BLOCKSEC

Security Audit

Report for atoshi-privacy-contracts

Date: July 31, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Atoshi
Target	atoshi-privacy-contracts

Version History

Version	Date	Description
1.0	July 31, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-17 03:29:13+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of atoshi-privacy-contracts of Atoshi.

Atoshi Privacy Contracts is a Solidity smart contract system for shielded transactions on Atoshi Chain. The core contract, `Shield`, accepts native-token and supported ERC-20 deposits, records commitments in an incremental Merkle tree, and verifies Groth16 proofs for withdrawals and private transfers. The system also includes `EnergySettlement`, an optional module that tracks relayer gas quotas, and `TokenRegistry`, which manages supported-token metadata and accounting. Correctness depends on the on-chain verifier contracts matching the audited privacy circuits and on the off-chain SDK correctly creating and handling encrypted notes.

Note this audit only focuses on the smart contracts in the following directories/files:

- `contracts/core/EnergySettlement.sol`
- `contracts/core/Shield.sol`
- `contracts/libraries/MerkleTree.sol`
- `contracts/verifiers/ShieldVerifier.sol`
- `contracts/verifiers/TransferVerifier.sol`
- `contracts/verifiers/UnshieldVerifier.sol`
- `contracts/tokens/TokenRegistry.sol`

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
atoshi-privacy-contracts	`Version 1`	`abd68f461b33092600f2d984b30b531b7b3be16d`
	`Version 2`	`e0deb2330935c56dd48decd19c643f0511572835`

[1] https://github.com/ATOSHI-ORG/atoshi-privacy-contracts

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation

- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **eight** potential security issues. Besides, we have **two** recommendations and **three** notes.

- High: 3
- Medium: 5
- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	High	Potential DoS due to mismatched verifier contracts	Security Issue	Partially Fixed
2	High	Lack of validation on `token` and `amount` in function `deposit()`	Security Issue	Fixed
3	High	Lack of relayer binding in function `withdraw()`	Security Issue	Fixed
4	Medium	Lack of access control in functions `recordDeposit()` and `recordWithdrawal()`	Security Issue	Fixed
5	Medium	Incorrect fee handling in function `withdraw()`	Security Issue	Fixed
6	Medium	Lack of withdrawal path for removed tokens	Security Issue	Fixed
7	Medium	Insufficient Merkle tree capacity	Security Issue	Fixed
8	Medium	Potential privacy leakage	Security Issue	Fixed
9	-	Remove redundant code	Recommendation	Fixed
10	-	Revise the misleading annotation	Recommendation	Fixed
11	-	Assumptions of `_encryptedNote`	Note	-
12	-	Shared Merkle tree across assets	Note	-
13	-	Potential centralized risk	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential DoS due to mismatched verifier contracts

Severity High

Status Partially Fixed in `Version 2`

Introduced by `Version 1`

Description Verifier contracts `ShieldVerifier`, `TransferVerifier`, and `UnshieldVerifier` should be generated from the `atoshi-privacy-circuits` [1] circuits. Contract `Shield` relies on these contracts to verify zero-knowledge proofs in functions `transfer()` and `withdraw()`. However, verifier contracts do not match the actual circuit. As a result, the contract may be unable to verify legitimate proofs, leading to a DoS in which deposited funds cannot be withdrawn.

[1] https://github.com/ATOSHI-ORG/atoshi-privacy-circuits

```
24    uint256 constant r =
          21888242871839275222246405745257275088548364400416034343698204186575808495617;
```

