ZENITH STUDIO

PQC CHAIN-AGNOSTIC LAYER

SECURING WEB3 INFRA FROM POST-QUANTUM ERA

ABSTRACT

Zenith PQC LAYER is a middleware layer providing post-quantum cryptography (PQC) signing and verification for blockchain transactions, independent of the underlying chain.

It enables wallets and smart contracts to use quantum-resistant signatures (e.g. Dilithium, Falcon, SPHINCS+) while preserving compatibility with any L1/L2. In effect, Zenith intercepts or augments normal transaction flows to add a PQC layer without forking the blockchain.

This deck outlines Zenith's architecture, SDK, supported algorithms, and use cases, combining deep technical detail showing how Zenith addresses the incoming quantum threat (Shor's algorithm) by standardising an on-chain PQ key registry and off-chain signing library, and how it fits into existing ecosystems.

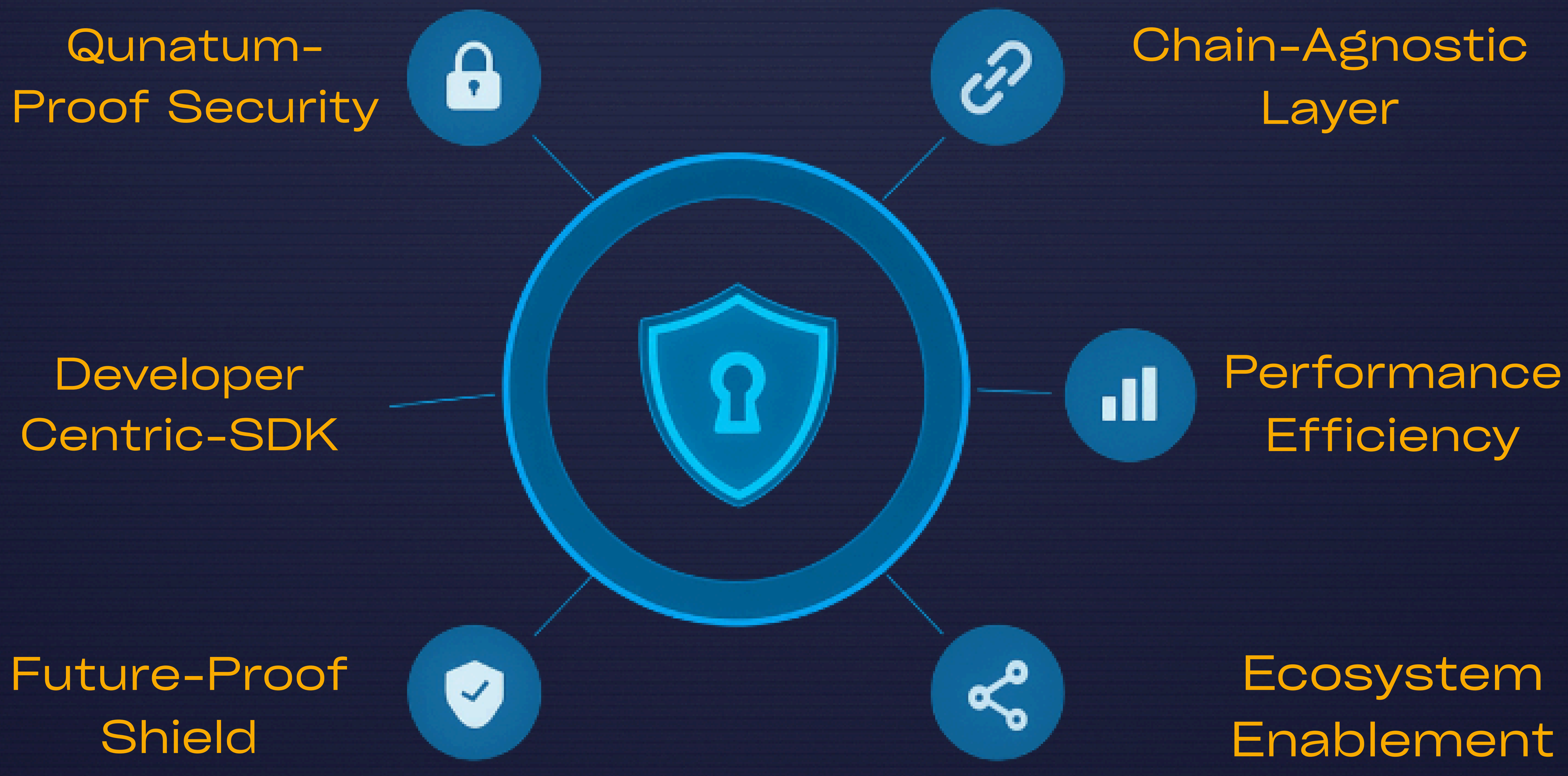
INTRODUCTION

Zenith is a developer-focused, chain-agnostic PQC framework . It provides APIs (signPQC, verifyPQC, etc.) and smart-contract tools so that any dApp or wallet can switch on quantum-safe signatures.
Unlike chain-specific solutions, Zenith works with Ethereum, Cosmos, Solana, etc. by handling signature and call-data formatting externally.

The goal is to give Web3 projects a migration path before “Q-Day” (when quantum computers break ECC). Zenith’s angle is simplicity and broad compatibility: developers just import Zenith SDK, and it will detect the chain’s hash method (Keccak vs SHA3) and apply the correct PQC scheme under the hood.

By using well-vetted NIST/PQC algorithms and offering a public key registry, Zenith lets teams focus on building features while quantum-proofing crypto signing.

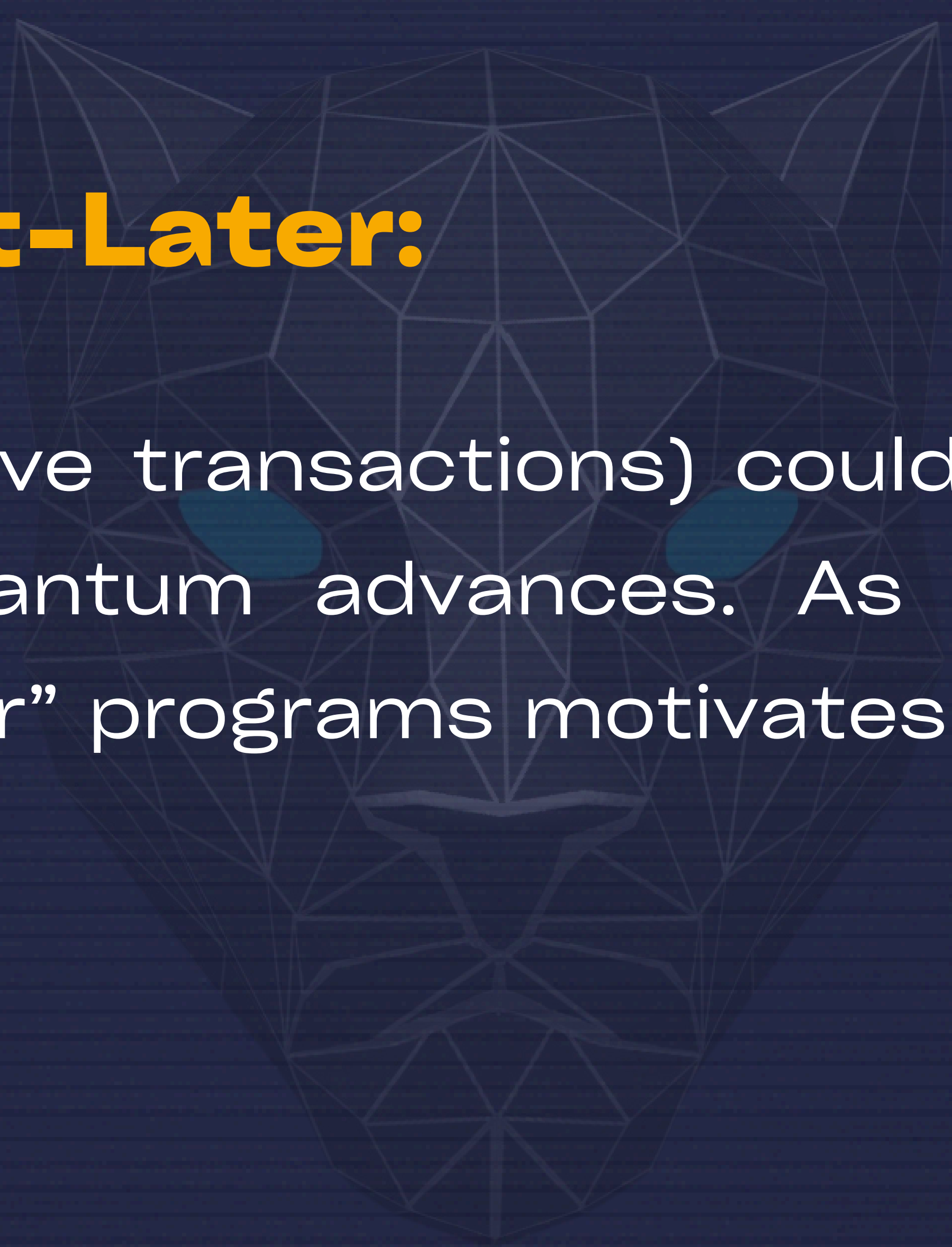
Goals of ZENITH PQC Layer



WHY QUANTUM-SAFE WEB3 MATTERS ?

Quantum Threat:

Current crypto (ECDSA, EdDSA) rely on elliptic-curve discrete log. Shor's algorithm can break these once large-scale quantum computers exist. Analysts now predict RSA-2048 (and by extension ECC256) could fall by ~2030 . The timeline has shifted from "if" to "when".



Store-Now Decrypt-Later:

Data (private keys, sensitive transactions) could be harvested today and decrypted later once quantum advances. As noted, the existence of "harvest now, decrypt later" programs motivates adopting PQC early.

NIST & Industry:

NIST finalized its first PQC standards (KYBER, Dilithium, Falcon, SPHINCS+, etc.) in 2024, and major projects (Algorand, Google, etc.) are integrating PQC. Zenith leverages these vetted algorithms to meet the growing demand.

