FAIRYPROOF

PQC Vault

AUDIT REPORT

Version 1.0.0

Serial No. 20260526000022027

Presented by Fairyproof

May 26, 2026

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the qVAULT project.

Audit Start Time:

May 5, 2026

Audit End Time:

May 26, 2026

Audited Code's Github Repository:

https://github.com/01-Quantum/qvault-contracts-evm

Audited Code's Github Commit Number When Audit Started:

6e4599ec258c7b54ab70359a8c2a677a82bd5da2

Audited Code's Github Commit Number When Audit Ended:

b870c99c1763cdf70cb26e20fffc2c4b35a4289d

Audited Source Files:

The source files audited include all the files as follows:

```
contracts
├── interface
│   ├── IFeePolicy.sol
│   ├── IPQCAuthorizer.sol
│   └── IPQCLock.sol
├── PQCVault.sol
└── ZeroFeePolicy.sol

2 directories, 5 files
```

The goal of this audit is to review qVAULT's solidity implementation for its Vault function, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

This audit only applies to the specified code, software or any materials supplied by the qVAULT team for specified versions. Whenever the code, software, materials, settings, environment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code Review, Including:

- Project Diagnosis

Understanding the size, scope and functionality of your project's source code based on the specifications, sources, and instructions provided to Fairyproof.

- Manual Code Review

Reading your source code line-by-line to identify potential vulnerabilities.

- Specification Comparison

Determining whether your project's code successfully and efficiently accomplishes or executes its functions according to the specifications, sources, and instructions provided to Fairyproof.

2. Testing and Automated Analysis, Including:

- Test Coverage Analysis

Determining whether the test cases cover your code and how much of your code is exercised or executed when test cases are run.

- Symbolic Execution

Analyzing a program to determine the specific input that causes different parts of a program to execute its functions.

3. Best Practices Review

Reviewing the source code to improve maintainability, security, and control based on the latest established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following source(s) of truth about how the token issuance function should work:

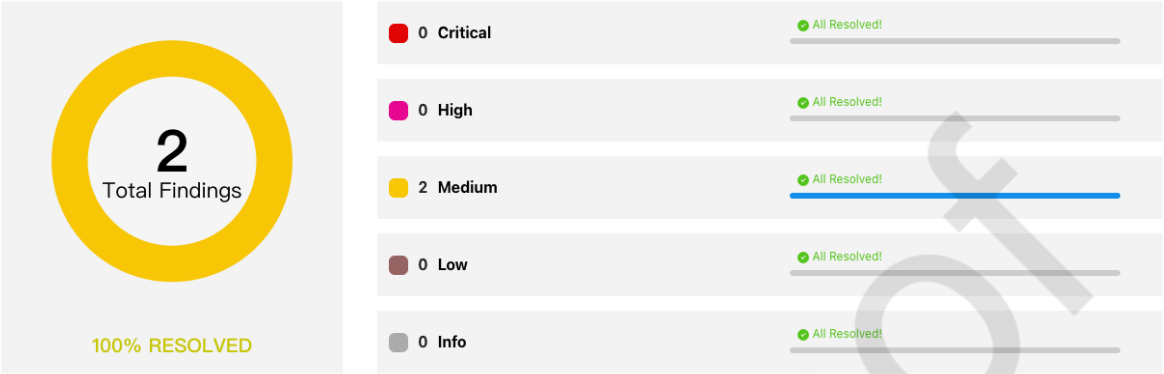
Website:https://qvault.qonetoken.io/

Source Code: https://github.com/01-Quantum/qvault-contracts-evm

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the qVAULT team or reported an issue.

