

BLOCKAID TECHNICAL EXPLANATION

Lemnicoin (LEMNI) — Polygon mainnet contract
0x02011c7B7D9C6dfc563c92197557e8BE490a85b4

Public source repository: <https://github.com/rmvince/lemnicoin-contracts> · Patent: European Patent Application **EP 26300368.3** and U.S. Application No. **19/745,426**, “Method for performing a transaction on a blockchain” (the “Patent Text”; quoted verbatim throughout).

Prepared October 2026. On-chain state cited as of Polygon block 95,237,490 (2026-10-09).

1. Purpose

MetaMask displays, for the LEMNI token, a “Malicious” label and five designations sourced from Blockaid: “*Suspicious code*”, “*Rugpull risk*”, “*Known malicious*”, “*Resembles malicious contract*”, and “*Possible price manipulation*”. Each designation is false, and the cause of the error is identifiable. LEMNI implements an **extremely novel**, patent-pending method — a token architecture in which the standard interface functions are overridden by post-quantum ownership-key verification (EP 26300368.3 / U.S. App. No. 19/745,426). Automated scanners, calibrated on the patterns of conventional tokens, have no category for this design, and that novelty is precisely why Blockaid’s software misreads the contract’s protective code as nefarious. Nor is the harm confined to MetaMask: because Blockaid’s threat feed is integrated across the industry, the other wallets and crypto applications that employ it republish this same damaging misrepresentation of our product to their own users. The explanation given herein is provided to ameliorate that misunderstanding — a misunderstanding that is actively harming our work product by interposing a false “Malicious” warning between the token and every prospective user, on every surface that consumes the feed. We submit it in a spirit of clarification, for the benefit of all — Blockaid and its integrators included, whose designations are only as sound as the analysis beneath them.

This document demonstrates the falsity of each designation against three sources any reviewer can check independently: (i) the public Solidity source of the deployed contract at github.com/rmvince/lemnicoin-contracts; (ii) the deployed contract’s live state on Polygon, read directly over public JSON-RPC; and (iii) the patent application whose method the contract implements, quoted verbatim below. For *each* designation, the demonstration follows the same three steps: **where the deployed, on-chain-verifiable code proves the designation false; how that same code comports to the patent; and why the mechanism’s novelty causes an automated scanner to read it as something other than what it is — beneficially functional protection**. Section 10 lists the exact calls and commands that reproduce every figure cited here.

The central point is simple: **the code patterns Blockaid’s scanner flagged are the claimed elements of a filed patent for quantum-resistant, theft-proof token ownership**. The designations do not merely mischaracterize the contract — they label the token’s protective mechanism as its danger.

2. Summary

Designation	Controlling fact	§
“Suspicious code”	Every one of the 209 function selectors live on the contract resolves to a function in the public repository, and the flagged constructs are the ERC-20 embodiment required by Claims 1–11 of the Patent Text.	4, 5
“Rugpull risk”	Every Uniswap V3 liquidity position is held by a lockbox contract with no withdrawal path before 2027-10-07, and afterward only to a 2-of-3 multisig with a Falcon-512 post-quantum signature; the ECDSA-fallback field is immutably zero. Supply has been 10,500,000 at every sampled point of the contract’s life.	6
“Known malicious”	No exploit, incident, or victim report exists; the contract has operated for 226 days without a recorded incident. “Known” asserts an established fact for which there is no instance.	7
“Resembles malicious contract”	The resemblance is a heuristic misfire on the patented verification gate: holders who never registered post-quantum keys transfer under standard ERC-20 semantics, and registered holders retain a signed sell path into the whitelisted pool.	8
“Possible price manipulation”	No transfer tax, no rebasing, no supply lever below a dual-gated hard cap, liquidity time- and signature-locked; price is set solely by public swaps on Uniswap V3.	9

3. The public repository is the deployed code

The repository <https://github.com/rmvince/lemnicoin-contracts> contains the Solidity source of the deployed LEMNI contract — **the code in this repository is the same code that is on-chain** at `0x02011c7B7D9C6dfc563c92197557e8BE490a85b4`. The contract is an EIP-2535 “Diamond” whose facets are built from the repository’s `diamond/` tree. Two independent, reproducible checks establish the identity:

- **Interface identity.** The live loupe call `facets()` returns 24 facets exposing 209 function selectors. Every one of the 209 selectors resolves to a function defined in this repository’s source — with zero unknowns.
- **Bytecode identity.** Compiling this repository with the deployment toolchain (solc 0.8.24/0.8.25, optimizer enabled, runs = 1, via-IR, EVM target “cancun”) reproduces, byte for byte, the runtime bytecode served by `eth_getCode` at the deployed facet addresses — including the facets that carry every function discussed in this document: `TokenFacet (0xa17dFA47...)`, `GovernanceFacet (0x0F66cF64...)`, `DiamondCutFacet (0x16c00350...)`, `OwnershipFacet (0x6e0AC722...)`, `DiamondLoupeFacet (0x1f871083...)`, `WalletWhitelistFacet (0xa70d9100...)`, `FeeExemptionFacet (0x1Adf8642...)`, `RingFacet (0x6F00d2fB...)`, `FalconGovernedCutFacet (0xD142C762...)`, `AttestorLivenessFacet (0x8c616A87...)` and `LAIPV3Facet (0x35bAF6e9...)`.