Listing 2.1: contracts/verifiers/TransferVerifier.sol

```
207   function withdraw(
208       uint256[2] calldata _pA,
209       uint256[2][2] calldata _pB,
210       uint256[2] calldata _pC,
211       uint256 _root,
212       uint256 _nullifierHash,
213       address _recipient,
214       address _relayer,
215       uint256 _fee,
216       address _token,
217       uint256 _amount
218   ) external override nonReentrant whenNotPaused validToken(_token) {
219       require(_nullifierHash < FIELD_SIZE, "Shield: invalid nullifier");
220       require(!nullifierHashes[_nullifierHash], "Shield: already spent");
221       require(isKnownRoot(_root), "Shield: unknown root");
222       require(_recipient != address(0), "Shield: invalid recipient");
223       require(_fee <= _amount, "Shield: fee exceeds amount");
224
225       // Verify ZK proof against the unshield circuit verifier.
226       // Public inputs (must match circuits/unshield.circom output order):
227       //   [0] root
228       //   [1] nullifierHash
229       //   [2] recipient (address as uint256)
230       //   [3] tokenId / token address as uint256
231       //   [4] amount
232       //   [5] fee
233       uint256[6] memory pubSignals = [
234           _root,
235           _nullifierHash,
236           addressToUint256(_recipient),
237           addressToUint256(_token),
238           _amount,
239           _fee
240       ];
241
242       require(
243           IUnshieldVerifier(unshieldVerifier).verifyProof(_pA, _pB, _pC, pubSignals),
244           "Shield: invalid proof"
245       );
```

Listing 2.2: contracts/core/Shield.sol

```
299   function transfer(
300       uint256[2] calldata _pA,
301       uint256[2][2] calldata _pB,
302       uint256[2] calldata _pC,
303       uint256 _root,
304       uint256 _nullifierHash,
```

```
305       uint256 _newCommitment,
306       bytes calldata _encryptedNote
307   ) external nonReentrant whenNotPaused {
308       require(_nullifierHash < FIELD_SIZE, "Shield: invalid nullifier");
309       require(_newCommitment < FIELD_SIZE, "Shield: invalid commitment");
310       require(!nullifierHashes[_nullifierHash], "Shield: already spent");
311       require(isKnownRoot(_root), "Shield: unknown root");
312
313       // Verify ZK proof against the transfer circuit verifier.
314       // Public inputs (transfer.circom output order):
315       //   [0] root
316       //   [1] nullifierHash (of the input note)
317       //   [2] newCommitment (of the output note)
318       uint256[3] memory pubSignals = [_root, _nullifierHash, _newCommitment];
319
320       require(
321           ITransferVerifier(transferVerifier).verifyProof(_pA, _pB, _pC, pubSignals),
322           "Shield: invalid proof"
323       );
```

Listing 2.3: contracts/core/Shield.sol

Impact The verifier contracts do not correspond to the audited circuits, preventing valid proofs from being verified.

Suggestion Fix the circuits in `atoshi-privacy-circuits` and generate the verifier contracts based on the circuits.

Feedback from the project The project will conduct an open community relay ceremony with publicly verifiable contribution logs before the mainnet launch. Under Groth16 Phase 2's 1-of-N trust assumption, the resulting keys will remain secure as long as at least one participant contributes honestly. Following the ceremony, the project will regenerate and redeploy the `Shield` and verifier contracts, update `DEPLOYMENT_HASHES.md` with the new addresses and hashes, perform end-to-end verification using `audit-verify.sh`, and publish a follow-up containing the results and contribution log.

Clarification from BlockSec This issue can be considered fully fixed only after the production trusted setup is completed, and the resulting artifacts, deployments, and verification records have been reviewed.

2.1.2 Lack of validation on `token` and `amount` in function `deposit()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Shield`, function `deposit()` accepts the user-supplied parameter `_commitment`, which is expected to bind the deposited `_token` and `_amount`. However, the function does not verify whether this `_commitment` is derived from the current `_token` and `_amount`. As a result, a user can deposit a small amount while submitting a commitment that encodes a larger

amount or a different token. This allows the user to withdraw more assets than actually deposited, causing losses to the pool.

```
171   function deposit(
172       uint256 _commitment,
173       address _token,
174       uint256 _amount,
175       bytes calldata _encryptedNote
176   ) external payable override nonReentrant whenNotPaused validToken(_token) {
177       require(_commitment < FIELD_SIZE, "Shield: invalid commitment");
178       require(_amount >= minDeposits[_token], "Shield: amount too small");
179
180       // Handle token transfer
181       if (_token == NATIVE_TOKEN) {
182           require(msg.value == _amount, "Shield: incorrect native amount");
183       } else {
184           require(msg.value == 0, "Shield: unexpected native token");
185           IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
186       }
187
188       // Insert commitment into Merkle tree
189       uint32 leafIndex = commitmentTree.insert(_commitment);
190
191       emit Deposit(_commitment, leafIndex, block.timestamp, _token, _amount, _encryptedNote);
192   }
```

Listing 2.4: contracts/core/Shield.sol

Impact A user can withdraw more assets than actually deposited, draining value from the pool.