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2026052600022027	Fairyproof Security Team	May 5, 2026 - May 26, 2026	Passed



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, two issues of medium-severity were uncovered. The qVAULT team fixed two issues of medium, and acknowledged the remaining issues.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to qVAULT

qVAULT applies security principles that are similar to the multi-signature wallets commonly used throughout Web3. In a standard multi-signature setup, two or more signatures are needed to release assets from a contract. In the case of qVAULT, the smart contract requires an additional signature that must be signed by a quantum resilient private key powered by Falcon (FN-DSA) encryption. As a result, a malicious actor cannot withdraw funds even if they compromise the classical key. qVAULT ensures protection at the smart contract level while maintaining speed and interoperability for users and developers. At launch, qVAULT will support `$qONE` and `$HYPE` on `HyperEVM`. More cryptocurrencies will be added in the near

future.

The above description is quoted from relevant documents of qVAULT.

04. Major functions of audited code

PQCVault is a per-beneficiary custody vault for ETH and ERC-20 assets, designed for secure day-to-day use with post-quantum authorization on normal withdrawals.
At launch, it runs with a zero-fee policy by default, while retaining the flexibility to introduce fees later through a replaceable fee-policy module. Any fee-policy change is controlled by a timelocked, PQC-gated upgrade flow, so monetization can evolve without weakening upgrade safety.
The vault also includes a timelocked lock-upgrade mechanism and a two-tier resilience model: a PQC-based emergency mode first, followed by delayed account recovery as a last resort.
Overall, PQCVault combines practical usability, gradual business flexibility, and strong operational safeguards for lock failures and upgrade risk.

05. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Access Control
- Admin Rights
- Arithmetic Precision
- Code Improvement
- Contract Upgrade/Migration
- Delete Trap
- Design Vulnerability
- DoS Attack
- EOA Call Trap
- Fake Deposit
- Function Visibility
- Gas Consumption
- Implementation Vulnerability
- Inappropriate Callback Function

- Injection Attack
- Integer Overflow/Underflow
- IsContract Trap
- Miner's Advantage
- Misc
- Price Manipulation
- Proxy selector clashing
- Pseudo Random Number
- Re-entrancy Attack
- Replay Attack
- Rollback Attack
- Shadow Variable
- Slot Conflict
- Token Issuance
- Tx.origin Authentication
- Uninitialized Storage Pointer

06. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Informational is not an issue or risk but a suggestion for code improvement.

07. Major areas that need attention

Based on the provided source code the Fairyproof team focused on the possible issues and risks related to the following functions or areas.

- Function Implementation

We checked whether or not the functions were correctly implemented.
We found one issue, for more details please refer to [FP-2] in "09. Issue description".

- Access Control

We checked each of the functions that could modify a state, especially those functions that could only be accessed by owner or administrator
We didn't find issues or risks in these functions or areas at the time of writing.

- Token Issuance & Transfer

We examined token issuance and transfers for situations that could harm the interests of holders.
We didn't find issues or risks in these functions or areas at the time of writing.

- State Update

We checked some key state variables which should only be set at initialization.
We didn't find issues or risks in these functions or areas at the time of writing.

- Asset Security

We checked whether or not all the functions that transfer assets were safely handled.
We found one issue, for more details please refer to [FP-1] in "09. Issue description".

- Miscellaneous

We checked the code for optimization and robustness.
We didn't find issues or risks in these functions or areas at the time of writing.

08. List of issues by severity

Index	Title	Issue/Risk	Severity	Status
FP-1	No emergency asset-withdrawal mechanism	Design Vulnerability	Medium	✓ Fixed
FP-2	Missing target confirmation parameter in finalize upgrade	Code Improvement	Medium	✓ Fixed

09. Issue descriptions

[FP-1] No emergency asset-withdrawal mechanism

Design Vulnerability Medium ✓ Fixed

Issue/Risk: Design Vulnerability

Description:

1. User withdrawals and owner lock-upgrade finalization both depend on `pqcLock.authorize`.
2. If an incorrect or non-functional lock is upgraded in, both withdrawals and future upgrades may be blocked.
3. In that state, user assets can become unavailable for a prolonged period.