All file and line citations in this document refer to this public repository, so every citation can be opened in a browser. Section 10 gives the verification commands.

4. The Solidity files implement the patent — claim-by-claim

Table 1 sets out the claims of the Patent Text *verbatim* and maps each to its implementation in the repository. Table 2 maps the patent’s ERC-20 description, phrase by phrase, to the branches of the deployed `transfer()` / `approve()` / `transferFrom()` code. File paths are repository paths; line numbers are from the repository as published. The application’s framing passages — its own statements of the overriding mechanism, quantum resistance and platform scope — are quoted in Appendix A.

Table 1 — Patent claims (verbatim) → deployed implementation

Claim	Claim text (verbatim)	Implementation (repository citation)
1	<i>“Method for performing a transaction on a blockchain, said transaction involving a cryptographic wallet, said wallet being associated with a first pair of a private key and a public key, called wallet private key and wallet public key, said method comprising the following steps performed by a processor (11): – receiving (a) a request for performing said transaction, said request comprising at least one candidate piece of data; – verifying (b) the validity of the request by verifying that the at least one candidate piece of data derives from the second pair of a private key and a public key associated with the wallet called ownership private key and ownership public key, the second pair being different from the first pair.”</i>	Every protected transfer takes the ownership public key and a post-quantum signature as the candidate data: <code>diamond/facets/QuantumTransferFacet.sol:42–142</code> (<code>quantumTransfer</code> , <code>quantumTransferWithHolderKey</code> , <code>quantumTransferDual</code>); verification in <code>diamond/libraries/LibLemnicoin.sol:111–141</code> (<code>requireLaipVerification</code>).
2	<i>“...wherein the step of verifying (b) comprises at least one of the following sub-steps: – verifying (b1) that the candidate piece of data matches a commitment associated with the wallet and generated on the basis of the ownership public key, – verifying (b2) that the candidate piece of data comprises a valid ownership possession proof generated on the basis of the ownership private key.”</i>	(b1): the submitted LAIP key is hashed and compared to the stored commitment; mismatch reverts — <code>LibLemnicoin.sol:120–141</code> (<code>LAIPCommitmentMismatch</code>); also <code>QuantumTransferFacet.sol:241</code> . (b2): Falcon-512 possession proof verified on-chain by the ZKNOX verifier — <code>QuantumTransferFacet.sol:260–265</code> ; the dual path adds the holder’s own signature over the same digest — <code>QuantumTransferFacet.sol:121–142</code> .
3	<i>“...comprising a step of approving (c) the execution of the transaction if the request is considered valid at step (b).”</i>	Token movement is the last step, executed only after every verification passes: <code>QuantumTransferFacet.sol:133–141</code> .
4	<i>“...wherein the second pair comprises quantum-resistant keys.”</i>	LAIP (3072-bit lemniscate), ML-DSA-87 (Dilithium), Falcon-512 and SLH-DSA keys: <code>crypto/LAIPOnCurve.sol</code> ; <code>Dilithium5VerifierFull.sol</code> ; <code>falcon/</code> ; <code>crypto/SlhDsaVerifierV2.sol</code> .
5	<i>“Method according to claim 4, wherein the second pair complies with Lemniscate AGM Isogeny, Dilithium, Winternitz One-Time Signature or SPHINCS+ schemes.”</i>	Lemniscate scheme: <code>LemniscateCurve.sol</code> , <code>crypto/LAIPOnCurve.sol</code> ; Dilithium: <code>Dilithium5VerifierFull.sol</code> ; SPHINCS+ (SLH-