Suggestion Add a zero-knowledge proof to verify that the `_commitment` is correctly derived from the deposited `_token` and `_amount`.

2.1.3 Lack of relayer binding in function `withdraw()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Shield`, function `withdraw()` should pay the withdrawal fee to the relayer authorized by the proof. However, the parameter `_relayer` is a caller-supplied parameter that isn't bound to the proof, and relayer validation only runs when `msg.sender == _relayer`. As a result, attackers can front-run a legitimate relayer transaction, submit the same proof with themselves as `_relayer`, skip validation, and steal the fee.

```
255       if (energySettlement != address(0) && msg.sender == _relayer) {
256           IEnergySettlement(energySettlement).consumeForRelayer(_relayer, withdrawGasCharge);
257       }
```

Listing 2.5: contracts/core/Shield.sol

```
263       // Transfer tokens
264       if (_token == NATIVE_TOKEN) {
265           // Transfer to recipient
266           (bool success1, ) = payable(_recipient).call{value: netAmount}("");
267           require(success1, "Shield: native transfer failed");
268
269           // Transfer relayer fee
270           if (_fee > 0 && _relayer != address(0)) {
271               (bool success2, ) = payable(_relayer).call{value: _fee}("");
272               require(success2, "Shield: relayer fee transfer failed");
273           }
274
275           // Transfer protocol fee
276           if (protocolFee > 0 && feeRecipient != address(0)) {
277               (bool success3, ) = payable(feeRecipient).call{value: protocolFee}("");
278               require(success3, "Shield: protocol fee transfer failed");
279           }
280       } else {
281           IERC20(_token).safeTransfer(_recipient, netAmount);
282
283           if (_fee > 0 && _relayer != address(0)) {
284               IERC20(_token).safeTransfer(_relayer, _fee);
285           }
286
287           if (protocolFee > 0 && feeRecipient != address(0)) {
288               IERC20(_token).safeTransfer(feeRecipient, protocolFee);
289           }
290       }
```

Listing 2.6: contracts/core/Shield.sol

Impact An attacker can front-run a legitimate relayer and steal the relayer fee.

Suggestion Bind the `relayer` in the proof, and update function `withdraw()` in contract `Shield` to require `_relayer` to match the proof-bound value before paying the fee.

2.1.4 Lack of access control in functions `recordDeposit()` and `recordWithdrawal()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `TokenRegistry`, functions `recordDeposit()` and `recordWithdrawal()` record the cumulative deposited and withdrawn amounts per token (i.e., `totalDeposited` and `totalWithdrawn`). These functions should be invoked only by contract `Shield`. However, both are declared external with no access control, and neither function checks the configured `nabled`, `maxDeposit`, and `minDeposit` values. As a result, any address can invoke these functions with arbitrary `_tokenId` and `_amount` values, corrupting the values of `totalDeposited` and `totalWithdrawn`.

```
240   function recordDeposit(uint256 _tokenId, uint256 _amount) external {
241       // In production, add access control (only Shield can call)
```

```
242         tokens[_tokenId].totalDeposited += _amount;
243     }
244
245     /**
246      * @notice Record a withdrawal (called by Shield contract)
247      * @param _tokenId Token ID
248      * @param _amount Amount withdrawn
249      */
250     function recordWithdrawal(uint256 _tokenId, uint256 _amount) external {
251         // In production, add access control (only Shield can call)
252         tokens[_tokenId].totalWithdrawn += _amount;
253     }
```

Listing 2.7: contracts/tokens/TokenRegistry.sol

Impact Any address can corrupt the `totalDeposited` and `totalWithdrawn` accounting.

Suggestion Restrict both functions to be callable only by the contract `Shield`.