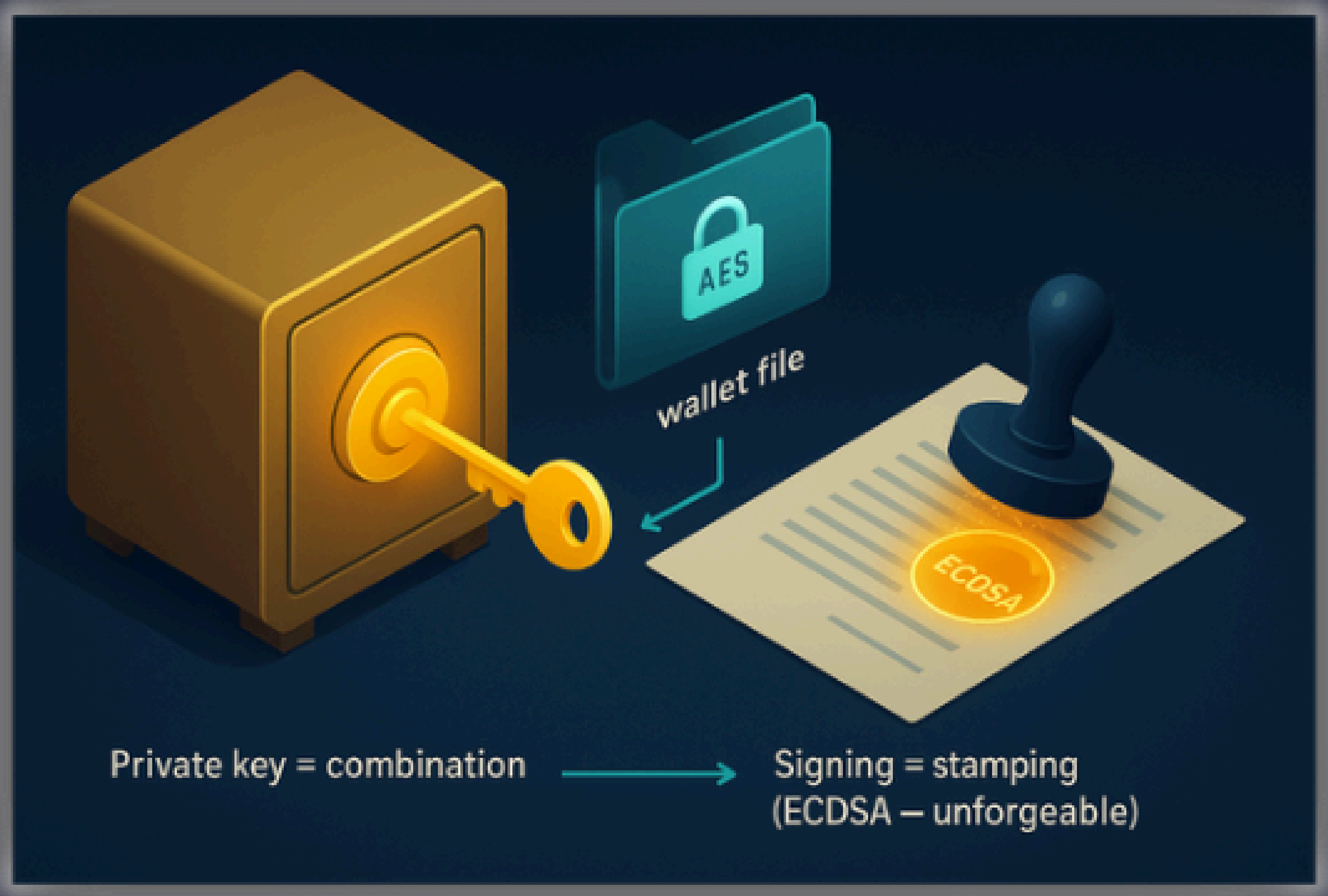
Mosca's Theorem:

Organizations must start migrating now to avoid a "Y2Q" crisis, according to Mosca's risk framework. Zenith offers the necessary tools for an orderly migration: protected wallets, PQC signatures on-chain, and governance fallback if emergency migration (fork) is needed.

Old-Encryption Method

Web3 Wallets (like metamask , trust wallet etc) uses browser storage to store the private key of the user when they set-up their wallet. The encryption method that is used for storing the private key or when the user sets up their own wallet password is PBKDF2-derived key + AES-GCM.

PBKDF2(Password-Based Key Derivation Function 2) + AES-GCM is widely used in wallets,password managers, and secure vaults. It converts human-readable passwordinto a cryptographically strong key.



How its is being used ?

1. It provides salting (to prevent from rainbow attack - a table of hash values to reverse engineer plain passwords)

2. Provide a specific number of iteration as input - the iterations increase the computational cost, making brute-force attacks more difficult.

3. A hash function (SHA-256) - cryptographic hash functions that will take the plain input password and convert it into 256-bits hash . Its extremely difficult to reverse engineer the hash with the normal computer but its very easy for a quantum computer to break it within seconds .

QUANTUM THREAT TO WALLETS AND BLOCKCHAINS

How Crypto Wallets Work

Wallets hold private keys (secret numbers) encrypted by your password. The private key is used to sign transactions (prove you own coins). Transaction signing uses public-key cryptography (e.g. ECDSA on secp256k1 for Bitcoin/Ethereum addresses). Encryption of the wallet file (usually AES-256) protects the key with your password, but the main protection is in the key's math lock.

A crypto wallet works like a safe: the private key is the combination, and signing a transaction is stamping with it. Files may be AES-encrypted, but real security relies on cryptography (ECDSA), which prevents signature forgery without the private key.

Quantum computers using Shor's algorithm can break elliptic-curve cryptography, exposing private keys once public keys appear on-chain. This threatens all classical wallets (MetaMask, Ledger, hardware). Cold storage offers no safety if keys were ever used. Additionally, Grover's algorithm boosts quantum miners, enabling 51% attacks in Proof-of-Work and forged votes in Proof-of-Stake.

Right now, your wallet's lock (ECDSA) is considered unbreakable by classical computers. However, a quantum "super-computer" could systematically break these locks. It's like having a super lock-picker that can try all combinations in seconds . Every used address (public key) becomes a beacon to the thief. Even mining can be attacked: a quantum rig would solve the mining puzzle much faster, letting one miner dominate . In short, current wallets are not future-proof.

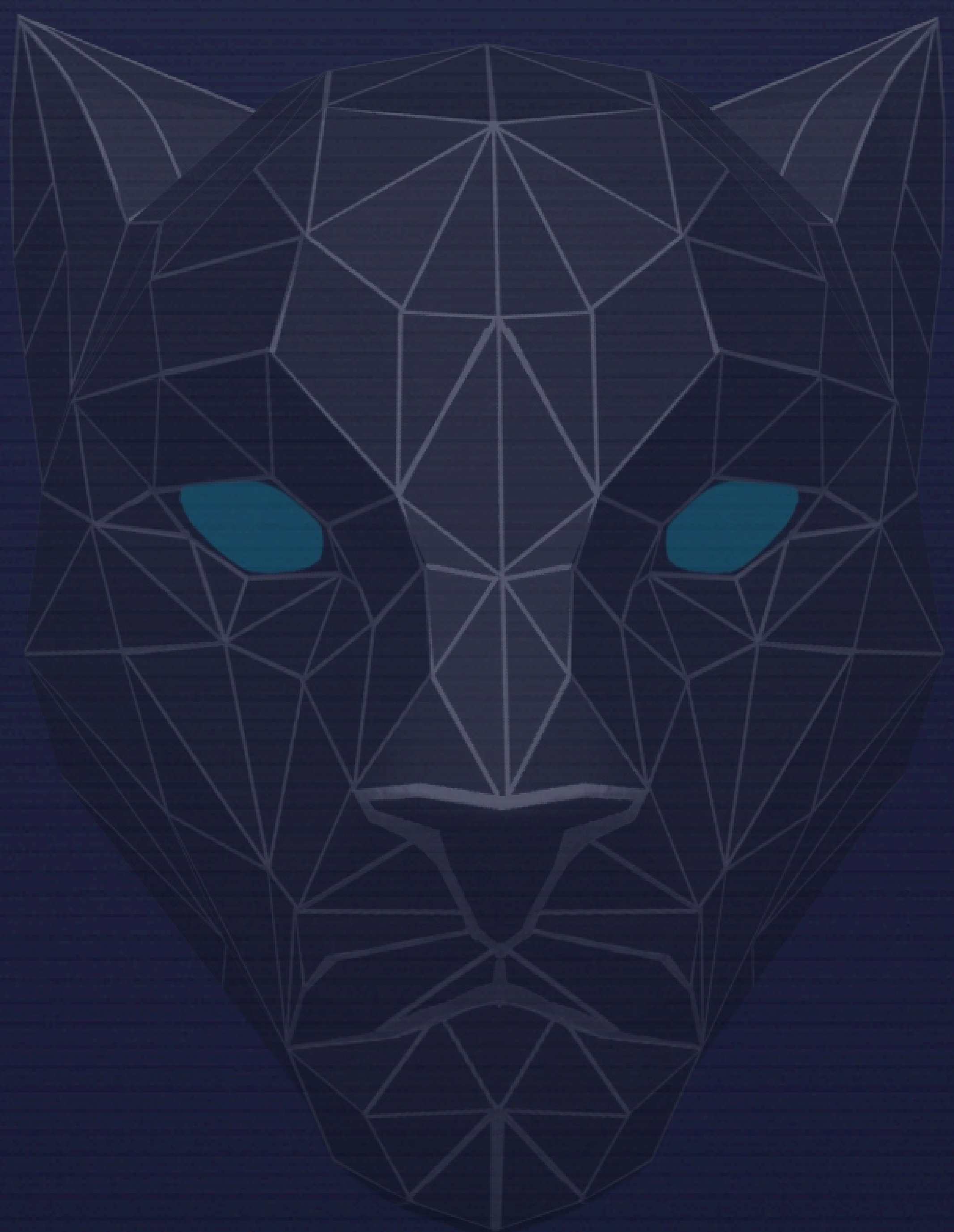
Scale of the Risk

Hundreds of Millions of Wallets: Chain-analysis reports >400 million cryptocurrency wallets with balances exist today . Almost all rely on vulnerable ECC/ECDSA.

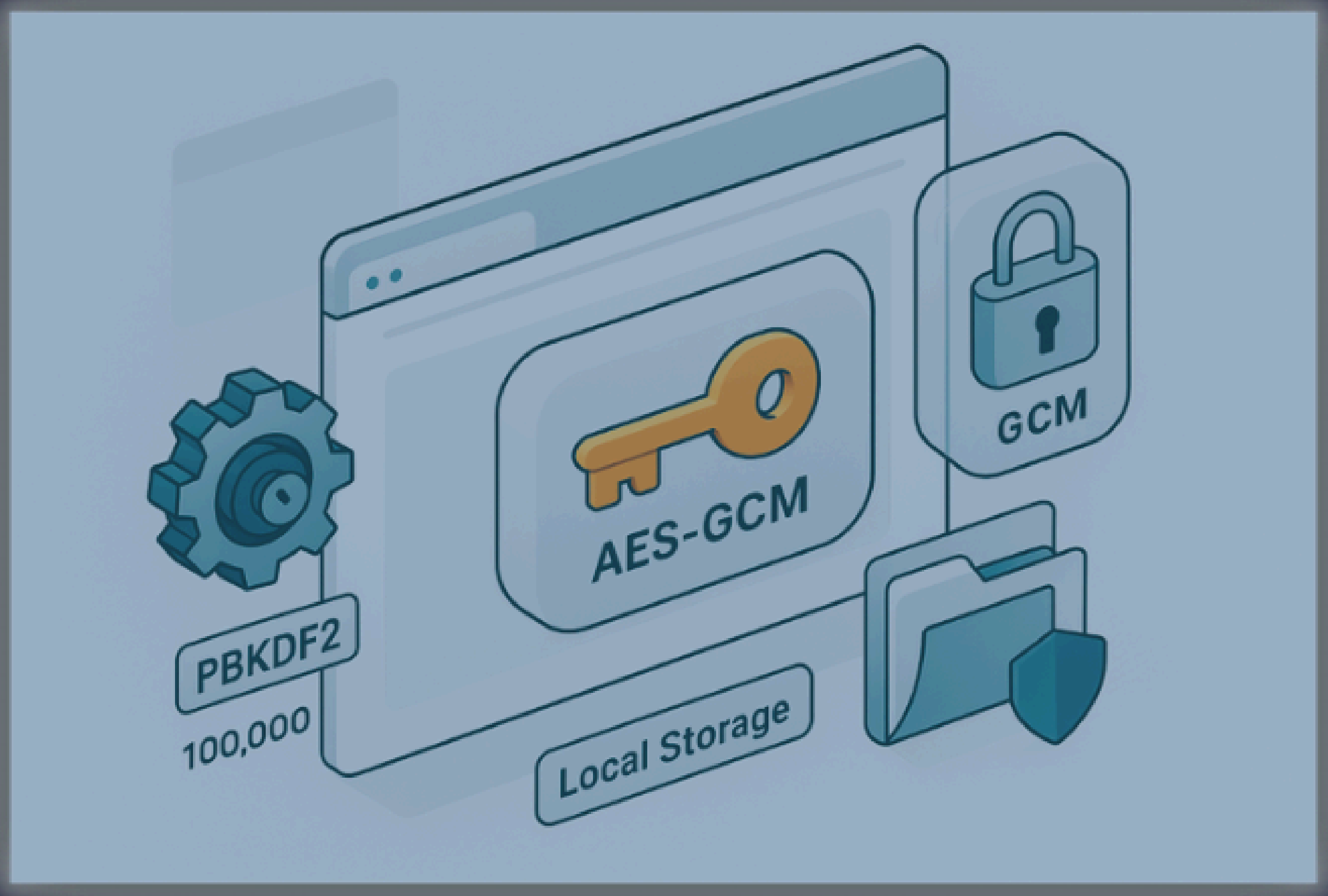
Example - Bitcoin Exposure: Over 10 million Bitcoin addresses with funds have exposed public keys, representing ~$648 billion of BTC at risk. All At Risk in Theory: That's a huge portion of the crypto economy. An attacker with a working quantum computer could, in principle, break any of these wallets.

To appreciate the scale: there are well over 400 million active crypto wallets (with non-zero balances).Project Eleven (a PQC startup) notes that just Bitcoin has >10 million UTXO addresses with public keys exposed.

In practice this means billions of dollars of crypto could be vulnerable to future quantum theft. The market is huge, and almost no wallet is currently safe from quantum attacks.



PROBLEMS WITH CURRENT SIGNATURE SCHEMES (ECDSA, EDDSA)



Quantum Vulnerability:

ECC-based signatures are not quantum-safe. A large quantum (with ~1 million qubits) can run Shor’s algorithm to recover private keys.

Single Point of Failure:

Popular chains use one signature method (e.g. Ethereum/BN256 or Ed25519). A future quantum break would compromise all funds unless preemptively migrated.

Key Sizes & Usage:

ECDSA/EdDSA signatures are small (64 bytes) and fast, but have no fallback. They also lack flexible post-compromise features like key rotation or social recovery.

Regulatory/Compliance:

Some sectors (finance, government) already plan for post-quantum compliance. Web3 must align, or risk obsolescence.

ZENITH'S SOLUTION

Zenith introduces a transparent PQC signing overlay: developers integrate Zenith's SDK into wallets and services so that every transaction is signed twice (once normally, once with PQC) or with a hybrid scheme. There is no need to change consensus or transaction format; Zenith simply appends/combines PQC signature data according to a standard format.

KEY FEATURES

Overlay Design:

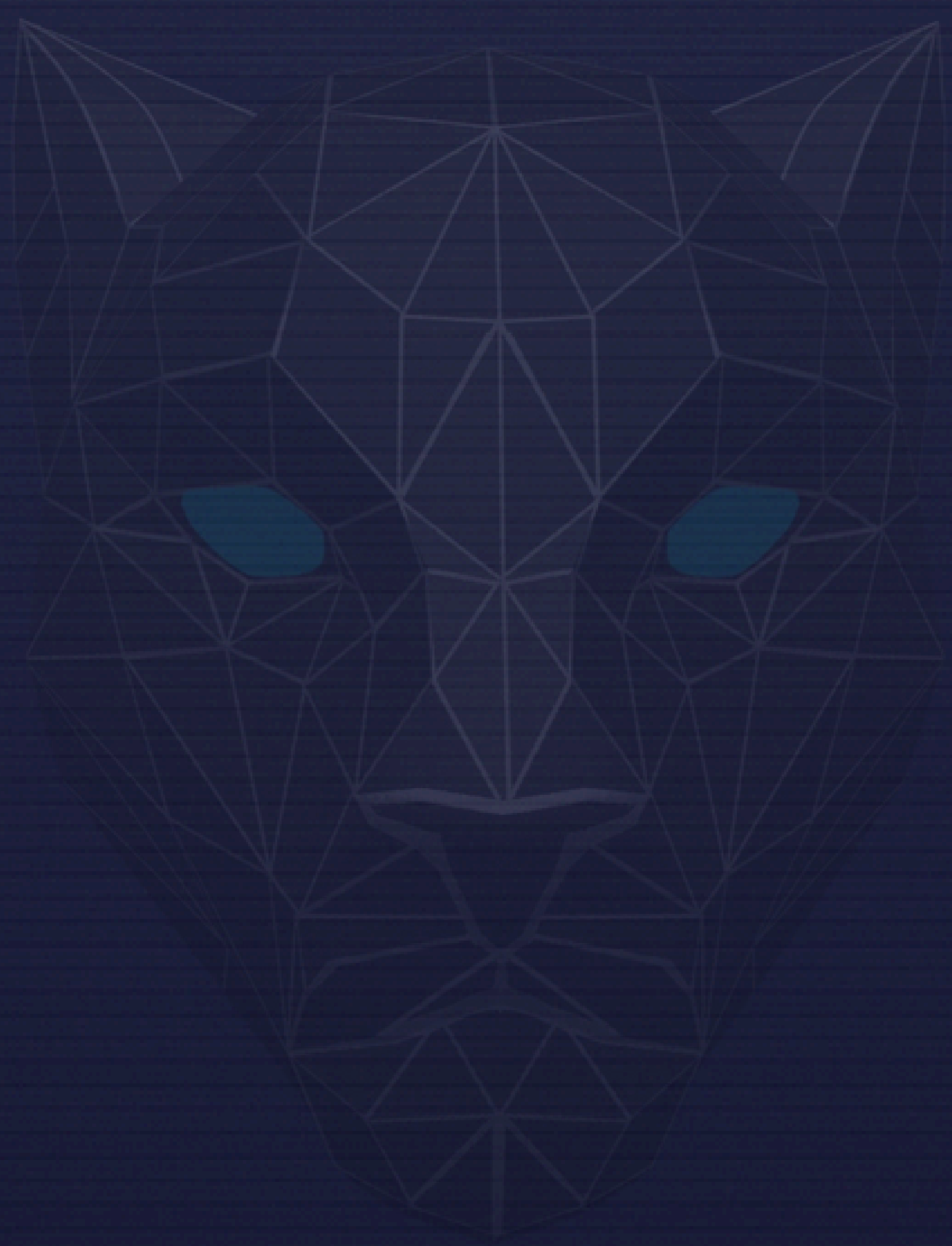
Zenith hooks into existing signing calls (signTransaction, signTypedData) and outputs a special payload including the PQ signature. Smart contracts can then verify PQC signatures via standard calls or precompiles.