Claim	Claim text (verbatim)	Implementation (repository citation)
		DSA): crypto/S1hDsaSha2_128s.sol, crypto/S1hDsaVerifierV2.sol.
6	<i>"...wherein the step of verifying (b) further comprises verifying (b3), using the wallet public key, that the request comprises a valid wallet cryptographic signature generated on the basis of the wallet private key."</i>	Every protected function is callable only by the wallet itself (msg.sender), i.e. under the wallet's ECDSA transaction signature verified by the chain — wallet signature (b3) and ownership proof (b2) together form the dual authority.
7	<i>"...wherein the transaction comprises transferring tokens from the wallet to another wallet."</i>	All quantum transfer paths end in the balance move: diamond/libraries/LibERC20.sol:26–42.
8	<i>"...wherein the ownership private key is not stored on the blockchain."</i>	Only hashes and commitments are stored at registration: diamond/facets/KeyManagementFacet.sol:117–130.
9	<i>"...wherein the steps are performed by a smart contract or by a spending script implemented by the processor (11)."</i>	The EIP-2535 diamond at 0x02011c...85b4: diamond/LemnecoinDiamond.sol.
10	<i>"Method for creating a cryptographic wallet for performing transactions on the blockchain, said method comprising the following steps performed by a processor (11): – binding (γ) to the wallet a pair, called ownership pair, of a private key and a public key referred to as ownership private key and ownership public key, respectively, said wallet being associated with another pair, called wallet pair... the wallet pair being different from the ownership pair."</i>	Binding/enrolment of the ownership pair: KeyManagementFacet.sol:73–111 (bindQuantumKeys), 117–173 (registerQuantumKeys, registerQuantumKeysWithValidation).
11	<i>"...wherein the wallet is associated with a wallet's address and wherein the step of binding (γ) comprises the following sub-steps: – receiving (γ4) a function call from the wallet's address, said function call comprising at least an enrolment piece of data deriving from the ownership public key, – verifying (γ5) that the enrolment piece of data matches a verification piece of data deriving from the ownership public key which is stored on the blockchain, – if said verifying is successful, considering (γ6) the ownership pair as being bound to the wallet."</i>	Challenge (γ1–γ3): KeyManagementFacet.sol:49–67 (createBindingChallenge); function call from the wallet's own address with key data, verified and bound (γ4–γ6): KeyManagementFacet.sol:73–111.
13	<i>"...comprising a step of transferring tokens to said wallet, from another wallet which is also associated with the ownership pair, using a recovery piece of data deriving from the ownership pair."</i>	diamond/facets/RecoveryFacet.sol:40–89 (recoveryTransfer, recoveryTransferDual); V3: diamond/facets/LAIPV3CapabilitiesFacet.sol (recoveryTransferV3).
14, 15	<i>"Computer program product comprising code instructions for executing a method according to any one of claims 1 to 13... /</i>	The deployed bytecode on Polygon, reproducible from this repository (§3).

Claim	Claim text (verbatim)	Implementation (repository citation)
	<i>Storage medium readable by a computer...</i>	

Table 1a — Further description passages (verbatim) → deployed implementation

Patent Text (verbatim)	Implementation (repository citation)
<i>“...the step γ) of binding comprises a sub-step $\gamma 7$) of validating the public ownership key on-chain. For cryptographic schemes where public key validity can be checked mathematically — for example, verifying that a point lies on a specific algebraic curve — the smart contract may optionally perform this validation on-chain and cache the result. Subsequent transfers check the cache; if the ownership public key was never validated, the transaction is rejected as a potential forgery.”</i> — p. 17, ll. 17–23	On-chain lemniscate curve validation (~2.25M gas) with cached result: <code>KeyManagementFacet.sol:139–173, 301–320</code> (registerQuantumKeysWithValidation, validateLaipCurve; cache laipValidatedQHash).
<i>“...the step γ) of binding can only be performed once per wallet’s address... If ownership keys have already been bound to the wallet’s address, any subsequent call to the binding function is rejected. This one-time binding restriction is advantageous in that it prevents an attacker who has stolen the wallet private key from re-binding different ownership keys against the wallet’s address...”</i> — p. 18, ll. 23–28	Registration is a one-way ratchet; re-registration reverts: <code>KeyManagementFacet.sol:457–465</code> (<code>_enforceKeyImmutability</code>, <code>OwnershipKeysImmutable</code>), 273–275 (<code>HolderFalconKeyImmutable</code>). No code path ever clears a registration.
<i>“...each request advantageously carries a monotonically increasing nonce and an expiration deadline... Once the request is processed, the nonce is marked as consumed on-chain, preventing any reuse of a previously submitted request.”</i> — p. 10, ll. 12–14; p. 12, ll. 28–30	Nonce + deadline are part of the signed digest and consumed per transfer: <code>LibLemnicoin.sol:231</code> (<code>attestationDigest</code>), <code>QuantumTransferFacet.sol:248–267</code>.
<i>“A unified ownership key commitment incorporating the public keys of all derived key types is stored on-chain, creating a single cryptographic anchor that binds the user’s ownership identity across platforms and verification methods.”</i> — p. 16, ll. 21–24	Permanent <code>C_own</code> commitment: <code>KeyManagementFacet.sol:387–425</code> (registerQuantumKeysV2, getOwnershipCommitment).
<i>“Token balances remain queryable via the standard interface — for example, <code>balanceOf()</code> for ERC-20 — such that any wallet, block explorer, or portfolio tracker can read them without modification.”</i> — p. 15, ll. 11–13	<code>diamond/facets/TokenFacet.sol:31–58</code> (name, symbol, decimals, totalSupply, balanceOf, allowance unchanged).