2.1.5 Incorrect fee handling in function `withdraw()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Shield`, function `withdraw()` deducts the relayer fee and protocol fee from the recipient's withdrawal amount regardless of whether the corresponding fee recipients are configured. The relayer fee is transferred only when `_relayer` is not the zero address, while the protocol fee is transferred only when `feeRecipient` is not the zero address. If either address is unset while its fee is non-zero, the fee is still deducted from the user but is not transferred to any recipient. As a result, the funds remain in the contract without a designated owner or recovery mechanism.

```
261         uint256 netAmount = _amount - _fee - protocolFee;
262
263         // Transfer tokens
264         if (_token == NATIVE_TOKEN) {
265             // Transfer to recipient
266             (bool success1, ) = payable(_recipient).call{value: netAmount}("");
267             require(success1, "Shield: native transfer failed");
268
269             // Transfer relayer fee
270             if (_fee > 0 && _relayer != address(0)) {
271                 (bool success2, ) = payable(_relayer).call{value: _fee}("");
272                 require(success2, "Shield: relayer fee transfer failed");
273             }
274
275             // Transfer protocol fee
276             if (protocolFee > 0 && feeRecipient != address(0)) {
277                 (bool success3, ) = payable(feeRecipient).call{value: protocolFee}("");
278                 require(success3, "Shield: protocol fee transfer failed");
279             }
```

```
280         } else {
281             IERC20(_token).safeTransfer(_recipient, netAmount);
282
283             if (_fee > 0 && _relayer != address(0)) {
284                 IERC20(_token).safeTransfer(_relayer, _fee);
285             }
286
287             if (protocolFee > 0 && feeRecipient != address(0)) {
288                 IERC20(_token).safeTransfer(feeRecipient, protocolFee);
289             }
290         }
291
292         emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
```

Listing 2.8: contracts/core/Shield.sol

Impact The unsent `_fee` or `protocolFee` is deducted from the user but permanently locked in the contract.

Suggestion Require `feeRecipient` to be non-zero in the constructor and configuration functions, and require `_relayer` to be non-zero in the function `withdraw()` when `fee` is non-zero.

2.1.6 Lack of withdrawal path for removed tokens

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Shield`, function `removeSupportedToken()` disables a token for both deposits and withdrawals. However, the contract cannot verify that all notes backed by the token have been spent before disabling it. Consequently, users with existing notes cannot withdraw their tokens while the token remains disabled. The tokens remain locked in the contract unless the owner re-enables the token.

```
110     modifier validToken(address _token) {
111         require(supportedTokens[_token], "Shield: unsupported token");
112         _;
113     }
```

Listing 2.9: contracts/core/Shield.sol

```
207     function withdraw(
208         uint256[2] calldata _pA,
209         uint256[2][2] calldata _pB,
210         uint256[2] calldata _pC,
211         uint256 _root,
212         uint256 _nullifierHash,
213         address _recipient,
214         address _relayer,
215         uint256 _fee,
216         address _token,
217         uint256 _amount
```

```
218   ) external override nonReentrant whenNotPaused validToken(_token) {
```

Listing 2.10: contracts/core/Shield.sol

```
391   function removeSupportedToken(address _token) external onlyOwner {
392       supportedTokens[_token] = false;
393   }
```

Listing 2.11: contracts/core/Shield.sol

Impact Existing notes for a removed token can not be withdrawn.

Suggestion Allow existing notes to be withdrawn when a token is disabled.

2.1.7 Insufficient Merkle tree capacity

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Contract `Shield` uses a single, shared, and append-only Merkle tree for all commitments, with capacity for at most 2^{20} leaves. Functions `deposit()` and `transfer()` insert a new leaf into this tree per invocation. Since these functions charge no fee, an attacker can fill the entire tree cheaply. Once full, no further transfers are possible, so undrawn notes in the old tree can no longer move, blocking private transfers for legitimate users.

```
48    uint32 public constant TREE_LEVELS = 20; // Supports 2^20 = ~1M deposits
```

Listing 2.12: contracts/core/Shield.sol

```
82    function insert(
83        TreeData storage self,
84        uint256 _leaf
85    ) internal returns (uint32 index) {
86        uint32 _nextIndex = self.nextIndex;
87        require(_nextIndex < uint32(2) ** self.levels, "Merkle tree is full");
88
89        uint32 currentIndex = _nextIndex;
90        uint256 currentLevelHash = _leaf;
91        uint256 left;
92        uint256 right;
```