Recommendation:

1. Add a constrained (preferably one-way) emergency mode and define a clear trigger policy.
2. Emergency activation procedure:
3. Owner activates emergency mode (with event emission and reason code).
4. Freeze lock upgrades and disable all new deposits.
5. Disable normal unlockETH/unlockERC20 paths that depend on pqcLock authorization.
6. Emergency withdrawal procedure:
7. Allow beneficiaries to withdraw only up to their recorded ledger balances (ETH and ERC20 separately).
8. Use direct balance accounting checks in the vault as the source of truth.
9. Emit dedicated emergency-withdraw events for auditability.
10. Safety controls:

11. Emergency mode cannot be turned off (or requires a stronger governance path than normal owner actions).
12. Keep reentrancy protection and zero-address/amount checks enabled.
13. Define strict access control, on-chain logs, and monitoring alerts for all emergency operations.

Update/Status:

The team has fixed the issue.

[FP-2] Missing target confirmation parameter in finalize upgrade

Code Improvement Medium ✓ Fixed

Issue/Risk: Code Improvement

Description:

1. Upgrade flow is propose/cancel/finalize.
2. Finalize only checks `pendingPQCLock != 0`, without requiring the caller to restate the intended target.
3. After multiple propose/cancel operations, operator confusion is possible.

Recommendation:

1. Add `_pendingPQCLock` argument to `finalizePQCLockUpgrade`.
2. Enforce `_pendingPQCLock == pendingPQCLock` at execution time.
3. Revert on mismatch to reduce operational mistakes.

Update/Status:

The team has fixed the issue.

10. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

- Consider managing the owner's access control with great care and transfering it to a multi-sig wallet or DAO when necessary.

