, not `inLen` elements. The implementation is correct; only the comment is misleading.

Impact

Informational. Documentation error only with no functional impact.

Recommendation

Update the comment to reflect the correct count:
`// Where the first inLen + 1 elements are 1`.

Developer Response

Fixed at commit 634bd93

2.12.4 Incorrect comment about minimum account RLP length

Technical Details

The comment at line 110 states the minimum value RLP length is 71 bytes, but the actual minimum is 70 bytes when balance is zero. The minimum case consists of: nonce (1 byte as `0x80`), balance (1 byte as `0x80`), storage root (33 bytes as `0xa0` + 32-byte hash), and code hash (33 bytes as `0xa0` + 32-byte hash), totaling 68 bytes of payload. With RLP list encoding for payloads > 55 bytes using long format `[0xf8, 68]` (2 bytes), the total minimum is 70 bytes.

Impact

Informational. Documentation inaccuracy only. The implementation uses correct bounds; only the comment is incorrect.

Recommendation

Update the comment to reflect the correct minimum: `// Min length: 70`.

Developer Response

Fixed at commit 5338706

2.12.5 RLP Parsing Ambiguity for Single-Byte MPT Leaf Keys

Technical Details

The RLP parsing logic in `merkle_patricia_trie_leaf.circom` assumes MPT leaf keys are always prefixed with `0x80+keyLen`:

```
// Expected:  [0xf8] [length] [0x80+keyLen] [key_bytes...] [value_RLP...]
// Actual:    [0xf8] [length] [0x35] [value_RLP...]
```

When the key is a single byte (e.g., `0x35`), it has no `0x80+len` prefix according to RLP encoding rules. The code cannot distinguish whether `0x35` is the entire key or a length prefix, making parsing ambiguous. This violates the consistent `[0x80+len]` prefix assumption and causes incorrect parsing.

Impact

Informational. For a 1-byte MPT leaf key to occur, the burn address hash must match an existing account's hash such that only 1-2 nibbles remain in the MPT leaf after trie compression. This requires the first 62-63 nibbles (252-256 bits) to exactly match the path to an existing account, which is computationally infeasible as it requires breaking keccak256 collision resistance.

Recommendation

Add a minimum key length check: `AssertGreaterThan(8)(keyLen, 1)` to ensure keys are at least 2 bytes, or document that 1-byte keys are theoretically unsupported but practically impossible to encounter.

Developer Response

Will fix to make code prettier

2.12.6 Hardcoded stateRoot offset assumes fixed RLP prefix length

Technical Details

The circuit hardcodes the stateRoot position at byte 91:

```
var stateRootOffset = 91; // stateRoot starts from byte 91 of the block-header
signal stateRoot[32];
for(var i = 0; i < 32; i++) {
    stateRoot[i] <== blockHeader[stateRootOffset + i];
}
```

This assumes a 3-byte RLP list prefix `[0xf9, len_high, len_low]` for the block header. Ethereum's RLP encoding uses variable-length prefixes: 1-byte for payloads less than 55 bytes, 2-byte for 56-255 bytes, 3-byte for 256-65,535 bytes, and 4-byte for payloads greater than 65,536 bytes. If a future block header exceeds 65,535 bytes, it would require a 4-byte prefix `[0xfa, 0x00, len_high, len_low]`, shifting the stateRoot to byte 92 instead of 91.

Impact

Informational. Current mainnet headers are ~842 bytes with maximum circuit support of 1,088 bytes, both well below the 65,536-byte threshold. Triggering a 4-byte prefix would require headers 60× larger than current maximums. If this occurs, proofs would fail rather than accept invalid state roots, making this a fail-safe condition.

Recommendation

Add an assertion to enforce the 3-byte prefix assumption: `blockHeader[0] === 0xf9`, or document the limitation that the circuit assumes block headers < 65,536 bytes and will require updates if future hard forks exceed this threshold.

Developer Response

The circuit already has a maximum limit on header size. So this won't change anything. Extremely unlikely. Won't fix.

2.12.7 Unused variables in Template D

Technical Details

In template `D`, variables `a64` and `b64` are created via `Fit(n, 64)` but never used:

```
signal a64[64] <== Fit(n, 64)(a);
signal b64[64] <== Fit(n, 64)(b);
signal aux0[64] <== ShR(64, shr)(a);     // Uses 'a' instead of 'a64'
signal aux1[64] <== ShL(64, shl)(a);     // Uses 'a' instead of 'a64'
signal aux2[64] <== OrArray(64)(aux0, aux1);
out <== XorArray(64)(b, aux2);           // Uses 'b' instead of 'b64'
```

The template is only called as `D(64, 1, 64-1)`, making `Fit(64, 64)` a no-op that copies the array without adding constraints.

Impact

Informational. No functional impact since when `n=64`, using `a` versus `a64` produces identical results. The `Fit()` calls are dead code.

Recommendation

Remove the unused variables for code clarity, or use the fitted variables consistently if they were intended to be used.

Developer Response

Fixed at commit 5a92e35

2.12.8 Incorrect variable in assertion check for maxKeyRlpLen

Technical Details

In `RlpMerklePatriciaTrieLeaf`, line 146 checks the wrong variable:

```
var maxKeyLen = 1 + maxAddressHashBytes;
assert(maxKeyLen <= 33);              // Line 138 - Correct
var maxKeyRlpLen = 1 + maxKeyLen;     // Line 144
assert(maxKeyLen <= 34);              // Line 146 - Should be maxKeyRlpLen
```

The assertion checks `maxKeyLen` twice instead of checking `maxKeyRlpLen`.

Impact

Informational. No functional impact since `maxKeyRlpLen = 1 + maxKeyLen` and the constraint `maxKeyRlpLen <= 34` is mathematically implied by `maxKeyLen <= 33`.

Recommendation

Fix the assertion to check the intended variable: `assert(maxKeyRlpLen <= 34)`, or remove the redundant assertion since it's implied by the previous constraint.

Developer Response

Fixed at commit fc70f40

2.13 Final Remarks

The WORM protocol demonstrates solid cryptographic foundations for privacy-preserving proof-of-burn. One high-severity issue remains: coin lineage tracking that compromises BETH circulation privacy. The team should prioritize: (1) expanding test coverage, (2) validating security-critical parameters (balance caps, MPT depth limits), (3) reconsidering the privacy architecture, (4) adding defensive mechanisms like emergency pause functionality, and (5) monitoring MPT depth as Ethereum's state tree grows. The protocol relies on reasonable but unvalidated assumptions - making these explicit and enforceable would strengthen robustness.