Table 2 — The patent’s ERC-20 description ↔ the deployed transfer logic, phrase by phrase

Patent Text, p. 7 l. 25 – p. 8 l. 7, describes the preferred ERC-20 embodiment. The deployed TokenFacet implements it clause for clause (diamond/facets/TokenFacet.sol):

Patent Text (p. 7–8, verbatim)	Deployed code (TokenFacet.sol)
“The transfer() function rejects all general-purpose calls and permits only narrowly whitelisted operations, including...”	transfer() — lines 74–128; the final catch-all revert directs the caller to the ownership-key path: line 127 (revert UseTransferWithEnvelope()).
“...zero-amount no-ops...”	Lines 78–81: zero-amount transfers succeed as ERC-20-compliant no-ops.
“...deployment-phase owner transfers...”	Lines 84–87: owner distribution during the deployment phase.
“...whitelisted exchange withdrawals...”	Lines 101–104: outbound transfers from whitelisted CEX addresses.
“...governance-approved address pairs...”	Lines 107–110: governance-approved sender→recipient pairs.
“...whitelisted liquidity pool outflows...”	Lines 116–119: outbound transfers from whitelisted DEX pools (buys reach the buyer).
“...and active time-limited sessions.”	Lines 122–125: transfers to addresses in the caller’s active session.
“All other calls revert and direct the caller to the ownership-key-protected alternative.”	Line 127: the revert that heuristic scanners flag — prescribed verbatim by the application.
“The approve() function rejects all general-purpose calls and permits only approvals for governance-approved address pairs or whitelisted decentralised exchange infrastructure. All other calls revert and direct the caller to the ownership-key-protected alternative.”	approve() — lines 133–147; unregistered holders approve normally (135–138); registered holders are directed to the ownership-key approval (146).
“The transferFrom() function restricts the recipient to a narrow set of permitted addresses, such as the caller itself, whitelisted centralised exchange addresses, or whitelisted decentralised exchange pool contracts. Even a valid allowance does not permit transfer to arbitrary addresses.”	transferFrom() — lines 155–196; recipient gate at lines 186–191 (caller itself, whitelisted CEX, whitelisted DEX pool). Unregistered token owners are exempt (171–175).
“The burn() and burnFrom() functions are removed entirely and replaced by an ownership-key-protected burn alternative requiring the same cryptographic verification as transfers. This prevents an attacker who has obtained the wallet private key from maliciously destroying a user’s tokens.”	burn()/burnFrom() absent from TokenFacet (lines 198–201); the replacement is facets/QuantumBurnFacet.sol (quantumBurn — burns only msg.sender’s own tokens, lines 79, 119).

This correspondence is not approximate; it is the enumerated branch list of the deployed function. The constructs a scanner flags as “suspicious” are the claimed subject matter of the application.

5. “Suspicious code” — false

Where the deployed code proves it false. The entire source is public (github.com/rmvince/lemnicoi-contracts) and is the deployed code (§3): all 209 live selectors resolve to it, and compiling it reproduces the deployed runtime bytecode of the facets listed in §3. There is no hidden or obfuscated logic to be suspicious of: no `selfdestruct` exists anywhere in the repository, and the only `delegatecall` surfaces are the standard EIP-2535 dispatch (`diamond/LemnicoiDiamond.sol:60–65`) and the standard cut initializer (`diamond/libraries/LibDiamond.sol:249`). Everything the contract does is in the open, and verifiably so on-chain.

How that code comports to the patent. Every construct the heuristics react to — the conditional revert in `transfer()`, the recipient whitelist in `transferFrom()`, the removal of `burn()` — is a required element of the patented method, prescribed verbatim by the application and implemented clause for clause (Tables 1, 1a, 2).

Why the scanner misreads it. A pattern scanner labels “suspicious” whatever departs from the templates it was calibrated on. A first-of-its-kind architecture departs from every template by definition — the departure *is* the invention. What the scanner registers as suspicion is the novelty of a beneficially functional protection mechanism, not the presence of anything harmful.

Conclusion: the code is published, deployed as published, and is the preferred embodiment of EP 26300368.3 / U.S. 19/745,426. “Suspicious code” is false.

6. “Rugpull risk” — false

Where the deployed code proves it false. A rug-pull requires a path by which insiders take holders’ funds or the pool’s liquidity. Each such path is closed, and the closures are on-chain facts, not policies:

Liquidity cannot be pulled.

On-chain check (any RPC node)	Result (block 95,237,490)
Uniswap V3 position NFTs held by the lockbox <code>0xa7AFD3B1F694f6AadF8A81eCa97d25A155DA5D9C</code>	All five positions of the LEMNI/USDC pool: token IDs 2889463, 2889464, 2889703, 2889704, 2889725 (each confirmed <code>token0 = LEMNI</code> , <code>token1 = USDC</code> , fee 1%). Zero positions held by any EOA or by the owner multisig.
<code>lockbox.unlockTime()</code>	<code>1822929133 = 2027-10-07 17:12:13 UTC</code> . <code>withdrawAll()</code> reverts before this date (<code>safe/LemnicoiLpLockbox.sol:451</code>); the date can only ever be moved later (<code>extendUnlock</code> , lines 440–446).
<code>lockbox.ecdsaFallbackTime()</code> — immutable	0 = never: the Falcon-512 post-quantum signature requirement on withdrawal can never lapse (lines 473–500). <code>falconActive()</code> reads true.

On-chain check (any RPC node)	Result (block 95,237,490)
lockbox.owner() — immutable, sole withdrawal destination	The project's 2-of-3 Gnosis Safe (0xb19E5837...). withdrawAll() takes no destination argument.
Lockbox capabilities	No decreaseLiquidity, no swap, no proxy, no token rescue in the contract (safe/LemnicoinLpLockbox.sol:22–38); the operator can only add liquidity and collect fees capped at 240,000 LEMNI / 2,400 USDC per 24 h, paid to the Safe (immutables, lines 152–157).

Removing this liquidity before 2027-10-07 therefore has **no code path**; removing it afterwards requires two of three multisig signers *and* a valid Falcon-512 post-quantum signature, and pays out only to the multisig. A conventional token whose LP tokens sit in a deployer wallet requires exactly one ECDSA key for the same action. On this axis LEMNI is gated more strongly than the standard token designs that carry no flag.

Supply cannot be inflated at will.

- totalSupply has read exactly 10,500,000 LEMNI at every one of 101 evenly spaced reads spanning the contract's entire life (block 83,457,504 → 95,237,490).
- Minting is hard-capped in code (MAX_SUPPLY check, diamond/facets/GovernanceFacet.sol:259) and executes only with the 2-of-3 multisig *plus* a Falcon-512 governance signature over the exact action payload with a consumed nonce (diamond/libraries/LibLemnicoin.sol:710–742).

Holders' balances cannot be taken.

- No deployed function transfers another holder's tokens. Burns affect only the caller's own balance (facets/QuantumBurnFacet.sol:79, 119) — exactly as the application prescribes: the burn alternative "...prevents an attacker who has obtained the wallet private key from maliciously destroying a user's tokens" (Patent Text, p. 8, ll. 5–7).
- There is no blacklist: no function in the source or the deployed selector set bans an address. The whitelists are strictly additive permissions for holders who opted into quantum protection, and carry permanent guardian "fuses" that switch the features off, not on (diamond/facets/WalletWhitelistFacet.sol:87–94, GovernanceFacet.sol:579–592).
- There is no transfer tax and no fee escalation lever on ERC-20 transfers: LibERC20._transfer moves the exact amount (diamond/libraries/LibERC20.sol:26–42). A native-POL fee exists only on the optional quantum-signed paths, priced 12–40 basis points by an oracle whose replacement requires a post-quantum-signed governance action, guardian approval and a 1-day delay.
- No selfdestruct exists anywhere in the repository.
- Every supply, pause, oracle, key-rotation and upgrade-authority operation is gated behind the 2-of-3 multisig plus the Falcon-512 post-quantum governance signature (Appendix B), with public events and continuous monitoring — a control layer standard Ownable-pattern tokens do not have.

Governance key custody: the owner is the 2-of-3 Gnosis Safe v1.4.1 with no modules enabled and a zero guard slot; ownership is held, not hidden.

How that code comports to the patent. The patent’s whole object is to make theft impossible even for an attacker holding the controlling private key — “wallet key theft alone is no longer sufficient to move, approve, or destroy tokens” (Patent Text, p. 7, ll. 12–13). The deployment applies that same discipline to the project’s own authority: the identical post-quantum signature class the patent prescribes for holder protection (Claims 4–5) also gates every governance operation and the liquidity lockbox’s withdrawal path. The design does not reserve a back door for its operators; it points the patented lock at them too.

Why the scanner misreads it. Rug-pull heuristics count privileged functions and infer capability from their existence. LEMNI’s privileged functions exist — and each is bounded in code and dual-gated behind a post-quantum signature, an arrangement no conventional token presents and the heuristic has no category for. The scanner tallies the surface and never reads the bounds; what it reports as rug-pull capability is, on inspection, the strongest set of anti-rug-pull constraints in the comparison class.

Conclusion: there is no mechanism by which holders’ funds or pool liquidity can be taken.
“Rugpull risk” is false.

7. “Known malicious” — false

Where the on-chain record proves it false. “Known malicious” asserts an established fact. There is no instance: no exploit, no security incident, no victim report — not in the public record, not in the project’s repositories, not in web search on the contract address (the only third-party page surfaced is a neutral market-data listing). The contract has operated continuously since 2026-02-25 15:55:36 UTC (deployment tx 0xf470b78032cfe3f51c019a2db4675cf9b3afdfbadbf6e2de248f5c7e40d5c2b7) — 226 days at the date of this document — with supply constant at every sampled point and the token’s protective mechanisms operating in production (e.g. dual post-quantum-signed transfer at block 86,202,341).

How the code comports to the patent. The design’s purpose is the opposite of malice: it is a filed patent’s method for ensuring “that wallet key theft alone is no longer sufficient to move, approve, or destroy tokens” (Patent Text, p. 7, ll. 12–13) and for enabling the lawful owner to recover tokens from a stolen wallet (Claim 13; Patent Text, p. 19, ll. 7–11).

Why the scanner misreads it. Since no incident exists, the “Known malicious” output cannot rest on anything known; it can only be an inference propagated from the same pattern-match that produced the other designations — the novel, beneficially functional code shapes examined in §§5 and 8 — and then published under a label that asserts established fact.

Conclusion: a “Known malicious” designation with no known incident, exploit, or victim is false by its own terms.

8. “Resembles malicious contract” — false

Where the deployed code proves it false. The “malicious contract” this one supposedly resembles is a honeypot — a token whose holders can buy but not sell. The deployed code forecloses that reading at every point:

- **The flag is set only by the holder’s own transaction.** Every key-registration function writes only `msg.sender’s state (diamond/facets/KeyManagementFacet.sol:123, 151, 201, 237, 266–280)` — no third party, the owner included, can register keys for someone else’s address, and no code path ever clears a registration. The restriction is a holder-elected second factor on that holder’s own outflows.
- **Unregistered holders are standard ERC-20.** For a sender without registered keys, `transfer()` executes a plain balance move with no fee and no recipient restriction (`TokenFacet.sol:93–96; LibERC20.sol:26–42`); `approve()` and `transferFrom()` likewise (`TokenFacet.sol:135–138, 171–175`). Any such holder can sell into the pool at any time; pool-outbound transfers on buys are expressly allowed (line 116). Buys and sells both function.
- **Registered holders can also sell at any time.** The quantum-signed approval `quantumApproveDual` (`QuantumTransferFacet.sol:147–175`) authorizes a router, whose `transferFrom` then passes both gates — approved spender (lines 180–184) and whitelisted-pool recipient (lines 186–191). The active pool’s whitelist flag reads true on-chain. If the attestation service ever goes silent, `quantumTransferWithHolderKey` (lines 80–106) opens automatically after the configured outage window — holders are never stranded. The only difference from a plain sale is that the instruction carries the holder’s post-quantum signature.
- The contract is built to simulate cleanly: zero-amount transfers succeed and emit standard events precisely so wallet pre-flight simulations complete (`TokenFacet.sol:74–81, 156–160`).

How that code comports to the patent. The gated branch is the verification step (b) of Claim 1; the holder-only registration call is the “function call from the wallet’s address” of Claim 11 (γ4); and the revert itself is prescribed verbatim — “All other calls revert and direct the caller to the ownership-key-protected alternative” (Patent Text, p. 7, ll. 30–31).

Why the scanner misreads it. Honeypot heuristics test a single bit — can `transfer()` revert for a holder? — and cannot distinguish a holder-elected second factor from a seller block; the two produce the same surface shape. LEMNI is the first deployment in this category, so the heuristic has never seen the benign cause of that shape. It reads a theft-prevention gate, prescribed word-for-word by a patent application, as its malicious look-alike.

Conclusion: the contract LEMNI “resembles” is the one claimed in EP 26300368.3 / U.S. 19/745,426 — a protection mechanism, not a honeypot. No holder is, or can be, blocked from selling. “Resembles malicious contract” is false.

9. “Possible price manipulation” — false

Where the deployed code proves it false. Every lever a manipulator would need is absent or closed:

- Price formation is entirely external to the contract: one public Uniswap V3 pool (LEMNI/USDC 1%, 0x28289c9D...), constant-liquidity curve, public swaps. The token code contains no price logic on transfers.
- No supply-side lever: no tax, no rebase, exact-amount transfers (`LibERC20.sol:26–42`); the only supply-changing functions are the dual-gated, hard-capped mint (§6) — with supply in fact constant across the contract’s life — and self-burns.
- No liquidity-side lever: all positions locked (§6); the lockbox operator can add liquidity but remove none; fee collection is capped per 24 h and pays the multisig.
- The volume circuit breaker (999,999 LEMNI / 24 h, live) halts — and cannot execute — large-scale transfers (`LibERC20.sol:141–165`).
- The price oracle prices only the native-POL protocol fee on quantum paths; it has no connection to pool pricing or transfer amounts (`LibLemnecoin.sol:160–193`).

How that code comports to the patent. The patent deliberately leaves market mechanics untouched: tokens protected by the method “are listed, traded, and tracked exactly like any other token on the same blockchain” (Patent Text, p. 15, ll. 24–26), with balances on the standard interface (p. 15, ll. 11–13). The invention operates on who may authorize a movement, never on price.

Why the scanner misreads it. The scanner sees an oracle and governance whitelists next to a transfer gate and infers market control — the pattern such parts form in manipulative tokens. Here the code shows the oracle pricing only a protocol fee, the whitelists only adding permissions, and the one novel element — the ownership-key gate — touching authorization, not price. The parts are present for a beneficially functional reason the heuristic’s training never contained.

Conclusion: the contract contains no mechanism by which its operators can move the market price. “Possible price manipulation” is false as a statement about this contract.

10. How to verify (minutes, not days)

Every figure above is reproducible with public infrastructure:

- **Source and bytecode.** `git clone https://github.com/rmvince/lemnicoin-contracts`; compile with `solc 0.8.24/0.8.25`, `optimizer runs = 1`, via-IR, EVM “cancun”; compare runtime bytecode against `eth_getCode` at the facet addresses in §3 (strip the trailing CBOR metadata block).
- **Interface.** Call `facets()` (selector `0x7a0ed627`) on the diamond; hash the repository ABIs; every selector resolves.
- **Transfer behavior.** Simulate `transfer()` from any unregistered holder: it succeeds. Check `isWhitelistedDEXPool(0x28289c9D...) → true`.
- **Liquidity custody.** On the Uniswap position manager (`0xC36442b4...`): `balanceOf(lockbox) = 5`; `tokenOfOwnerByIndex` yields the five token IDs; on the lockbox: `unlockTime()`, `ecdsaFallbackTime()`, `falconActive()`, `owner()` as cited in §6.
- **Governance state.** `paused() = false`; `governanceFalconInitialized() = true`; `circuitBreakerThreshold()/Window()` as cited; owner’s Safe: `getOwners()`, `getThreshold()`, `getModulesPaginated()`.
- **Patent.** EP 26300368.3 (filed 10 June 2026) and U.S. App. No. 19/745,426; the application text quoted here is available in full from the applicant and, upon publication, from the EPO/USPTO registers.

11. Requested action

Each of the five designations is demonstrated false by public source code, live chain state, and a filed patent application — evidence checkable in minutes with the procedures in §10. We request that Blockaid remove the five designations (“Suspicious code”, “Rugpull risk”, “Known malicious”, “Resembles malicious contract”, “Possible price manipulation”) and the resulting “Malicious” label for contract `0x02011c7B7D9C6dfc563c92197557e8BE490a85b4` on Polygon, and that Consensus cease republishing them in MetaMask. The project will promptly answer any technical question and provide any further material needed for re-review, including the full patent application text.

Appendix A — The patented protection method, in the application’s own words

LEMNI implements the method of the Patent Text (EP 26300368.3, filed 10 June 2026; U.S. App. No. 19/745,426): every wallet may be bound to a second key pair — the *ownership keys* — distinct from the wallet’s ECDSA keys, and the token’s standard interface functions are overridden so that, for wallets that have opted in, no token movement can occur without cryptographic verification against the ownership keys. In the application’s own words:

“...the standard token interface functions — that is, the functions that every token contract exposes for transferring, approving, and destroying tokens — are overridden so that they reject all general-purpose calls. In their place, the smart contract exposes alternative functions that require cryptographic verification using the ownership keys before any token movement can occur. This overriding mechanism is the core of the present invention and ensures that wallet key theft alone is no longer sufficient to move, approve, or destroy tokens.” — Patent Text, p. 7, ll. 7–13

“...the ownership keys allow to perform a transaction more securely than in the prior art without requiring a new blockchain, forks, new wallet formats or protocol changes. The method is applicable to existing blockchains...” — Patent Text, p. 6, ll. 3–6

“According to an advantageous embodiment, the ownership keys are quantum-resistant. Thus, the method is really secured because, even if the wallet private key has been determined by a malicious person using quantum technics, said person will not be able to determine the ownership private key... According to a specific embodiment, the ownership keys comply with Lemniscate AGM Isogeny, Dilithium, Winternitz One-Time Signature or SPHINCS+ schemes.” — Patent Text, p. 6, ll. 9–15

“...the invention is platform-agnostic and applies wherever a smart contract or equivalent programmable execution environment can override or wrap standard token transfer functions, store cryptographic commitments in contract state, and verify cryptographic proofs during transaction execution.” — Patent Text, p. 14, ll. 27–30

“The invention permits the token to reside on standard, conventional blockchains in standard wallets. Token balances remain queryable via the standard interface — for example, `balanceOf()` for ERC-20 — such that any wallet, block explorer, or portfolio tracker can read them without modification.” — Patent Text, p. 15, ll. 10–13

The protection is **opt-in and holder-elected**: a wallet that never registers ownership keys transacts as an ordinary ERC-20 holder (§8). What a scanner sees as a “restricted transfer” is the verification step (b) of Claim 1, applied exactly as the patent prescribes.

Appendix B — Privileged functions and their bounds

Complete enumeration of gated functions in the deployed selector set. “Owner” is the 2-of-3 Gnosis Safe; “+Falcon” means a Falcon-512 post-quantum governance signature over the exact action payload, with a consumed nonce (diamond/libraries/LibLemnicoi.sol:710–742), is additionally required.

Function (repository citation)	Gate	What it cannot do
diamondCut — diamond/facets/DiamondCutFacet.sol:11–18	owner	Act without 2-of-3 signatures or without emitting a public DiamondCut event (independently monitored at 15 s intervals).
diamondCutWithFalcon / transferOwnershipWithFalcon — diamond/facets/FalconGovernedCutFacet.sol:44–86	owner +Falcon	Execute with ECDSA keys alone; transfer ownership to the zero address.
mint — GovernanceFacet.sol:251–261	owner +Falcon	Exceed MAX_SUPPLY (LibAppStorage.sol:361); supply unchanged at every sampled point of the contract’s life.
pause / unpause — GovernanceFacet.sol:263–281	owner +Falcon	Move, take or burn any balance — a pause only halts movement (LibERC20.sol:107).
setCircuitBreakerParams / Exclusions — GovernanceFacet.sol:204–245	owner +Falcon	Set a window under 600 s when enabled; move funds.
withdrawETH — GovernanceFacet.sol:295–305	owner +Falcon	Touch any LEMNI balance (sweeps only native POL to the owner).
Key/role rotation (attestor, verifier, guardian, rekey operator, feature flags) — GovernanceFacet.sol:96–191, 283–293, 312–323	owner (or rekey operator) +Falcon	Move funds; attestor-only transfer paths are permanently closed in code regardless of attestor key (LibLemnicoi.sol:440–442).
Oracle replacement / disable — GovernanceFacet.sol:329–412	owner +Falcon, guardian approval, 1- day delay	Impose any fee on plain ERC-20 transfers.
revokeQuantumApproval — GovernanceFacet.sol:423–436	owner +Falcon	Transfer any balance (clears an approval flag only).
CEX/DEX whitelist edits; CEX bypass toggle — GovernanceFacet.sol:442–592	governance signer +Falcon (+ guardian, 1- day delay)	Restrict any holder who has not registered quantum keys; survive the permanent guardian fuse.

Function (repository citation)	Gate	What it cannot do
setFeeExempt — diamond/facets/FeeExemptionFacet.sol:31-38	owner	Charge or block anyone (waives fees only).
Wallet-whitelist pair management — diamond/facets/WalletWhitelistFacet.sol:33-94	owner / guardian	Block any transfer otherwise allowed; affect unregistered holders (additive permissions only; permanent off-fuse).
Time locks — diamond/facets/TimeLockFacet.sol:54-131	owner	Lock tokens the owner does not hold; reclaim anything but a still-locked grant the owner itself funded.
Vesting / deployment-phase distribution — diamond/facets/VestingFacet.sol:40-121	owner	Spend any other holder's balance; shorten nothing — cliff amendments extend only, lock removal releases only.
registerQuantumKeysTrusted — KeyManagementFacet.sol:189-219	owner	Register or alter keys for any address other than itself (writes msg.sender only).
Attestor liveness configuration — diamond/facets/AttestorLivenessFacet.sol:42-75	owner / heartbeat signer	Move funds; outage window bounded to 900-1,296,000 blocks (or disabled).
Ring / V3 policy configuration — diamond/facets/RingFacet.sol:88-94; LAIPV3TransferFacet.sol:65-104	owner	Move funds; affect unregistered holders; ring-bypass delay bounded to 1-30 days.

Appendix C — Deployed facets (live loupe read)

24 facets, 209 selectors, read from `facets()` on 2026-10-09. Grouped by function:

- **ERC-20 core:** `TokenFacet` — `name`, `symbol`, `decimals`, `totalSupply`, `balanceOf`, `allowance`, `transfer`, `approve`, `transferFrom`, `isQuantumApprovedSpender`.
- **EIP-2535 infrastructure:** `DiamondCutFacet` (`diamondCut`); `DiamondLoupeFacet` (`facets`, `facetAddress(es)`, `facetFunctionSelectors`, `supportsInterface`); `OwnershipFacet` (`owner`, `transferOwnership`); `FalconGovernedCutFacet` (`diamondCutWithFalcon`, `transferOwnershipWithFalcon`).
- **Governance:** `GovernanceFacet` — the dual-gated operations of Appendix B plus read-only views (`paused`, `guardian`, `circuit-breaker` views, `governance nonces`, `whitelist` views).
- **Key management:** `KeyManagementFacet` — `createBindingChallenge`, `bindQuantumKeys`, `registerQuantumKeys(...WithValidation/...Trusted/...OffChain)`, `registerHolderFalconKey`, `registerQuantumKeysV2`, `validateLaipCurve`, `commitment` views (all write `msg.sender` only).
- **Quantum transfer paths:** `QuantumTransferFacet` — `quantumTransfer`, `quantumTransferDual`, `quantumTransferWithHolderKey`, `quantumApprove(Dual)`, `transferWithFalconEnvelope`, `digest/nonce` views; `QuantumBurnFacet` — `quantumBurn` (self only).
- **Recovery and sessions:** `RecoveryFacet` — `recoveryTransfer(Dual)`; `SessionFacet` — time-boxed holder-signed CEX sessions.
- **Distribution controls:** `VestingFacet` (three single-purpose deployments), `TimeLockFacet`, `WalletWhitelistFacet`, `FeeExemptionFacet`.
- **Attestor liveness:** `AttestorLivenessFacet` — heartbeat, outage-gate configuration and views.
- **Ring and V3 paths:** `RingFacet`, `RingTransferFacet`, `LAIPV3Facet`, `LAIPV3TransferFacet`, `LAIPV3CapabilitiesFacet`, `RingV3TransferFacet` — multi-key ring transfers, V3 registration/migration, V3 transfer/approve/burn/recovery/sessions, SLH-DSA policy.

Generation record: source repository `rmvince/lemnicoin-contracts` at commit `704000e819acc167c1b62acd6f4ca37ed7965ede`; on-chain reads 2026-10-09, Polygon head block 95,237,490, via public JSON-RPC (`polygon-bor-rpc.publicnode.com`, `polygon.drpc.org`); compiler `solc 0.8.24/0.8.25`, optimizer runs = 1, via-IR, EVM target `cancun`; patent passages quoted verbatim from the application text “Method for performing a transaction on a blockchain” (EP 26300368.3 / U.S. 19/745,426).