11. Appendices

11.1 Unit Test

1. UnittestOutput

```
Running node:test tests

  ZeroFeePolicy
    ✔ version() returns the expected string
    ✔ every quoter returns (0, zero address)

  PQCVault: fee constants
    ✔ exposes MAX_FEE_BPS = 100 (1%)

  PQCVault: lockETH fee mechanics
    ✔ ZeroFeePolicy credits full msg.value and emits no fee event
    ✔ non-zero fee: credits net, routes fee, emits LockFeeCharged with net
ETHLocked
    ✔ consults the policy on every lock (quote revert propagates)
    ✔ reverts FeeExceedsCap when policy quotes above MAX_FEE_BPS
    ✔ fee exactly at MAX_FEE_BPS cap is accepted (strict > boundary)
    ✔ reverts FeeRecipientZeroAddress when fee > 0 and recipient is zero
    ✔ fee == 0 with zero recipient is allowed (no fee transfer, no fee event)
    ✔ reverts ETHTransferFailed when fee recipient rejects ETH
    ✔ blocks reentrancy via a malicious fee recipient (re-entry surfaces as
ETHTransferFailed; positive control succeeds when disarmed)

  PQCVault: receive() fee mechanics
    ✔ credits msg.sender net of the lockETH-side fee

  PQCVault: lockERC20 fee mechanics
    ✔ ZeroFeePolicy credits full delta and emits no fee event
    ✔ non-zero fee: credits net, routes fee in tokens, emits LockFeeCharged
    ✔ reverts FeeExceedsCap; no tokens move to the vault or fee recipient
    ✔ reverts FeeRecipientZeroAddress when fee > 0 and recipient is zero
    ✔ policy quote revert propagates and rolls back the token pull
    ✔ fee is applied on the post-FoT delta when token is fee-on-transfer

  PQCVault: unlockETH fee mechanics
    ✔ ZeroFeePolicy pays full amount and emits no fee event
    ✔ non-zero fee: pays net to `to`, routes fee, emits UnlockFeeCharged with
net ETHUnlocked
    ✔ reverts FeeExceedsCap; ledger and PQC nonce untouched
    ✔ reverts FeeRecipientZeroAddress when fee > 0 and recipient is zero
```

```
    ✔ policy quote revert propagates and leaves ledger untouched
    ✔ reverts ETHTransferFailed when the user `to` rejects ETH (with fee > 0)
    ✔ reverts ETHTransferFailed when the fee recipient rejects ETH
    ✔ blocks reentrancy via a malicious fee recipient on unlock (positive
control proves the guard caused it)

  PQCVault: unlockERC20 fee mechanics
    ✔ ZeroFeePolicy pays full amount; no fee event
    ✔ non-zero fee: pays net to `to`, routes fee in tokens, emits
UnlockFeeCharged with net ERC20Unlocked
    ✔ reverts FeeExceedsCap; ledger and PQC nonce untouched

  PQCVault: proposeFeePolicyUpgrade
    ✔ reverts when called by non-owner
    ✔ reverts on zero address
    ✔ reverts on same address as current policy
    ✔ reverts when a proposal is already pending
    ✔ reverts InvalidFeePolicyInterface when candidate's version() reverts
    ✔ stores pending, sets timelock, and emits FeePolicyUpgradeProposed

  PQCVault: cancelFeePolicyUpgrade
    ✔ reverts when called by non-owner
    ✔ reverts when nothing is pending
    ✔ clears state and emits FeePolicyUpgradeCancelled
    ✔ allows re-proposing a different policy after cancel

  PQCVault: finalizeFeePolicyUpgrade
    ✔ reverts when called by non-owner
    ✔ reverts when nothing pending
    ✔ reverts when expectedPendingFeePolicy does not match
    ✔ reverts when the timelock has not yet expired
    ✔ reverts when pqcLock rejects authorize; no policy swap
    ✔ swaps feePolicy, clears pending, and emits FeePolicyUpdated
    ✔ uses the new policy for subsequent operations after a successful
upgrade

  PQCVault: probePendingFeePolicyUpgrade
    ✔ reverts when nothing is pending
    ✔ returns OK and the version string for a healthy candidate
    ✔ returns INTERFACE_ERROR when version() reverts after proposal

  PQCVault: activatePQCEmergencyMode + fee policy
    ✔ cancels a pending fee policy upgrade and emits
FeePolicyUpgradeCancelled
    ✔ does not emit FeePolicyUpgradeCancelled when nothing was pending
    ✔ freezes the fee-policy upgrade flow under notEmergency

  PQCVault: emergency and recovery paths bypass the fee policy
    ✔ emergencyWithdrawETH does not consult the policy and pays full amount
    ✔ emergencyWithdrawERC20 does not consult the policy and pays full amount
    ✔ accountRecoveryWithdrawETH does not consult the policy and pays full
amount
```

```
    ✔ accountRecoveryWithdrawERC20 does not consult the policy and pays full
amount

  PQCVault: constructor
    ✔ reverts if pqcLock is the zero address
    ✔ reverts if recoveryPQCLock is the zero address
    ✔ reverts if feePolicy is the zero address
    ✔ reverts if initialOwner is the zero address (OZ Ownable check)
    ✔ stores pqcLock, recoveryPQCLock, feePolicy, and owner; emits
PQCLockUpdated, RecoveryPQCLockSet, FeePolicyUpdated

  PQCVault: lockETH
    ✔ reverts on zero beneficiary
    ✔ reverts on zero msg.value
    ✔ credits msg.value to beneficiary and emits ETHLocked
    ✔ accumulates multiple deposits to the same beneficiary

  PQCVault: receive (self-payable)
    ✔ credits a plain ETH transfer to msg.sender
    ✔ reverts a zero-value plain ETH send

  PQCVault: unlockETH
    ✔ reverts on zero recipient
    ✔ reverts on zero amount
    ✔ reverts on insufficient balance
    ✔ reverts when pqcLock.authorize returns false
    ✔ reverts with ETHTransferFailed when recipient rejects ETH
    ✔ fully reverts state when ETH transfer fails after authorize
    ✔ debits balance, transfers ETH, and emits ETHUnlocked on success
    ✔ supports repeated partial withdrawals against the same binding
    ✔ blocks reentrancy via a malicious pqcLock

  PQCVault: lockERC20
    ✔ reverts on zero token address
    ✔ reverts on zero beneficiary
    ✔ reverts on zero amount
    ✔ pulls tokens, credits beneficiary, and emits ERC20Locked
    ✔ accumulates multiple deposits to the same beneficiary
    ✔ credits the observed delta for fee-on-transfer tokens
    ✔ reverts AmountZero when fee-on-transfer fully consumes the deposit
    ✔ reverts when allowance is missing (bubbles up the ERC20 error)

  PQCVault: unlockERC20
    ✔ reverts on zero token address
    ✔ reverts on zero recipient
    ✔ reverts on zero amount
    ✔ reverts on insufficient balance
    ✔ reverts when pqcLock.authorize returns false
    ✔ debits balance, transfers tokens, and emits ERC20Unlocked
    ✔ supports repeated partial ERC20 withdrawals

  PQCVault: views
```

```
    ✔ getETHBalance returns 0 by default and the credit after a lock
    ✔ getERC20Balances returns aligned per-token balances
    ✔ getERC20Balances returns an empty array for empty input

  PQCVault: proposePQCLockUpgrade
    ✔ reverts when called by non-owner
    ✔ reverts on zero address
    ✔ reverts when proposing the current lock
    ✔ reverts when a proposal is already pending
    ✔ reverts when the candidate fails the version() probe
    ✔ stores pending state and emits PQCLockUpgradeProposed on success

  PQCVault: cancelPQCLockUpgrade
    ✔ reverts when not owner
    ✔ reverts when nothing is pending
    ✔ clears pending state and emits PQCLockUpgradeCancelled
    ✔ resets timelock when a cancelled proposal is replaced

  PQCVault: finalizePQCLockUpgrade
    ✔ reverts when not owner
    ✔ reverts when nothing is pending
    ✔ reverts when the timelock has not yet expired
    ✔ reverts when expected pending lock does not match on-chain pending
    ✔ reverts when the current lock rejects authorize
    ✔ swaps pqcLock, clears pending state, and emits PQCLockUpdated
    ✔ uses the new lock for subsequent unlocks after a successful upgrade

  PQCVault: probePendingPQCLockUpgrade
    ✔ reverts when nothing is pending
    ✔ returns OK and the version string for a healthy candidate
    ✔ returns INTERFACE_ERROR when version() reverts after proposal

  PQCVault: transferOwnershipPQC
    ✔ reverts when called by non-owner
    ✔ reverts on zero new owner (OwnableInvalidOwner)
    ✔ reverts when the lock rejects authorize
    ✔ transfers ownership and emits OwnershipTransferred on success
    ✔ clears the internal guard after a successful transfer (next OZ
transferOwnership still reverts)
    ✔ enforces new owner authority across lock-upgrade actions after transfer

  PQCVault: disabled OZ ownership paths
    ✔ transferOwnership(addr) always reverts with UseTransferOwnershipPQC
    ✔ renounceOwnership() always reverts with OwnershipRenouncementDisabled

  PQCVault: emergency recovery flow
    ✔ activates PQC emergency mode and freezes normal unlock path
    ✔ supports fallback account-recovery withdrawals after 7-day timelock

  PQCVault integrated with real EthFalcon
    ✔ unlockETH succeeds with a real Falcon proof and bumps nonce (219ms)
    ✔ replay proof fails on second unlockETH call (180ms)
```

```
    ✔ unlockETH rejects a signature from a wrong private key (154ms)
    ✔ unlockETH rejects proof signed for another account
    ✔ supports sequential partial withdrawals with nonce 0 then 1 (183ms)
    ✔ unlockERC20 succeeds with a real Falcon proof (98ms)
    ✔ finalizePQCLockUpgrade works with real EthFalcon owner proof (96ms)
    ✔ authorize/unlock domain separation prevents cross-action replay
    ✔ finalizePQCLockUpgrade rejects wrong proof, stale nonce proof, and
authorizeUpgrade-domain proof (226ms)
    ✔ after upgrade to a new EthFalcon, old-lock proof fails while new-lock
proof succeeds (238ms)

    gas: falcon_core.verify = 1369400
  falcon_core: end-to-end signature verification
    ✔ accepts a valid Falcon-512 signature produced by the WASM reference
    ✔ rejects a signature when the message is tampered

  falcon_core_kat: NIST-submission Known-Answer Tests
    ✔ verifies KAT vector count=0 (mlen=33)
    ✔ verifies KAT vector count=1 (mlen=66)
    ✔ verifies KAT vector count=2 (mlen=99)
    ✔ verifies KAT vector count=3 (mlen=132)
    ✔ verifies KAT vector count=4 (mlen=165)
    ✔ verifies KAT vector count=50 (mlen=1683)
    ✔ verifies KAT vector count=99 (mlen=3300)

  falcon_core lane-range guard (_allLanesBelowQ via falcon_core)
    ✔ accepts a baseline all-zero input
    ✔ rejects when any s2 compact lane is >= q
    ✔ rejects when any ntth compact lane is >= q
    ✔ rejects a max 16-bit lane value (0xffff)

  EthFalcon off-chain helpers: signPoly over hashToPointEVM polynomial
    ✔ is deterministic: signPoly over the same (hm, sk) yields the same
polynomial

  EthFalcon: lock / authorize / unlock with deterministic Falcon-512 seed
    ✔ version() returns the expected string
    ✔ PQ_DOMAIN_SALT is exactly 40 bytes and matches the documented constant
    ✔ constructor emits AuthorizationLocked +
AuthorizationChanged(address(this), false) exactly once
    ✔ lock() stores the Falcon NTT public key limbs in pqPublicKey
    ✔ lock() returns false when the account is already locked (state
untouched)
    ✔ lock() rejects a malformed proof (wrong byte length)
    ✔ lock() leaves other accounts independent
    ✔ authorize() verifies Falcon, increments nonce, and keeps pqPublicKey
    ✔ authorize() returns true when there is no binding; proof ignored; state
untouched
    ✔ authorize() with empty proof returns true when unbound (does not
revert)
    ✔ authorize() then unlock() requires the next-nonce signature (114ms)
[EthFalcon gas] lock=757189 unlock=1365447 total=2122636
```

```
    ✔ lock() and unlock() gas (logged on stdout) (96ms)
    ✔ unlock() releases the binding when given a valid Falcon signature
(94ms)
    ✔ unlock() can be relayed by an arbitrary msg.sender (93ms)
    ✔ unlock() returns false if the account has no binding (state untouched)
    ✔ unlock() returns false when the s2 component is tampered (binding
preserved)
    ✔ unlock() reverts with MalformedProof on wrong proof length
    ✔ authorize() reverts with InvalidS2Length on ABI-valid but short decoded
s2
    ✔ unlock() reverts with InvalidS2Length on ABI-valid but short decoded s2
    ✔ nonce is monotonic across lock/unlock cycles and prevents replay
(209ms)
    IPQCAuthorizer
      ✔ constructor self-locks ntth on address(this)
      ✔ isAuthorized is true iff the account has NO Falcon key bound (lock
revokes, unlock restores) (146ms)
      ✔ authorizeUpgrade runs Falcon and bumps pqNonce[address(this)]
      ✔ authorizeUpgrade digest binds tokenContract — swapping it returns
false (state untouched)
      ✔ authorizeUpgrade digest binds newAuthorizer — swapping it returns
false
      ✔ authorizeUpgrade reverts when msg.sender != sender
      ✔ authorizeUpgrade reverts if currentAuthorizer is not this contract
      ✔ authorizeUpgrade reverts for zero tokenContract
      ✔ authorizeUpgrade reverts for zero newAuthorizer
      ✔ authorizeUpgrade returns false on wrong s2 (state untouched)
      ✔ authorizeUpgrade reverts with MalformedProof on wrong proof length
      ✔ authorizeUpgrade reverts with InvalidS2Length on ABI-valid short
decoded s2
      ✔ a second authorizeUpgrade must use the next nonce in the digest
(78ms)

  hashToPointEVM: TS port matches Solidity
    ✔ matches on vector: all-zero salt, all-zero msg
    ✔ matches on vector: all-0xff salt, all-0xff msg
    ✔ matches on vector: incrementing bytes
    ✔ matches on vector: deterministic-seed #1
    ✔ matches on vector: deterministic-seed #2
    ✔ matches on vector: deterministic-seed #3
    ✔ matches fixed golden coefficients on zero vector
    ✔ matches Solidity across 64 deterministic seeded vectors (108ms)


  ✔ helpers.ts

    gas: NTT_HALFMUL_Compact = 1032046
    gas: NTT_Expand          = 144520
  NTT_falcon: JS vs Solidity half-multiply
    ✔ matches Solidity NTT_HALFMUL_Compact + expand on deterministic inputs
    ✔ compact ↔ expand round-trips in JS
    ✔ NTT_Compact masks oversized lanes (defense-in-depth)
```

```
194 passing (194 nodejs)
```

11.2 External Functions Check Points

1. PQCVault.sol.md

File: contracts/PQCVault.sol

contract: PQCVault is Ownable

(Custom interfaces exclude members inherited from third-party dependencies)

Index	Custom Interface	StateMutability	Modifier	UnitTest	Miscellaneous
1	lockETH(address)	payable	`notEmergency`	Passed	
2	receive()	payable	`notEmergency`	Passed	
3	unlockETH(uint256, address, bytes)		`nonReentrant` `notEmergency`	Passed	
4	lockERC20(address, address, uint256)		`nonReentrant` `notEmergency`	Passed	
5	unlockERC20(address, uint256, address, bytes)		`nonReentrant` `notEmergency`	Passed	
6	getETHBalance(address)	view			
7	getERC20Balances(address[], address)	view			
8	proposePQCLockUpgrade(address)		`onlyOwner` `notEmergency`	Passed	
9	cancelPQCLockUpgrade()		`onlyOwner` `notEmergency`	Passed	
10	finalizePQCLockUpgrade(address, bytes)		`onlyOwner` `nonReentrant` `notEmergency`	Passed	
11	probePendingPQCLockUpgrade()	view			
12	transferOwnershipPQC(address, bytes)		`onlyOwner` `nonReentrant`	Passed	
13	renounceOwnership()	pure			
14	activatePQCEmergencyMode(bytes, bytes32)		`onlyOwner` `nonReentrant`	Passed	
15	emergencyWithdrawETH(uint256, address, bytes)		`nonReentrant` `onlyPQCEmergency`	Passed	
16	emergencyWithdrawERC20(address, uint256, address, bytes)		`nonReentrant` `onlyPQCEmergency`	Passed	
17	proposeAccountRecoveryMode(bytes32)		`onlyOwner`	Passed	
18	finalizeAccountRecoveryMode()		`onlyOwner`	Passed	
19	accountRecoveryWithdrawETH(uint256)		`nonReentrant` `onlyAccountRecovery`	Passed	
20	accountRecoveryWithdrawERC20(address, uint256)		`nonReentrant` `onlyAccountRecovery`	Passed	

FAIRYPROOF

https://medium.com/@FairyproofT

https://twitter.com/FairyproofT

https://www.linkedin.com/company/fairyproof-tech

https://t.me/Fairyproof_tech

Reddit: https://www.reddit.com/user/FairyproofTech