Listing 2.13: contracts/libraries/MerkleTree.sol

```
299   function transfer(
300       uint256[2] calldata _pA,
301       uint256[2][2] calldata _pB,
302       uint256[2] calldata _pC,
303       uint256 _root,
304       uint256 _nullifierHash,
305       uint256 _newCommitment,
306       bytes calldata _encryptedNote
307   ) external nonReentrant whenNotPaused {
```

```
308     require(_nullifierHash < FIELD_SIZE, "Shield: invalid nullifier");
309     require(_newCommitment < FIELD_SIZE, "Shield: invalid commitment");
310     require(!nullifierHashes[_nullifierHash], "Shield: already spent");
311     require(isKnownRoot(_root), "Shield: unknown root");
312
313     // Verify ZK proof against the transfer circuit verifier.
314     // Public inputs (transfer.circom output order):
315     //   [0] root
316     //   [1] nullifierHash (of the input note)
317     //   [2] newCommitment (of the output note)
318     uint256[3] memory pubSignals = [_root, _nullifierHash, _newCommitment];
319
320     require(
321         ITransferVerifier(transferVerifier).verifyProof(_pA, _pB, _pC, pubSignals),
322         "Shield: invalid proof"
323     );
324
325     // Mark old nullifier as spent
326     nullifierHashes[_nullifierHash] = true;
327
328     // If a relayer broadcast this transfer, charge their quota.
329     // Self-broadcast transfers (where msg.sender is just the
330     // owner of the input note and isn't on the relayer registry)
331     // skip quota -the user pays L2 gas directly.
332     if (
333         energySettlement != address(0) &&
334         IEnergySettlement(energySettlement).isRegistered(msg.sender)
335     ) {
336         IEnergySettlement(energySettlement).consumeForRelayer(msg.sender, transferGasCharge);
337     }
338
339     // Insert new commitment
340     commitmentTree.insert(_newCommitment);
```

Listing 2.14: contracts/core/Shield.sol

Impact An attacker can exhaust tree capacity, preventing normal users from performing private transfers.

Suggestion Increase the capacity of the Merkle tree.

Feedback from the project The project states that the current implementation adopts a Merkle tree of depth 32. Based on the deployment network's throughput, an attacker would need many years to fill the tree.

2.1.8 Potential privacy leakage

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Shield`, functions `deposit()` and `transfer()` insert the caller-supplied `_commitment` into the Merkle tree via function `insert()`. In function `deposit()`, the commitment

is linked to the depositor (i.e., `msg.sender`), while function `transfer()` is expected to be submitted by a relayer to break that link. However, function `insert()` does not check whether a commitment already exists in the tree, allowing transfer to reuse a commitment that was already inserted during a deposit. When multiple leaves share the same commitment, observers can link the transfer back to the original depositor, leading to potential privacy leakage.

```
188       // Insert commitment into Merkle tree
189       uint32 leafIndex = commitmentTree.insert(_commitment);
190
191       emit Deposit(_commitment, leafIndex, block.timestamp, _token, _amount, _encryptedNote);
```

Listing 2.15: contracts/core/Shield.sol

```
339       // Insert new commitment
340       commitmentTree.insert(_newCommitment);
341
342       emit Transfer(_nullifierHash, _newCommitment, _encryptedNote);
```

Listing 2.16: contracts/core/Shield.sol

```
82    function insert(
83        TreeData storage self,
84        uint256 _leaf
85    ) internal returns (uint32 index) {
86        uint32 _nextIndex = self.nextIndex;
87        require(_nextIndex < uint32(2) ** self.levels, "Merkle tree is full");
88
89        uint32 currentIndex = _nextIndex;
90        uint256 currentLevelHash = _leaf;
91        uint256 left;
92        uint256 right;
```

Listing 2.17: contracts/libraries/MerkleTree.sol

Impact Reusing a commitment may correlate different leaves, leading to potential privacy leakage.

Suggestion Add uniqueness validation on `_commitment` to reject a commitment that already exists in the Merkle tree.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to remove the following unused code for better code readability and avoid misuse.