Chain-Agnostic

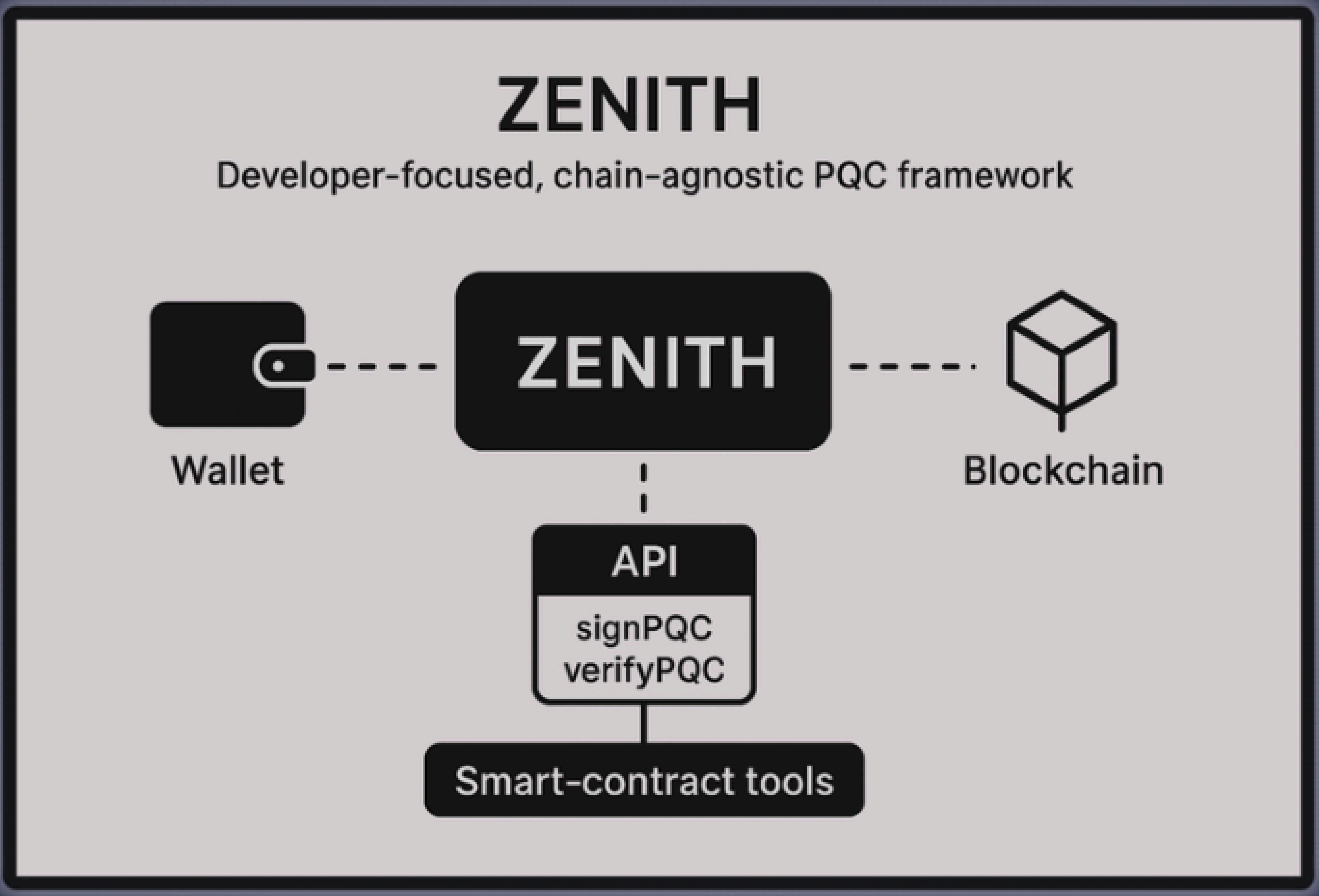
Works on EVM (Ethereum, BSC, etc.), Cosmos (Tendermint), Solana, etc. The SDK abstracts the hashing differences

Open Source & Pluggable

Built on liboqs and open standards, so any algorithm (current or future) can be added. SDKs in TypeScript, Rust, and WASM allow easy binding.



System Architecture (Wallet ↔ PQC Layer ↔ Chain



Zenith PQC LAYER as middleware :

A wallet sends raw calldata to Zenith instead of the node. The PQC Layer hashes it (Keccak256/SHA-3), signs with signPQC, and appends the PQ signature. The blockchain verifies it via Zenith's registry smart contract. If valid, the transaction executes; if invalid, it reverts.

SUPPORTED ALGORITHMS: DILITHIUM, FALCON, SPHINCS+ (VIA LIBOQS)

Zenith supports the leading NIST-PQC signature schemes through liboqs:

CRYSTALS-Dilithium:

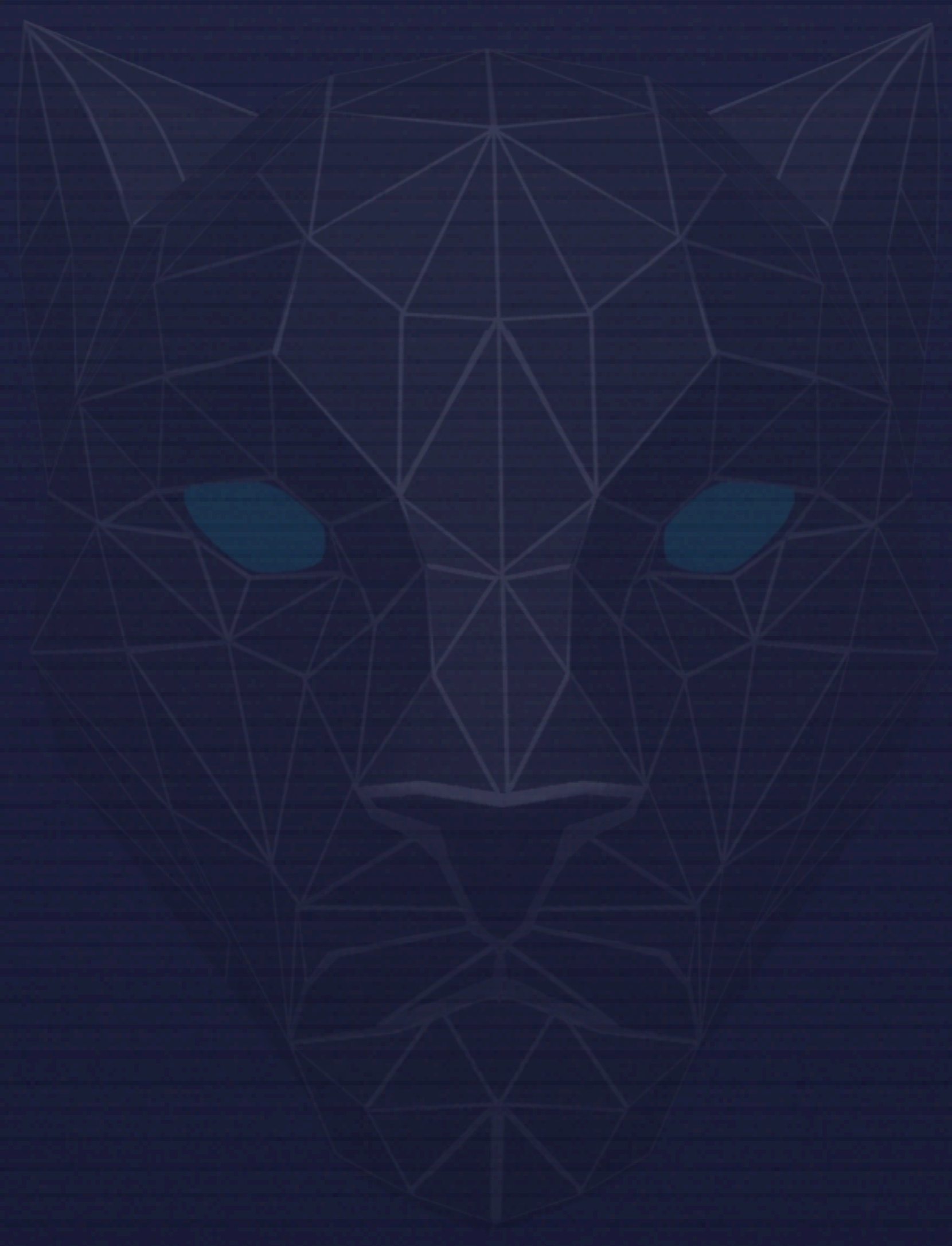
A lattice-based signature. Offers strong security with moderate signature sizes (e.g. ~1320 bytes for Dilithium2). Fast verification, suitable for most use cases.

Falcon:

Another lattice (NTRU) signature with compact size: Falcon-512 signatures are ~897 bytes. Smaller keys/signatures but more compute to sign, making it good when bandwidth is tight.

SPHINCS+:

Zenith's SDK supports per-key algorithm choice and hybrid signing. Developers use unified Zen APIs via liboqs, ensuring automatic updates as NIST evolves. Falcon suits EVM gas limits, while off-chain messaging supports broader schemes.



WASM, RUST, AND NATIVE BINDING LAYERS

Zenith’s core crypto operations are implemented in Rust using liboqs, then compiled to different targets.

WebAssembly (WASM):

For browser and Node environments. The WASM module is wrapped by a TypeScript interface. This lets Web3 front-ends (dApps, browser wallets) perform PQC signing without trust issues.

Rust/Native:

For back-end or CLI tools, native Rust crates (oqs-sys/oqs) provide fast, safe APIs. Servers or desktop wallets can use these directly.

Language Bindings:

The oqs Rust crate offers safe wrappers. In TypeScript, the WASM functions are exposed via npm packages. In Solidity, public keys are simply bytes storage.

The multi-layer approach means Zenith works anywhere: a React dApp (WASM), a Node script, or a Rust program all use the same underlying PQ algorithms.

ON-CHAIN KEY REGISTRY (SOLIDITY EXAMPLES)

Zenith uses a smart contract registry to map user accounts to their PQ public keys or hashes. This can be a simple Solidity contract.

```
contract PQKeyRegistry {
    mapping(address => bytes32) public commitments; // or mapping to bytes public keys
    function registerKey(bytes32 commitment) external {
        require(commitments[msg.sender] == 0, "Already set");
        commitments[msg.sender] = commitment;
    }
}
```

A user first calls registerKey(sha256(PublicKey)), committing their PQ public key or its hash on-chain. This pattern mirrors Ethereum Magicians' proposal for post-quantum account recovery.

Once stored, any smart contract can fetch and use commitments[user] to verify a PQ signature (by checking a preimage or using a zk-proof). Zenith provides examples of such Solidity code. In practice, the registry should be governed (upgradable or multisig) so that keys can be revoked or format-upgraded via DAO governance.

CALLDATA FORMATTING STANDARDS

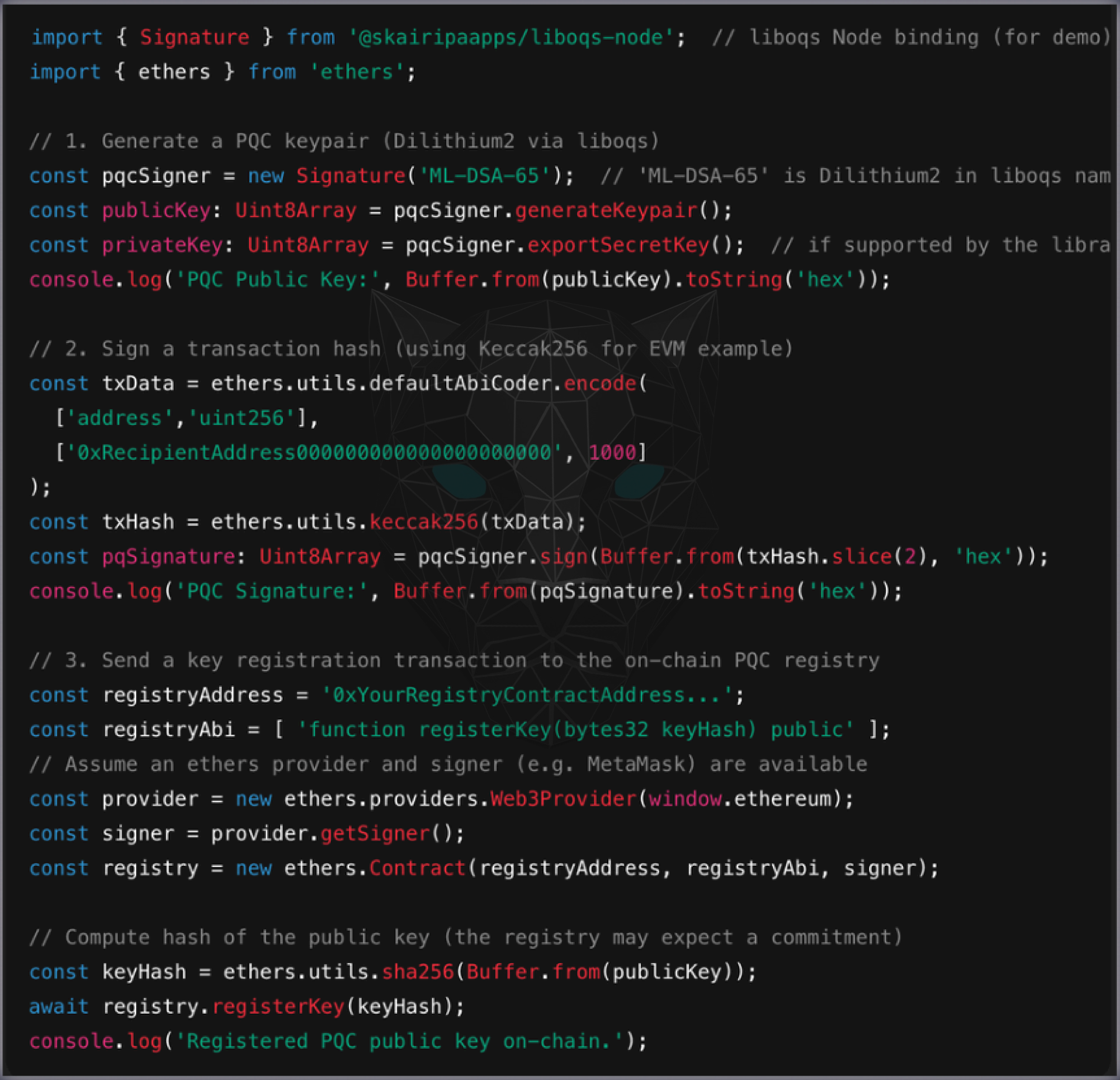
Transactions must carry the PQ signature data in a standardized way. Zenith proposes a convention such as:

```
function transfer(address to, uint256 amount, uint256 algoID, bytes pqSignature) external;
```

Zenith appends algoID+pqSignature to calldata. Smart contracts run transfer logic, then call verifyPQCSignature, validating against stored PQ keys via liboqs. Standardization ensures cross-chain interoperability, with off-chain tooling like Ethers.js handling encoding seamlessly.

TYPESCRIPT SDK (FRONTEND)

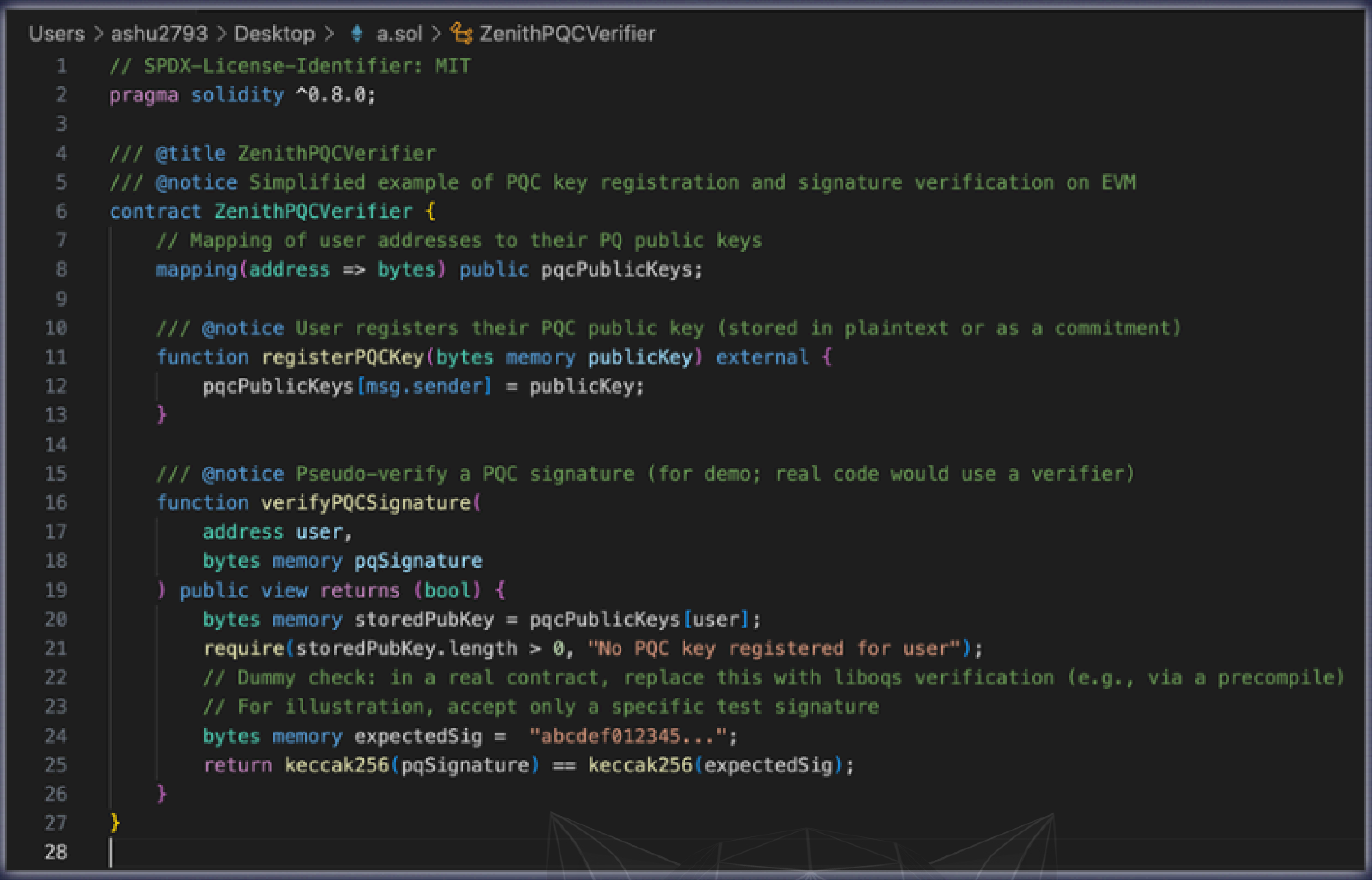
Zenith's TypeScript SDK uses liboqs bindings to generate PQC keys, sign transactions, and register them on-chain. Developers leverage Ethers.js to call registerKey, with registry storing full keys or hashed commitments standardized to NIST/oqs constants.



```
import { Signature } from '@skairipaapps/liboqs-node';  // liboqs Node binding (for demo)
import { ethers } from 'ethers';

// 1. Generate a PQC keypair (Dilithium2 via liboqs)
const pqcSigner = new Signature('ML-DSA-65');  // 'ML-DSA-65' is Dilithium2 in liboqs nam
const publicKey: Uint8Array = pqcSigner.generateKeypair();
const privateKey: Uint8Array = pqcSigner.exportSecretKey();  // if supported by the libra
console.log('PQC Public Key:', Buffer.from(publicKey).toString('hex'));

// 2. Sign a transaction hash (using Keccak256 for EVM example)
const txData = ethers.utils.defaultAbiCoder.encode(
  ['address','uint256'],
  ['0xRecipientAddress0000000000000000000000', 1000]
);
const txHash = ethers.utils.keccak256(txData);
const pqSignature: Uint8Array = pqcSigner.sign(Buffer.from(txHash.slice(2), 'hex'));
console.log('PQC Signature:', Buffer.from(pqSignature).toString('hex'));

// 3. Send a key registration transaction to the on-chain PQC registry
const registryAddress = '0xYourRegistryContractAddress...';
const registryAbi = [ 'function registerKey(bytes32 keyHash) public' ];
// Assume an ethers provider and signer (e.g. MetaMask) are available
const provider = new ethers.providers.Web3Provider(window.ethereum);
const signer = provider.getSigner();
const registry = new ethers.Contract(registryAddress, registryAbi, signer);

// Compute hash of the public key (the registry may expect a commitment)
const keyHash = ethers.utils.sha256(Buffer.from(publicKey));
await registry.registerKey(keyHash);
console.log('Registered PQC public key on-chain.');
```

SOLIDITY (EVM BACKEND)

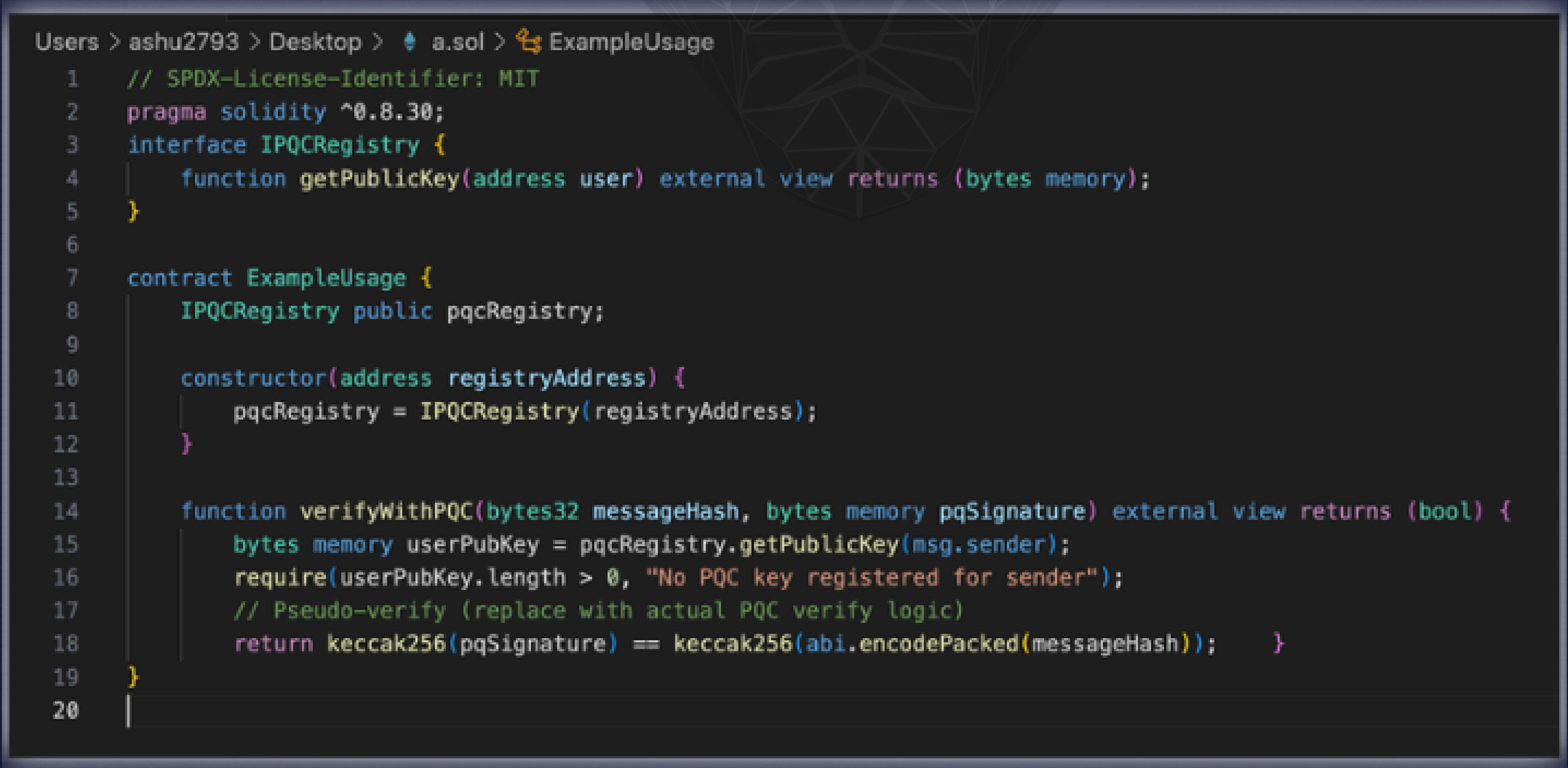
On EVM chains, a smart contract can fetch a user's PQ public key from the registry and verify a PQC signature (typically via a precompile or library). The example below shows a simple contract that stores user PQ public keys and pseudo-verifies a PQ signature.



Users > ashu2793 > Desktop > a.sol > ZenithPQCVerifier

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

/// @title ZenithPQCVerifier
/// @notice Simplified example of PQC key registration and signature verification on EVM
contract ZenithPQCVerifier {
    // Mapping of user addresses to their PQ public keys
    mapping(address => bytes) public pqcPublicKeys;

    /// @notice User registers their PQC public key (stored in plaintext or as a commitment)
    function registerPQCKey(bytes memory publicKey) external {
        pqcPublicKeys[msg.sender] = publicKey;
    }

    /// @notice Pseudo-verify a PQC signature (for demo; real code would use a verifier)
    function verifyPQCSignature(
        address user,
        bytes memory pqSignature
    ) public view returns (bool) {
        bytes memory storedPubKey = pqcPublicKeys[user];
        require(storedPubKey.length > 0, "No PQC key registered for user");
        // Dummy check: in a real contract, replace this with liboqs verification (e.g., via a precompile)
        // For illustration, accept only a specific test signature
        bytes memory expectedSig =  "abcdef012345...";
        return keccak256(pqSignature) == keccak256(expectedSig);
    }
}

```

This contract demonstrates hybrid signing: a transaction could also carry an ECDSA signature, which the contract would check (not shown) before or after verifying the PQC signature. To interact with a centralized registry contract, a contract can call an interface. For example.



Users > ashu2793 > Desktop > a.sol > ExampleUsage

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;
interface IPQCRegistry {
    function getPublicKey(address user) external view returns (bytes memory);
}

contract ExampleUsage {
    IPQCRegistry public pqcRegistry;

    constructor(address registryAddress) {
        pqcRegistry = IPQCRegistry(registryAddress);
    }

    function verifyWithPQC(bytes32 messageHash, bytes memory pqSignature) external view returns (bool) {
        bytes memory userPubKey = pqcRegistry.getPublicKey(msg.sender);
        require(userPubKey.length > 0, "No PQC key registered for sender");
        // Pseudo-verify (replace with actual PQC verify logic)
        return keccak256(pqSignature) == keccak256(abi.encodePacked(messageHash));    }
}

```

RUST (COSMOS CHAINS)

On Cosmos (Tendermint) chains, transactions use SHA-256 hashing, and the PQC registry can be a CosmWasm contract or native module. In Rust (CosmWasm) code, one can use the oqs crate to verify PQC signatures. For example:

This snippet uses **oqs::sig::Sig::verify** to check a Dilithium-2 signature. The contract first queries the registry for the user's public key, then applies verify_pqc_signature. (In a production CosmWasm contract, JSON query messages and error handling would be fleshed out.) Zenith's SDK abstracts these details, auto-hashing with SHA-256 for Cosmos chains.

```rust
use cosmwasm_std::{to_binary, Binary, Deps, StdResult, MessageInfo, Response};
use oqs::sig::{Sig, Algorithm};

/// Verifies a Dilithium2 signature using the `oqs` crate
pub fn verify_pqc_signature(message: &[u8], signature: &[u8], public_key: &[u8]) -> bool
    // Initialize the PQC signature algorithm (Dilithium2)
    let sig_alg = Sig::new(Algorithm::Dilithium2).unwrap();
    sig_alg.verify(message, signature, public_key).unwrap_or(false)
}

// Example CosmWasm execute function integrating registry lookup
pub fn execute_verify(
    deps: Deps,
    info: MessageInfo,
    message: Binary,
    pq_signature: Binary,
) -> StdResult<Response> {
    // Query the PQC key registry contract for the sender's public key
    let query = cosmwasm_std::WasmQuery::Smart {
        contract_addr: "pqc_registry_address".to_string(),
        msg: to_binary(&{"get_public_key": {"address": info.sender.to_string()}})?,
    };
    let res: Binary = deps.querier.query(&query)?;
    let stored_pubkey = res.0; // Retrieved public key bytes

    // Verify the PQC signature
    let is_valid = verify_pqc_signature(message.as_slice(), pq_signature.as_slice(), &sto
    assert!(is_valid, "Invalid PQC signature");
    Ok(Response::new().add_attribute("action", "pqc_verified"))
}
```

LIBOQS INTEGRATION: COMPILATION AND API USAGE

Zenith's crypto relies on liboqs (Open Quantum Safe library). Key points:

Building liboqs:

It's a portable C library with all NIST PQC schemes. The oqs-sys Rust crate will compile liboqs for you (or you can link a system liboqs). It supports cross-compilation to WASM.

API Usage:

We mainly use the "SIG" API. For each algorithm:

```
OQS_SIG *sig = OQS_SIG_new(OQS_SIG_alg_CRYSTALS_Dilithium_2);
OQS_SIG_keypair(sig, public_key, secret_key);
OQS_SIG_sign(sig, signature, &sig_len, message, msg_len, secret_key);
OQS_SIG_verify(sig, message, msg_len, signature, sig_len, public_key);
OQS_SIG_free(sig);
```

Security Notes :

liboqs is open-source and regularly reviewed. Zenith pins specific versions and runs fuzz tests. As with any crypto library, we watch for updates (NIST vulnerabilities or patches). All memory with keys/signatures is zeroized after use to prevent leaks. The WASM modules are audited to ensure no side-channel leaks in JavaScript environments.

WASM Constraints:

When compiling to WASM for browsers, liboqs's algorithms fit well – just include only needed algorithms to keep module size small. Zenith's build automates this selection.

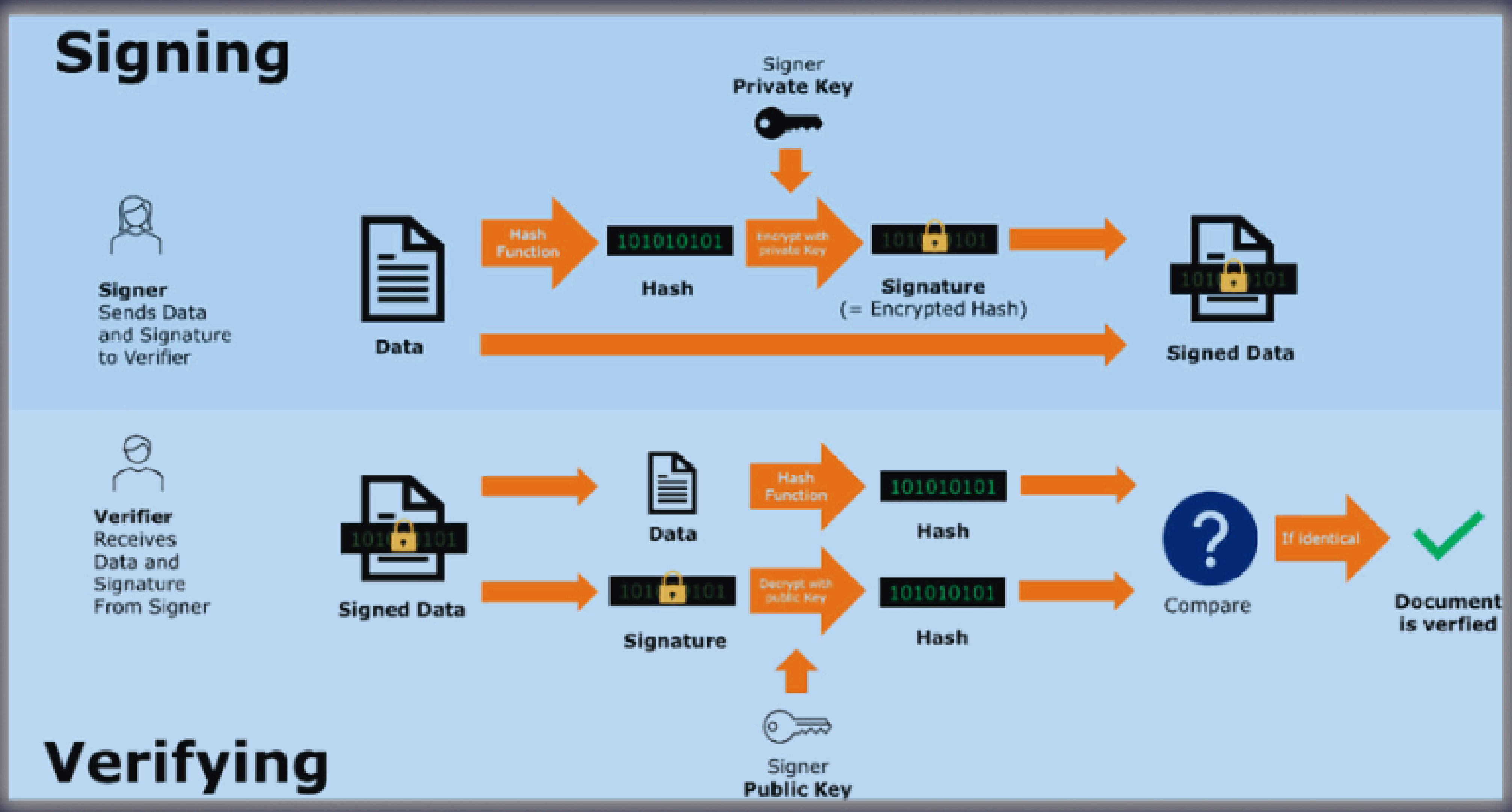
Zenith PQC Layer is designed to be Chain-Agnostic

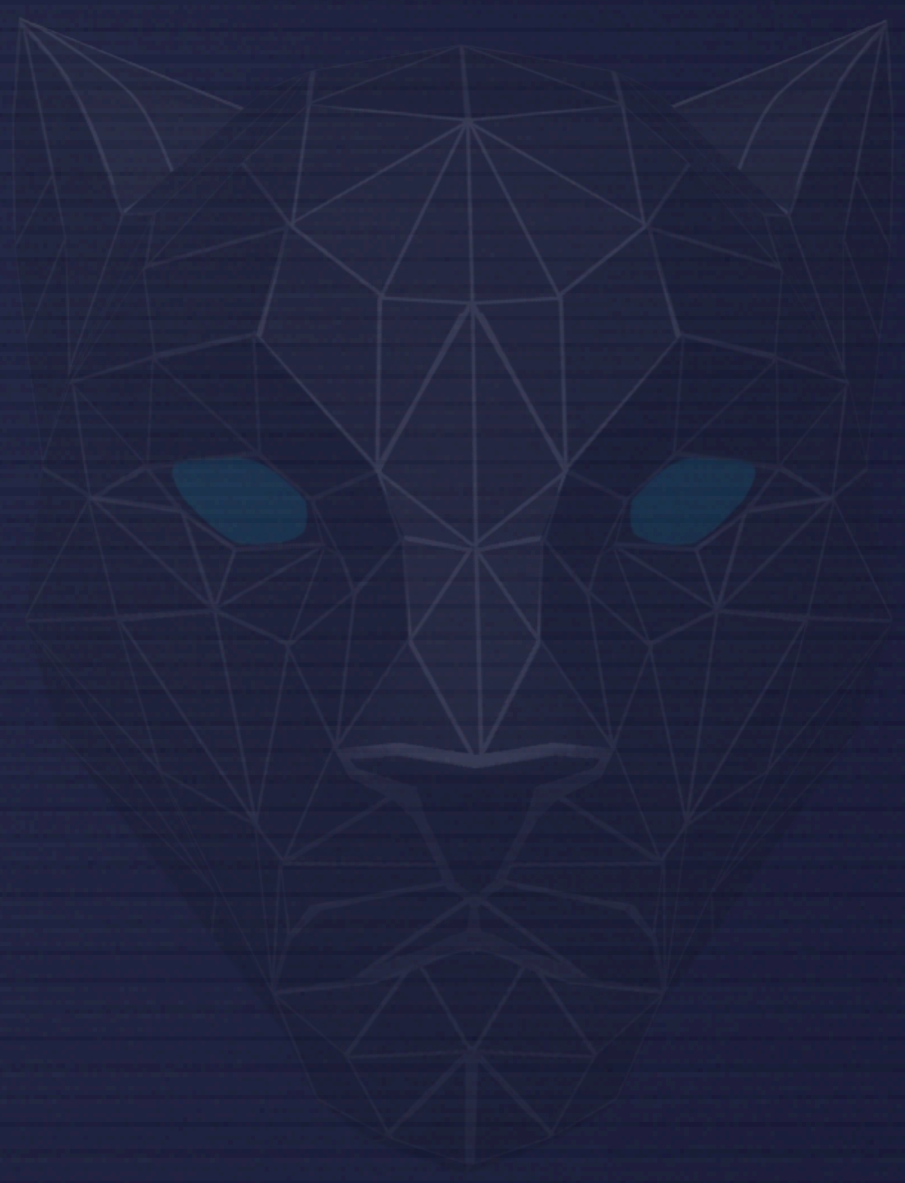
EVM Chains:

On Ethereum/BSC/Polygon, Zenith transactions include PQ signatures in calldata. Verification can use existing opcodes or a planned precompile (e.g. EIP proposals for lattice-based verify). Because Zenith only adds calldata, it works seamlessly with EVM-based smart contracts.

Tendermint/Cosmos Chains:

These use SHA-256 hashing and different signature schemes. Zenith's SDK auto-detects this and uses, e.g., SHA-256 hashing before PQ signing. The on-chain registry contract can be a Cosmos module or smart contract on CosmWasm.

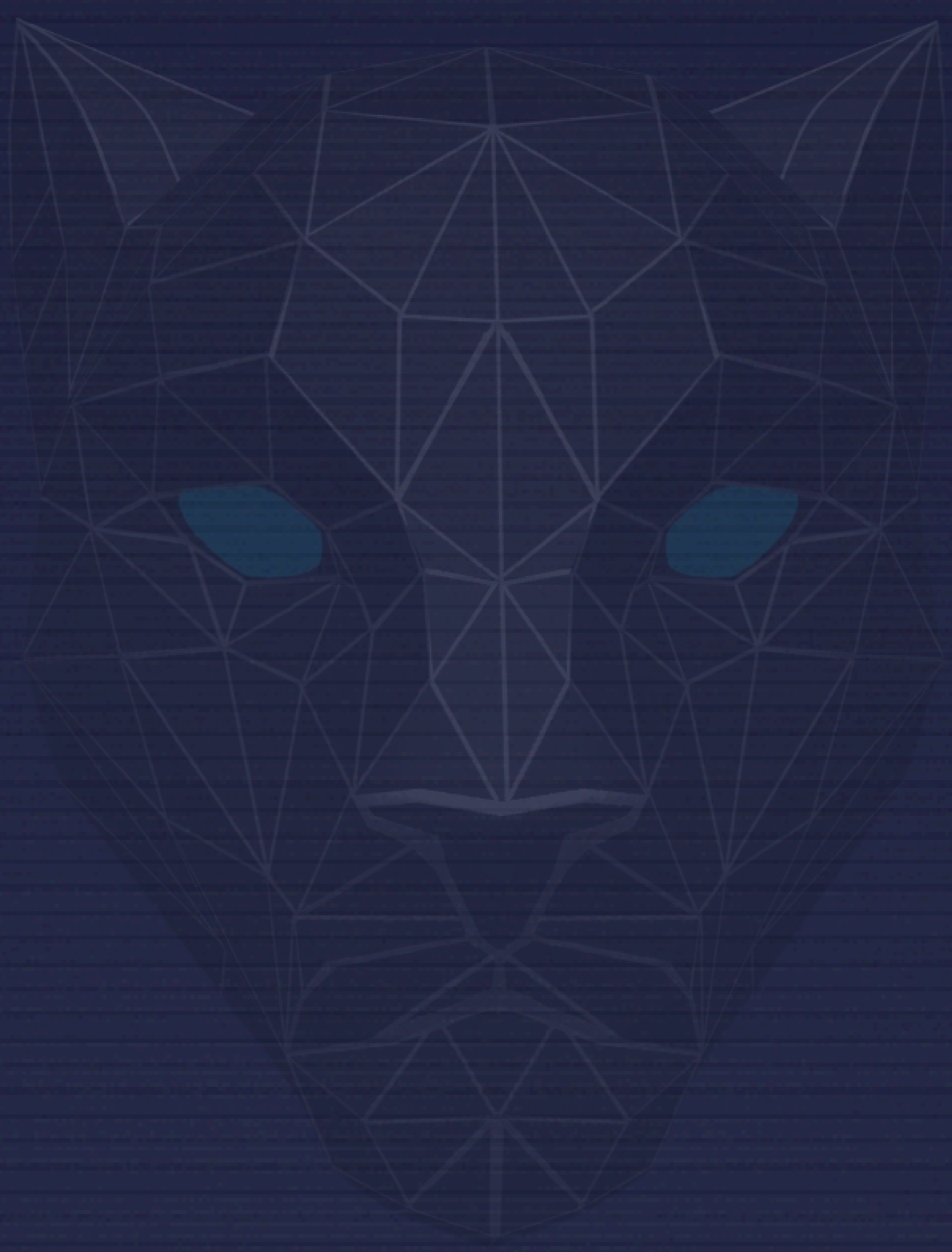




Solana & Others Non-EVM :

Non-EVM chains (Solana uses Borsh, Ed25519) can integrate Zenith by modifying the client-side code: when building a transaction message, pass it through Zenith's signPQC before finalizing. The registry can be on-chain account data.

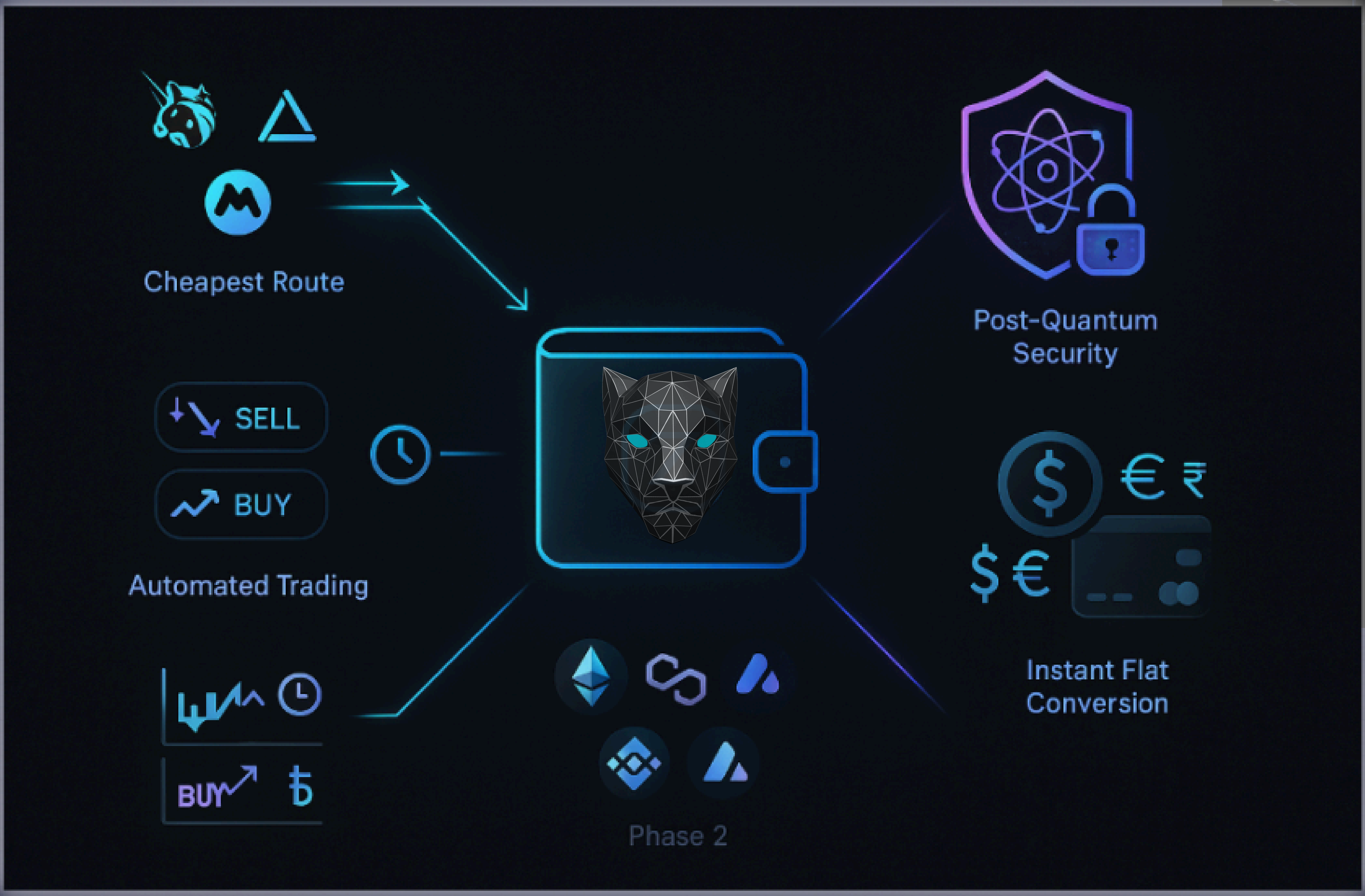
Zenith's key is that no change to consensus rules is needed on any chain. It purely extends the client and contract logic. Any chain that allows arbitrary byte arrays in transactions or accounts can support Zenith.



OTHER PRODUCTS IN ZENITH ECOSYSTEM

Zenith Wallet - The World’s First Multi-Aggregator Web3 Wallet

Zenith Wallet is a next-generation, non-custodial crypto wallet built to redefine security and efficiency in Web3. Unlike competitors, it integrates multiple DEX aggregators (1inch, Paraswap, Matcha, OpenOcean) to guarantee the lowest trading fees with a flat 0.5% service fee cheaper than any other wallet in the market.



With multi-chain support and upcoming Solana integration, Zenith Wallet enables seamless cross-chain swaps. Features like automated trading triggers, instant fiat conversion, and direct Visa/Mastercard integration make it a complete gateway for mass Web3 adoption.

Zenith Gaming Multi-Platform Game Development

Zenith Gaming is the interactive entertainment arm of the Zenith ecosystem, focused on delivering cinematic, culturally rich experiences across PC, mobile, and console (PS5) platforms. Our development process combines in-house IP creation with custom game builds for enterprise and Web3 clients, leveraging the latest in gaming engines, latest integration frameworks with post-quantum secure transaction systems.

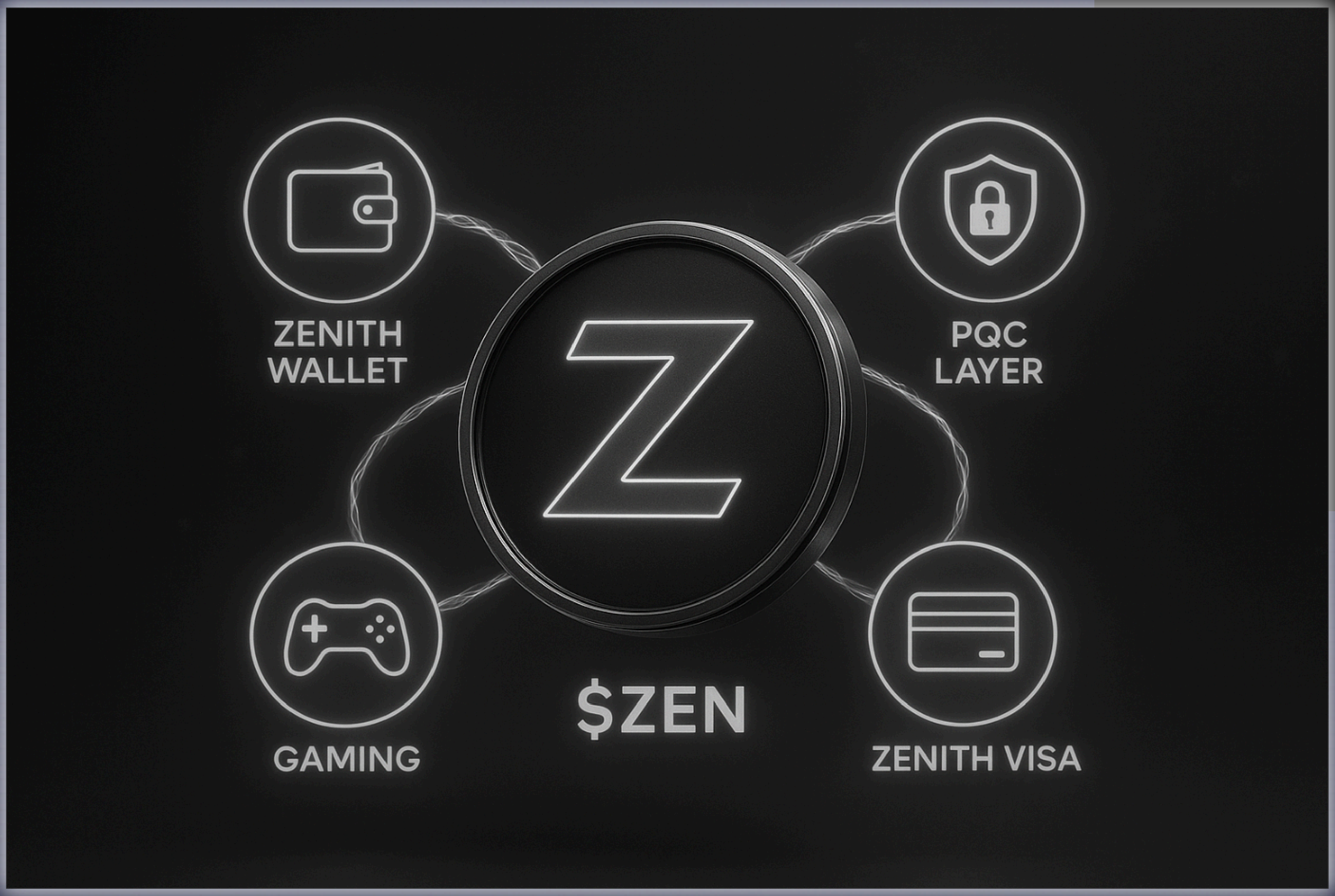
By combining technical mastery, blockchain-native design, and platform versatility, Zenith Gaming positions itself as a full-stack Web3 gaming studio capable of delivering products that resonate globally — whether in-house IPs or bespoke client experiences.

Zenith Visa - The Future of Crypto Payments

Zenith Visa bridges crypto and real-world spending with instant on-chain to fiat conversion, accepted globally wherever Visa/Mastercard work. Users spend $ZEN, stablecoins, or crypto seamlessly, earning rewards in $ZEN with token-gated premium perks. A 1,000-user pilot in 2025 demonstrates strong traction, making Zenith Visa a high-impact adoption driver.

$ZEN- Powering the Quantum-Secure Web3 Future

$ZEN is the native utility token fueling the Zenith ecosystem. It powers wallet fees, PQC SDK licensing, card rewards, in-game economies, and governance. With strategic distribution to developers and ecosystem participants, $ZEN ensures adoption across wallets, payments, and gaming establishing itself as the backbone of Zenith's quantum-secure Web3 infrastructure.



ROADMAP

2025 - LAUNCH YEAR

- Zenith Wallet (multi-aggregator for cheapest fees)
- Zenith PQC Layer v1.0 (SDK for L1/L2)
- Zenith Visa Card launch (pilot)
- Kishkindha (The Tale of 2 Brothers) - PC game launch
- $ZEN Token Launch - December, 2025

THANK YOU

WEBSITE

TWITTER

info@zenithstudio.live