1. Contracts `Poseidon` and `Verifier` are currently not used, and their implementations do not match the real Poseidon hash or the actual circuit verifier.

2. In contract `Shield`, function `withdraw()` invokes `EnergySettlement.consumeForRelayer()` when `msg.sender == _relayer`. The annotation, "lets users pay gas with the L1 energy they have delegated to a relayer pool", implies that the contract settles cross-chain Energy. However, the function does not implement cross-chain settlement. Additionally, a registered relayer incurs the same gas cost as an unregistered broadcaster.

Suggestion Remove unused code to avoid misuse.

2.2.2 Revise the misleading annotation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `TokenRegistry`, the annotation lists SBT (Soulbound Token) and ERC721 as future-supported asset types. However, both conflict with the current implementation. First, SBTs are non-transferable, whereas the deposit operation requires transferring the token to the protocol. Second, the contract exposes the function `registerERC721()`. It is recommended to revise the misleading annotation and remove the unsupported function.

```
13 * Supported token types:
14 * - Native token (ATOS)
15 * - ERC20 tokens
16 * - ERC721 NFTs (future)
17 * - SBT (Soulbound Tokens) (future)
```

Listing 2.18: contracts/tokens/TokenRegistry.sol

```
156   /**
157    * @notice Register a new ERC721 token (NFT)
158    * @param _token NFT contract address
159    * @return tokenId The assigned token ID
160    */
161   function registerERC721(
162       address _token
163   ) external onlyOwner returns (uint256 tokenId) {
```

Listing 2.19: contracts/tokens/TokenRegistry.sol

Suggestion Revise these misleading annotations.

2.3 Note

2.3.1 Assumptions of `_encryptedNote`

Introduced by `Version 1`

Description In contract `Shield`, both functions `deposit()` and `transfer()` accept a caller-supplied `_encryptedNote` parameter and emit it in events. The contract cannot verify this parameter, and it is not bound to the submitted proof. We have the following assumptions about this parameter.

1. The Atoshi Privacy SDK is responsible for creating, encrypting, scanning, and decrypting `_encryptedNote`. Wallet recovery and note discovery depend on these mechanisms being implemented correctly.
2. Private transfers must be submitted through a private transaction channel to prevent front-running. If they are submitted through the public mempool, an observer can see the pending transaction, front-run it, and submit the same valid proof with a forged `_encryptedNote`. Because the function `transfer()` does not restrict `msg.sender`, this front-run transaction executes successfully, preventing the legitimate recipient from discovering the note.

Feedback from the project

1. The project states that the validation of `_encryptedNote` cannot be enforced at the contract level. Instead, the SDK guarantees correctness by uploading the correct `_encryptedNote` and filtering out fake notes.
2. Function `transfer()` does not restrict `msg.sender` by design. By default, the SDK submits transactions through the relayer endpoint with a private mempool and disables direct broadcasting.

2.3.2 Shared Merkle tree across assets

Introduced by `Version 1`

Description Contract `Shield` maintains a global `commitmentTree` for all supported tokens. Function `deposit()` inserts commitments for different tokens into the same Merkle tree, and the function `transfer()` continues to insert new commitments into the same tree. Therefore, commitments are not separated by asset, and asset integrity depends on the circuit correctly binding each note to its token.

```
81   // Merkle tree for commitments
82   MerkleTree.TreeData private commitmentTree;
```

Listing 2.20: contracts/core/Shield.sol

```
188      // Insert commitment into Merkle tree
189      uint32 leafIndex = commitmentTree.insert(_commitment);
190
191      emit Deposit(_commitment, leafIndex, block.timestamp, _token, _amount, _encryptedNote);
```

Listing 2.21: contracts/core/Shield.sol

Feedback from the project The project confirms that the Merkle tree is shared across assets to enhance privacy.

2.3.3 Potential centralized risk

Introduced by `Version 2`

Description In this project, the contract `Shield` owner can conduct sensitive operations. These operations include adding or removing supported tokens, updating verifier contracts, changing protocol fees and fee recipients, and pausing the privacy pool. If the private key of the privileged account is lost or maliciously exploited, it could pose a significant risk to the protocol.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS