

QAT: a quantum-safe box for onchain assets

Version 1.1, October 2026 QAT contributors

Status: this paper describes the v4 contracts and app, revised after two internal adversarial reviews of the contracts, a focused check of the v4 change, and two reviews of the web app, all on 9 October 2026 (Sections 6.5 and 8). The factory is deployed on Robinhood Chain, with its source verified on Sourcify (Section 7.7). The code has not been audited by a third party.

Abstract

QAT ("the quantum cat") is a vault contract for Robinhood Chain, an Arbitrum Orbit layer-2 network with chain id 4663. Assets in a QAT vault, a "box", leave only in a transaction that is sent by the box's owner wallet and carries a fresh XMSS signature, verified onchain. XMSS is the stateful hash-based signature scheme of RFC 8391 and NIST SP 800-208. We use the parameter set XMSS-SHA2_10_256, whose security rests only on SHA-256. An attacker able to recover every ECDSA private key, as a large quantum computer running Shor's algorithm could, controls the first lock and nothing else.

The contracts keep the scheme's state. Each box has its own key, derived in the browser from 24 BIP-39 words, the chain id and the box address; the factory registers each key once, ever, after the key proves itself. Every digest names the chain, the box and the key generation. Once a valid signature reaches the box its one-time key is burnt, even if the action is refused. The last two keys of each tree are reserved, one for recovery and one for rotation, so the owner can always rotate and a lost owner wallet can always be replaced with the quantum key alone, after 7 days. For the day ECDSA really falls, the owner can switch a box to Q-Day mode with both locks, after which any wallet may send what the quantum key signs, and a public "canary" address with no private key tells the owner when that day has come.

Our verifier and JavaScript signer are byte-compatible with the official reference implementation on its vectors. A verification costs 702,715 to 787,905 gas; a two-transfer withdrawal about 0.94 million gas in total (about US\$0.05 on 9 October 2026), and creating a box about 2.8 million (about US\$0.14). The factory is deployed on Robinhood Chain with verified source, and a first box has been created and opened there.

1 The problem

1.1 What a quantum computer breaks

Every externally owned account on an Ethereum-style chain, Robinhood Chain included, is controlled by an ECDSA key on the curve secp256k1. Its security rests on the elliptic-curve discrete logarithm problem. Shor's algorithm solves discrete logarithms in polynomial time on a fault-tolerant quantum computer [1]. Against hash functions, the best known generic quantum attack is Grover's search, which gives only a square-root speed-up [2]. We use "Q-Day" for the first day a machine can recover private keys from public keys in useful time.

An address is the last 20 bytes of the keccak-256 hash of a public key. Until an account signs something, only this hash is public, and Shor's algorithm does not apply to it [3]. Any ECDSA signature reveals the public key, so the first transaction makes it public for good. Off-chain signatures, such as permits or sign-in messages, reveal it to whoever receives them. Every wallet that has ever signed is therefore exposed on Q-Day. A contract account has no private key: only its code decides what it does. QAT is built on this.

On Robinhood Chain this is not a corner case. On 9 October 2026 we took the 50 largest holders of USDG, WETH and 179 priced stock tokens from the chain's Blockscout explorer, priced them at DexScreener pool prices, set aside contracts, and read each remaining address's nonce on Robinhood Chain, Ethereum, Base and Arbitrum. All 20 of the largest wallets had sent transactions, so all 20 public keys were onchain, holding at least \$16.9 million between them (`data/whales.json` , `scripts/whales.mjs`). Holding stock tokens means trading them, and trading means signing.

1.2 How far away

On 31 March 2026, Google Quantum AI and co-authors including Justin Drake (Ethereum Foundation) and Dan Boneh (Stanford) published resource estimates for breaking 256-bit ECDLP on secp256k1 [4]. They need about 1,200 logical qubits and 90 million Toffoli gates, or about 1,450 logical qubits and 70 million, on fewer than 500,000 physical superconducting qubits, about 20 times fewer than earlier estimates [5]. Such a machine would take 18 to 23 minutes per key, or about 9 minutes after a public key appears with half the work precomputed. The authors published a zero-knowledge proof of their circuits rather than the circuits, without peer review at the time [5]. No machine of this size exists. In a survey of 26 experts published on 9 March 2026, a cryptographically relevant quantum computer within 10 years was rated "quite possible (28–49%)", and within 15 years "likely (51–70%)" [6].

The week before this paper:

- **7 October 2026.** Justin Drake wrote: "IMO it is now reasonable to brace for the possibility that ECDSA breaks before qday, in the worst case in months not years", a break meaning "fast private key recovery (e.g. in one week) on available hardware (e.g. a large GPU cluster)" [7]. Others disputed it; Yehuda Lindell, head of cryptography at Coinbase, called it "the very definition of FUD" [7].
- **7 October 2026.** Europol's European Cybercrime Centre published "Quantum Computing and Cryptocurrencies" and "Harvest Now, Decrypt Later" [8]. As reported, they name wallet keys, not chains, as the weak point, call hash functions comparatively resistant, and say of already-exposed wallets that "the only solution is pre-emptive migration" [9].
- **8 October 2026.** CoinDesk, using Glassnode data, reported that more than 6 million BTC (31.2% of circulating supply) sit behind visible public keys, with custodian shares from 2% (Fidelity) to 100% (Robinhood); the figures measure address usage, not immediate risk [10].

Robinhood itself lists the risk. Its quarterly report for the period ended 30 June 2026 says that, because Robinhood Chain is a layer 2 on Ethereum, "advances in computing technology such as quantum computing that could undermine the cryptography underlying Ethereum, could adversely affect Robinhood Chain" [39].

Ethereum accounts are reused by design, so exposure there is higher. Published estimates of the ETH supply held in accounts with a revealed key range from 50–65% [11] to more than 65% [12].

1.3 Deadlines

NIST IR 8547, still an initial public draft (12 November 2024), proposes to deprecate quantum-vulnerable algorithms, ECDSA among them, after 2030 and to disallow them after 2035 [13]. The NSA's CNSA 2.0 suite requires software and firmware signing to use the hash-based schemes LMS or XMSS: "support and prefer" by 2025, exclusively by 2030 [14]. US Executive Order 14412 of 22 June 2026 requires post-quantum signatures for high-value federal systems by 31 December 2031 [15].

We are not aware of a published post-quantum migration plan for Arbitrum, the stack Robinhood Chain runs on [16]. Whatever the date of Q-Day, a key cannot be migrated after it has been broken. QAT offers an account-level migration that works today, without a protocol change.

1.4 Scope

QAT protects how assets leave an account. It does not protect against failures of the chain itself (its sequencer, its bridge or its fraud proofs). It also does not change the powers that a token's own contract gives its issuer.

2 Design goals and threat model

2.1 Goals

- **G1.** Nothing leaves a box without a valid XMSS signature over the exact action, checked by the contract.
- **G2.** The owner wallet must also send the transaction, so a leaked phrase alone cannot move assets.
- **G3.** A one-time key is spent the first time a valid signature for it reaches the contract, whatever happens to the action.
- **G4.** A key serves one box, on one chain, in one generation.
- **G5.** No attacker action can leave the owner without a usable key, or a user who lost the wallet without a way to recover.
- **G6.** A lost wallet can be replaced with the quantum key, after a delay during which the owner can cancel.
- **G7.** No admin, no upgrade path, no custody: only the key holders can affect an existing box.
- **G8.** A published parameter set, byte-compatible with the reference implementation, at a cost of cents per withdrawal.

2.2 Adversaries

- **A1. Q-Day attacker.** Knows the private key behind every ECDSA public key ever revealed, including the owner wallet's and the factory admin's. SHA-256 and keccak-256 remain secure up to generic quantum speed-ups.
- **A2. Phisher.** Has the 24 words but not the owner wallet.

- **A3. Both.** Has the words and the wallet key. This covers A1 combined with A2, or a fully compromised device.
- **A4. Lost secret.** The legitimate user has lost the owner wallet but still has the words, or the reverse.
- **A5. Griefer.** Has no secret. Wants to burn keys, block a recovery or lock assets.
- **A6. Replayer.** Holds valid signatures from the past, from another box, another key generation or another chain.
- **A7. Observer.** Sees transactions, and so signatures, before they are included: a mempool, the sequencer endpoint, or the RPC provider. May also be A1.

We assume that the user's device and browser are honest while the phrase is in use, that the chain executes contracts correctly, and that SHA-256 and keccak-256 are secure.

3 Why hash-based signatures

3.1 The options

Table 1 compares the standardised post-quantum signature families. The gas figures come from open-source EVM verifiers, as published by their authors. None of those verifiers has a third-party audit.

Table 1. Post-quantum signature options on the EVM

Scheme	Standard	Security rests on	State	Reported EVM verification gas
XMSS-SHA2_10_256	RFC 8391, NIST SP 800-208	SHA-256 (second preimage, PRF)	stateful, 1,024 one-time keys per tree	702,715–787,905 (this work, measured); 712,531 (xmss-solidity README) [17]
SLH-DSA-128s	FIPS 205	hash functions	stateless	about 2.0M, 7,856-byte signature [18]
ML-DSA-44	FIPS 204	module lattices	stateless	1,224,368 [19]; 1,196,707, or 847,709 for an EVM-friendly variant [20]
Falcon-512	selected by NIST	NTRU lattices	stateless	about 3.9M; about 1.5M for an EVM-friendly variant [21]

3.2 Security from SHA-256 alone

Every function in XMSS-SHA2_10_256 is SHA-256 with a different 32-byte prefix (Section 4.2). The scheme's security reduces to the second-preimage resistance and pseudorandomness of these

keyed functions [22]. It does not depend on plain collision resistance, and there is no algebraic structure to attack. With $n = 32$ bytes, generic attacks cost about 2^{256} hash evaluations classically and about 2^{128} with Grover's algorithm (RFC 8391 §9.3 and erratum 6024) [22].

Lattice schemes are fast and stateless, but they rest on structured problems with a shorter history as a basis for signatures. A hash-based scheme adds no new kind of assumption: the vault already trusts keccak-256 for its digests.

3.3 Standards, and what we do not claim

XMSS is specified in RFC 8391 (2018) [22], and NIST SP 800-208 (2020) approves XMSS and LMS with SHA-256 at $n = 32$ [23]. SP 800-208 changed key generation (a new function `PRF_keygen`) but not verification. We implement its key generation from a 96-byte seed exactly as the official reference implementation does [24].

SP 800-208 also requires keys and signatures to be generated inside hardware modules that never export private keys (§8.1). A browser signer derived from a written phrase does not meet that requirement. We claim compatibility with the RFC 8391 parameter set and the reference implementation, not NIST conformance.

3.4 State, and why a chain is the right place for it

XMSS is stateful. Each leaf is a WOTS+ one-time key that must sign one message only, and RFC 8391 §4.1.9 says an implementation "MUST NOT output the signature before the private key is updated" [22]. On a device with backups, that state is fragile. A blockchain is a shared state machine, so QAT keeps the counter in the contract, where every party sees the same value: it accepts leaf `i` only if `i >= nextLeaf`, then sets `nextLeaf = i + 1`, even when it goes on to refuse the action. Because the factory registers each key once, no second counter can ever exist for the same key. The remaining duty, never letting two different signatures for one leaf leave the device, falls on the client (Section 6.2).

SLH-DSA would remove state altogether, at about 2.0 million gas and 7,856-byte signatures [18], against about 0.75 million gas and 2,500 bytes here.

3.5 Ethereum's direction

The Ethereum Foundation formed a post-quantum team in January 2026. Its plan moves validator signatures from BLS to leanXMSS, a hash-based XMSS-family signature of about 3,000 bytes aggregated by leanVM/leanMultisig, with core post-quantum infrastructure "by approximately 2029" [25]. The underlying leanSig construction is a generalised XMSS instantiated with the Poseidon hash [26]. The hash is different, but the family is the one QAT uses.

3.6 Related onchain work

The Solana Winternitz Vault (January 2025) gates withdrawals with a Winternitz one-time key and moves the remainder to a fresh vault each time [27]. QRL has used XMSS on its own layer 1 since 2018 [28]. poqeth (AsiaCCS 2025) measured hash-based verification in the EVM; its XMSS uses keccak and costs 5.6 to 7.7 million gas at $w = 16$, $h = 10$ [29]. xmss-solidity (8 October 2026) is an RFC 8391 verifier checked with Halmos against a transcription of the RFC, without a third-party audit [17]. BunkerVault is a WOTS vault over keccak on Ethereum mainnet [30], and Canary an Arbitrum

Stylus program verifying ML-DSA and Falcon [31]. QAT's verifier is its own Yul implementation, tested against the C reference.

4 The scheme as implemented

4.1 Parameters

Table 2. XMSS-SHA2_10_256 as used by QAT

Symbol	Value	Meaning
<code>n</code>	32	bytes per hash value
<code>w</code>	16	Winternitz parameter; <code>lg w = 4</code> bits per digit
<code>len_1</code>	64	message digits, <code>8n / lg w</code>
<code>len_2</code>	3	checksum digits, <code>floor(lg(len_1 * (w - 1)) / lg w) + 1</code>
<code>len</code>	67	WOTS+ chains per one-time key
<code>h</code>	10	tree height
<code>2^h</code>	1,024	one-time keys (leaves) per tree: leaf 0 proves possession, leaves 1–1021 serve <code>execute</code> and <code>setOwner</code> , leaf 1022 recovery or rotation, leaf 1023 rotation
OID	<code>0x00000001</code>	XMSS-SHA2_10_256; pinned in code, not stored
signature	2,500 bytes	<code>4 + 32 + 67*32 + 10*32</code>
public key	64 bytes	<code>root</code> (32) and <code>PUB_SEED</code> (32)
secret seed	96 bytes	<code>SK_SEED</code> (32), <code>SK_PRF</code> (32), <code>PUB_SEED</code> (32)

The RFC's serialised public key has 68 bytes, because it starts with the 4-byte OID. QAT stores only `root` and `PUB_SEED` and fixes the parameter set in code.

4.2 Hash functions

With `toByte(x, y)` meaning the y-byte big-endian encoding of x:

$F(\text{KEY}, M)$	$= \text{SHA-256}(\text{toByte}(0, 32) \parallel \text{KEY} \parallel M)$	M: 32 bytes, input
$H(\text{KEY}, M)$	$= \text{SHA-256}(\text{toByte}(1, 32) \parallel \text{KEY} \parallel M)$	M: 64 bytes, input
$H_{\text{msg}}(\text{KEY}, M)$	$= \text{SHA-256}(\text{toByte}(2, 32) \parallel \text{KEY} \parallel M)$	KEY: 96 bytes, input
$\text{PRF}(\text{KEY}, M)$	$= \text{SHA-256}(\text{toByte}(3, 32) \parallel \text{KEY} \parallel M)$	M: 32 bytes, input
$\text{PRF_keygen}(\text{KEY}, M)$	$= \text{SHA-256}(\text{toByte}(4, 32) \parallel \text{KEY} \parallel M)$	M: 64 bytes (SP 800)

Onchain, every SHA-256 call goes to the EVM precompile at address `0x02` .

4.3 Addresses (ADRS)

Every keyed hash call is tagged with a 32-byte address made of eight 32-bit big-endian words. The contract handles ADRS as a single 256-bit word: word `k` sits at bit offset `224 - 32k` , so word 3 is `shl(128, type)` and word 7 is the low 32 bits.

Table 3. ADRS layout

Word	0	1-2	3	4	5	6	7
OTS (WOTS+ chain)	layer 0	tree 0	type 0	OTS address = <code>idx</code>	chain address <code>i</code>	hash address <code>j</code>	keyAndMask
L-tree	0	0	type 1	L-tree address = <code>idx</code>	tree height	tree index	keyAndMask
Hash tree	0	0	type 2	padding 0	tree height	tree index	keyAndMask

`keyAndMask` is 0 for a key and 1 or 2 for a bitmask. Each hash call thus has its own key and mask, unique per user (through `PUB_SEED`), leaf, chain and position, so an attacker cannot spread one brute-force search over many published values [22].

4.4 WOTS+ chains

The 32-byte message hash `M'` (Section 4.6) is split by `base_w` into 64 digits in `0..15` , high nibble first:

$$m[2k] = M'[k] \gg 4$$

$$m[2k+1] = M'[k] \& 15$$

for $k = 0..31$

A checksum is appended so that raising any message digit lowers at least one checksum digit:

```
csum = (  $\sum_{i=0..63} (15 - m[i])$  ) << 4    the sum is at most 960; csum fits in
m[64..66] = base_w(toByte(csum, 2), 16, 3)  i.e. the three hex digits of the uns
```

One chain step at hash address `j` is

```
KEY_j = PRF(PUB_SEED, ADRS{chain i, hash j, keyAndMask 0})
BM_j  = PRF(PUB_SEED, ADRS{chain i, hash j, keyAndMask 1})
c[j+1] = F(KEY_j, c[j] XOR BM_j)
```

The secret start of chain `i` for leaf `idx` is `c[0] = PRF_keygen(SK_SEED, PUB_SEED || ADRS{type 0, OTS idx, chain i, hash 0, keyAndMask 0})`. The public end is `c[15]`. A signature reveals `c[m[i]]` on each chain, and the verifier runs steps `j = m[i] .. 14` to reach `c[15]`.

A verification runs `S =  $\sum_{i=0..66} (15 - m[i])$` chain steps. With `s` the unshifted checksum sum, `S = s + 45 - (m[64] + m[65] + m[66])`; since `16  $\equiv$  1 (mod 15)`, `s` is congruent to the sum of its hex digits, so `S` is always a multiple of 15. `S` lies between 45 and 990; over 200,000 random digests its mean is 506, its standard deviation 39 and its 99th percentile 600. Gas is linear in `S` (Section 7.5).

4.5 L-tree and Merkle path

The 67 chain ends are compressed into one leaf by an L-tree. Pairs are combined with

```
RAND_HASH(L, R, ADRS) = H(KEY, (L XOR BM_0) || (R XOR BM_1))
  where KEY, BM_0, BM_1 = PRF(PUB_SEED, ADRS with keyAndMask 0, 1, 2)
```

An odd last node is carried up unchanged. The level sizes are 67, 34, 17, 9, 5, 3, 2, 1, which makes 66 `RAND_HASH` calls at heights 0 to 6. The leaf is then hashed up the tree with the 10 authentication nodes in the signature. At height `k` the address is type 2, height `k`, index `idx >> (k+1)`. The current node goes on the left when bit `k` of `idx` is 0, and on the right otherwise.

Each chain step costs 3 SHA-256 calls (two PRF and one F) and each `RAND_HASH` costs 4, so one verification makes `3S + 4*(66 + 10) + 1 = 3S + 305` SHA-256 calls. For the reference vector at leaf 1022, `S = 510` and the count is 1,835.

4.6 Message hashing

The vault passes a 32-byte digest `D` (Section 5.3). XMSS signs it as

```

r = PRF(SK_PRF, toByte(idx, 32))
M' = H_msg(r || root || toByte(idx, 32), D)
    = SHA-256(toByte(2, 32) || r || root || toByte(idx, 32) || D)

```

Including `r`, the root and the index means a forger cannot reuse one search across leaves or across users, and a plain SHA-256 collision does not help (RFC 8391 §9.1) [22]. As in the RFC, `r` depends only on the secret and the leaf index. Signing is therefore deterministic: signing the same `D` at the same leaf twice gives byte-identical output, which is what makes an identical resend safe. The verifier accepts any `r`.

4.7 Signature layout

Offset	Bytes	Field
0	4	<code>idx</code> , big-endian
4	32	<code>r</code>
36	2,144	WOTS+ values <code>σ₀ .. σ₆₆</code>
2,180	320	authentication path <code>auth₀ .. auth₉</code> , where <code>auth_k</code> is the node at height <code>k</code> with index <code>(idx >> k) XOR 1</code>
total	2,500	

4.8 Verification

`XMSS.verify(root, pubSeed, D, sig)` in `XMSS.sol` does the following. It returns the claimed index so the vault can check it against its counter.

```

verify(root, PUB_SEED, D, sig):
    if len(sig) != 2500:                return (false, 0)
    idx = uint32_be(sig[0:4])
    if idx >= 1024:                      return (false, idx)
    r  = sig[4:36]
    M' = SHA-256(toByte(2,32) || r || root || toByte(idx,32) || D)
    m[0..63] = nibbles of M', high first
    csum      = ( $\sum (15 - m[i])$  for i in 0..63) << 4
    m[64..66] = top three nibbles of the 16-bit csum

    for i in 0..66:                      # WOTS+ chains
        x = sig[36 + 32i : 68 + 32i]
        for j in m[i] .. 14:
            A = ADRS(type 0, ots idx, chain i, hash j)
            x = SHA-256(toByte(0,32) || PRF(A,0) || (x XOR PRF(A,1)))
        pk[i] = x

    l = 67; height = 0                  # L-tree
    while l > 1:
        for t in 0 .. floor(l/2) - 1:
            pk[t] = RAND_HASH(pk[2t], pk[2t+1], ADRS(type 1, ltree idx, height,
            if l is odd: pk[floor(l/2)] = pk[l-1]; l = floor(l/2) + 1
            else:      l = l / 2
            height = height + 1

    node = pk[0]                        # Merkle path
    for k in 0..9:
        auth = sig[2180 + 32k : 2212 + 32k]
        A = ADRS(type 2, height k, index idx >> (k+1))
        node = (bit k of idx) ? RAND_HASH(auth, node, A) : RAND_HASH(node, auth,
    return (node == root, idx)

PRF(A, b)          = SHA-256(toByte(3,32) || PUB_SEED || A with keyAndMask b)
RAND_HASH(L, R, A) = SHA-256(toByte(1,32) || PRF(A,0) || (L XOR PRF(A,1)) || (R

```

The contract works in scratch memory above the free-memory pointer and makes no external call other than the precompile. A failed precompile call, which can only happen when gas runs out, reverts.

4.9 Key derivation: one key per box

The quantum key is a fresh BIP-39 phrase of 24 English words, which encodes 256 bits of entropy [32]. The 96-byte XMSS seed is derived with HKDF-SHA256 [33] for one chain, one box and one generation **k** :

```

entropy = BIP39_mnemonic_to_entropy(normalize(words))    32 bytes; checksum ver
seed     = HKDF-SHA256(IKM = entropy,
                        salt = "QAT XMSS-SHA2_10_256",
                        info = "qat/v1/" || chainId || "/" || lowercase(box address
                        L     = 96)
SK_SEED  = seed[0:32]
SK_PRF   = seed[32:64]
PUB_SEED = seed[64:96]

```

`chainId` and `k` are decimal, the address lowercase hex; `normalize` trims, lowercases and single-spaces the words, and a phrase needs a valid BIP-39 checksum. The box address is known in advance: the app draws a random 32-byte salt, and the factory's `predictVault(creator, salt)` gives the CREATE2 address (Section 5.9). One set of words can thus guard several boxes with unrelated keys, and no key can serve two boxes or two chains. Generation `k = 0` is a box's first tree; a rotation within the same words moves to `k + 1` (the app offers this only when no recovery is pending). The box stores `k` as `keyHint`, an untrusted hint; the app tries generations until one matches the stored root and `PUB_SEED`.

The three 32-byte blocks of one HKDF output are independent pseudorandom values, so publishing `PUB_SEED` reveals nothing about the others. The derivation uses the raw BIP-39 entropy, not the PBKDF2 seed wallets derive from the same words. The seed layout is the one taken by the reference function `xmssmt_core_seed_keypair`, so a given seed yields the same keys in both. A search over phrases costs 2^{256} classically and about 2^{128} with Grover's algorithm.

4.10 Key generation and signing cost

Key generation computes 1,024 WOTS+ public keys (67 `PRF_keygen` calls, 67×15 chain steps and 66 L-tree nodes each) plus 1,023 inner nodes: 3,430,396 SHA-256 evaluations. `xmss.js` keeps hash values as 32-bit words, reuses the SHA-256 state after the fixed first block of `PRF` and `PRF_keygen`, and runs in a Web Worker. A full tree takes about a second: 1,076 ms with the same module under Node.js 24 on the development laptop. The 2,047 public tree nodes (65,504 bytes) hold no secret and can be cached; signing then takes 0.5 to 0.7 ms, and verification in JavaScript 0.7 to 0.8 ms.

5 The vault

5.1 State

Each box is a full deployment of `QatVault`, created by the factory with CREATE2. It stores `owner`, `root`, `pubSeed`, `nextLeaf`, `epoch` (the key generation), `keyHint`, `pendingOwner`, `pendingOwnerEta` and `open` (Q-Day mode, Section 5.11), and the factory address as an immutable. A new box starts at `nextLeaf = 1`, because leaf 0 signed the proof of possession at creation. The last two leaves of each tree are reserved:

- `RECOVERY_LEAF = 1022` may sign only `startRecovery` or `rekey`;
- `RESERVED_LEAF = 1023` may sign only `rekey`.

`execute` and `setOwner` therefore use leaves 1 to 1021. `keysLeft()` reports `1022 - nextLeaf`, so a fresh box has 1,021 keys for general use. `RECOVERY_DELAY` is 7 days.

5.2 Two locks

`execute`, `rekey` and `setOwner` act only when the sender is `owner`, which is ECDSA and today's lock, and the transaction carries an XMSS signature over a digest that names the action, which is the post-quantum lock. Before Q-Day a thief needs both the wallet and the words. After Q-Day the first lock can be forged and the second cannot. Recovery (Section 5.7) and Q-Day mode (Section 5.11) are the only paths where the XMSS signature alone is enough.

5.3 Digests

```
executeDigest = keccak256(abi.encode(EXECUTE_TYPEHASH, chainid, address(this)
rekeyDigest   = keccak256(abi.encode(REKEY_TYPEHASH,   chainid, address(this)
setOwnerDigest = keccak256(abi.encode(SET_OWNER_TYPEHASH, chainid, address(this)
recoverDigest  = keccak256(abi.encode(RECOVER_TYPEHASH, chainid, address(this)
setOpenDigest  = keccak256(abi.encode(SET_OPEN_TYPEHASH, chainid, address(this)
createDigest    = keccak256(abi.encode(CREATE_TYPEHASH,  chainid, factory, vault)

EXECUTE_TYPEHASH = keccak256("QAT.execute(uint256 chainId,address vault,uint32
REKEY_TYPEHASH   = keccak256("QAT.rekey(uint256 chainId,address vault,uint32 e
SET_OWNER_TYPEHASH = keccak256("QAT.setOwner(uint256 chainId,address vault,uint3
RECOVER_TYPEHASH  = keccak256("QAT.recover(uint256 chainId,address vault,uint32
SET_OPEN_TYPEHASH = keccak256("QAT.setOpen(uint256 chainId,address vault,uint32
CREATE_TYPEHASH   = keccak256("QAT.create(uint256 chainId,address factory,adde
```

Each digest binds the action type, the chain id, the box address, the key generation, a deadline (except at creation) and the parameters, and `execute` also binds `callGas`, the gas its calls will receive. These are not EIP-712 messages (no domain separator, no `0x1901` prefix) and are signed by XMSS, never by the wallet. The leaf index is not in the digest, but `H_msg` hashes it with the root (Section 4.6), which binds every signature to one leaf of one key. A review fuzz test checked that the digests built by the app's own `box.js` code match these, byte for byte, and the v4 check compared the new `setOpen` digest the same way.

5.4 Spending a leaf

`execute`, `rekey`, `setOwner`, `startRecovery` and `setOpen` go through one internal function, `_spend`:

1. `XMSS.verify(root, pubSeed, digest, sig)` must succeed, otherwise `BadSignature` (an invalid signature names no leaf, so there is nothing to burn);
2. `leaf >= nextLeaf`, otherwise `LeafUsed`;
3. a refusal reason is noted: `EXPIRED` if `block.timestamp > deadline`, else `NOT_OWNER` if an owner-only action was sent by another address while Q-Day mode is off;

4. for leaves 1022 and 1023: `ReservedLeaf` if the action may not use that leaf, and `ReservedRefused` if a reason was noted;
5. `nextLeaf = leaf + 1` ; if a reason was noted, the box emits `Burned(leaf, digest, reason)` and the action returns `false` .

Actions then check their own inputs: a zero new owner, or a new key that fails its checks in `rekey` . A failure there is a third reason, `BAD_INPUT` , handled the same way: the leaf stays burnt, `Burned` is emitted and the call returns `false` . On leaves 1022 and 1023 the call reverts with `ReservedRefused` instead, so a refused call never burns a reserved leaf. Gaps are allowed: skipping from leaf 1 to leaf 5 makes leaves 2 to 4 unusable.

Why burn instead of revert? A reverted transaction still publishes its signature, but leaves its leaf live, and the natural retry signs a different digest at that leaf, the one thing XMSS forbids (Section 6.2). Burning also keeps a failed withdrawal from becoming valid again once the box is refilled. So once a valid signature reaches the box, its leaf is normally dead. After a valid signature, the transaction still reverts in these cases:

- `LeafUsed` : the leaf is already dead;
- `ReservedLeaf` or `ReservedRefused` : a reserved leaf must survive;
- `NotEnoughGas` (Section 5.5): the signature is bound to its gas and can be resent byte for byte;
- running out of gas anywhere, which no contract can prevent.

In each case the leaf stays live and rules R1 and R2 (Section 6.2) apply. A signed `callGas` too large to ever fit cannot lock the box: the owner's next action at a higher leaf supersedes it, and anyone can burn it, since `NOT_OWNER` is decided before the gas check (in Q-Day mode anyone can instead land it, or burn it with too little gas). The flip side of burning on arrival is that anyone who sees a signed transaction before inclusion can submit it from another address and burn its leaf (Section 6.1). They gain nothing, and the owner signs again with the next leaf.

5.5 execute

`execute(calls, callGas, deadline, sig)` takes a list of `Call{to, value, data}` : withdrawals, swaps, any contract call. After `_spend` , the box encodes the batch as a call to its own `run(calls)` . It then reverts with `NotEnoughGas` if `callGas > 264 - 1` or `gasleft() < callGas + callGas/63 + 30,000` . Otherwise it calls `run` with exactly `callGas` gas, copying no return data, and emits `Executed(leaf, digest, success)` .

`run` can be called only by the box. It refuses any call to the factory or to the box itself (`ForbiddenTarget`), then executes the calls in order with `CALL` , never `DELEGATECALL` . If one call fails, it reverts the whole batch with at most 256 bytes of revert data. A batch is therefore all-or-nothing, and a failed batch still spends the leaf. `run` does not check ERC-20 return values, so a token that returns `false` instead of reverting counts as a success; Robinhood Chain stock tokens and USDG revert on failure.

Signing `callGas` stops a tight gas limit from starving the calls under EIP-150's 63/64 rule while the leaf burns (first review M-1). Checking after the copy keeps large batches covered too: a test sends a 60 KB batch at the lowest gas limit that does not revert, and its call still gets the signed gas (second

review L-1). Without `ForbiddenTarget`, any box could call `claimRoot` from a batch and register someone else's root with no proof (second review M-1). Capping revert data, and copying none in `execute`, contains returndata bombs (Table 5).

5.6 rekey, the reserved leaves, and one key per box

`rekey(newRoot, newPubSeed, newKeyHint, deadline, sig, newSig)` moves the box to a new key: the next generation of the same words, or new words. The current key signs `rekeyDigest` with any leaf $\geq \text{nextLeaf}$, including 1022 and 1023. The new key signs the same digest with its leaf 0, as proof of possession. The new root must differ from the current one, and the factory's `claimRoot` must accept it. It does so only for a root never registered before, at creation or by any rekey, and since `run` cannot reach the factory, only from `rekey` after that proof. If all checks pass, the box installs the key, sets `nextLeaf = 1`, increments `epoch` and cancels any pending recovery. Every older signature is then dead, because each digest includes `epoch`. A box can never return to an earlier key.

Leaf 1023 exists because anyone holding the words can send `startRecovery`, and gaps are allowed: a phisher can sign a recovery at leaf 1022, moving `nextLeaf` to 1023, and leaf 1023 still lets the owner rekey, which cancels the recovery and removes the phisher's key. A refused call can never burn it: a stranger replaying the owner's reserved-leaf rekey from another address gets `ReservedRefused`.

After a leak, the new key must come from new words. The next generation of the leaked words is known to the phisher too: the box address and `keyHint` are public. The second review showed a phisher undoing a same-words rotation and taking the box (M-2). The app now disables same-words rotation while a recovery is pending.

`rekey` carries a signed deadline. The app uses 30 minutes on normal leaves, so an abandoned rotation cannot land months later (second review L-5). Every deadline the app signs is counted from the timestamp of the latest block, read together with the box's state, not from the device's clock: a phone that runs late would otherwise sign an action that is already expired onchain, and lose a one-time key to a refusal. Recovery deadlines on screen are judged by block time too. On leaves 1022 and 1023 it signs the maximum deadline ($2^{256} - 1$) and stores the bytes. A refused call there reverts rather than burning, so the identical transaction can be resent until it lands. The epoch then changes, which kills it.

5.7 setOwner and recovery

`setOwner(newOwner, deadline, sig)` hands the box to another wallet immediately. It needs both locks and cancels any pending recovery.

`startRecovery(newOwner, deadline, sig)` has no sender check, so anyone may relay it. The XMSS signature over `recoverDigest` is the only lock. It may use any leaf up to 1022, sets `pendingOwner = newOwner` and `pendingOwnerEta = now + 7 days`, and emits `RecoveryStarted`. A second call replaces the first one and restarts the clock. After the delay, anyone may call `finishRecovery()`, which makes the pending owner the owner. Leaf 1022 is reserved for recovery so that an owner who has used every normal key and then loses the wallet can still recover (second review L-6).

The owner cancels a recovery by calling `rekey` or `setOwner` during the delay; both need the wallet and the words. There is deliberately no wallet-only veto: a Q-Day attacker holds the owner key, and

could use such a veto to freeze every recovery a user starts after losing the wallet. The cost is that the words alone take the box if the owner does not react within 7 days (first review M-3). That is why the delay was raised from 3 to 7 days. When a box with a recovery under way is opened, the app shows a banner that sends the owner to new words, and devices subscribed to the box get a push alert within minutes (Section 6.6). If the wallet is lost and the words have also leaked, the last recovery wins, and a phisher who signs on leaf 1022 leaves the legitimate user no move.

5.8 Other entry points

- `receive()` accepts ETH and emits no event, so plain 2,300-gas sends from older contracts work. Tokens are sent to the box address as to any address. ERC-721/1155 receiver hooks and ERC-165 let NFTs arrive through safe transfers. There is no `fallback()`, so ETH sent with calldata and ERC-777 `tokensReceived` hooks revert.
- `initialize` can only be called by the factory, once, in the transaction that deploys the box.

5.9 The factory, its fee, and the absence of an admin over boxes

`QatFactory.createVault(root, pubSeed, keyHint, salt, proof)` checks the fee and refuses a root already registered (`RootUsed`). It also requires `proof`, the key's leaf-0 signature over `createDigest(predictVault(caller, salt), caller, root, pubSeed, keyHint)` (`BadProof` otherwise). It then registers the root, deploys the box with CREATE2 under the salt `keccak256(abi.encode(caller, salt))`, and initialises it with the caller as owner. A root can be registered only with a leaf-0 proof from its key: here, or through `rekey`. A copied creation transaction sent from another address targets another box address, so its proof fails.

The fee is 0 at deployment; the admin may set it up to `MAX_FEE = 0.01 ether`, with a non-zero recipient. It is waived when a \$QAT token is set, `holderMin` is non-zero, and the creator holds at least `holderMin`. That balance is read with a `staticcall` capped at 50,000 gas that must return 32 bytes; otherwise the fee applies, so a broken token cannot block creation. `msg.value` must equal the fee, which is forwarded in the same transaction. A fee recipient that reverts blocks paid creation, but not free creation for holders. The token can be set once; the threshold can change.

The admin can call `setFee`, `setToken` and `proposeAdmin`; a new admin must call `acceptAdmin`. `claimRoot` can only be called by boxes this factory deployed. A box stores the factory address and uses it only to accept its one initialisation and to register new roots; no other box function is reserved to the factory, and none to the admin. An admin key broken on Q-Day could change the price of new boxes and nothing more.

5.10 Trading inside a box

Because `execute` runs any batch of calls, a box can trade without its assets ever leaving it. The app builds one batch per trade: an exact-amount `approve` of the input token to Uniswap's SwapRouter02 on Robinhood Chain, then `SwapRouter02.multicall(deadline, [exactInput(path, recipient = box, amountIn, amountOutMinimum)])`. Routes go through USDG (`token → USDG → token`, using each token's deepest USDG pool, listed in `data/pools.json`: 71 pools held at least \$50 of USDG on 9 October 2026). ETH is wrapped by the router (`msg.value` plus `refundETH`) and unwrapped on the way out (`recipient = router`, then `unwrapWETH9(minimum, box)`). The quote comes from QuoterV2; the minimum output is the quote minus 1%. One XMSS signature covers the whole batch,

the batch is simulated from the box before signing (Section 6.2), and the output lands in the box. The approval is consumed by the swap, so no allowance is left behind.

The same mechanism lets a box collect a coin's creator fees. Pons, a launchpad on Robinhood Chain, credits creator fees to the recipient named at launch in a fee escrow, and only that recipient can call `claim()` (ETH) or `claimToken(token)`. A box can be that recipient: the app reads the escrow balance, and one signed batch of `claim` calls brings the fees into the box. `receive()` takes the ETH without an event, so the 2,300-gas sends of older contracts also work.

5.11 Q-Day mode

The owner lock is ECDSA, the scheme QAT expects to fall. Once it does, an attacker who forges the owner key cannot open the box, but can still get in the way: drain the owner wallet's gas money, race its nonces, or burn its pending actions. The only way out would then be a 7-day recovery to a new ECDSA wallet, itself exposed after its first transaction.

`setOpen(on, deadline, sig)` switches the box to Q-Day mode and back. It always needs both locks, in either direction: the sender must be `owner` and the XMSS signature must cover `setOpenDigest(on, deadline)`. It uses a normal leaf (1 to 1021), and a refusal burns that leaf like any other action. While `open` is true, `_spend` skips the sender check, so any wallet can send `execute`, `setOwner` or `rekey`, and the XMSS signature is the only lock. A user whose owner wallet is broken or empty can then open the box from any fresh wallet, with the words alone. `setOpen` itself still needs the owner wallet, so neither the words alone (a phisher) nor the wallet alone (a Q-Day attacker) can change the mode.

The cost is the one the v4 check pointed out: in Q-Day mode, whoever has the words has the box, at once and without the 7-day window of a recovery. That is the right trade once ECDSA is broken, and the wrong one before. The app shows the mode as a chip on the box, asks for confirmation with this warning, and points to the canary (Section 6.7) as the signal to switch. A rotation (`rekey`) keeps the mode: new words change the key, not who may send.

6 Security analysis

6.1 What each adversary can do

Table 4. Adversaries against one box

Adversary	Can	Cannot
A1 Q-Day attacker (all ECDSA keys)	send transactions as the owner; land the owner's signed actions early, or burn them as A7 does; set new-box fees as admin	move assets, change the key or the owner, or start a recovery without a fresh XMSS signature; cancel a recovery; reuse a spent leaf; starve the calls of gas (<code>callGas</code> is signed and checked after the batch is copied)
A2 Phisher (the words)	start a recovery to themselves on any leaf up to 1022 (7-day delay), pushing <code>nextLeaf</code> up to 1023	act as owner; use leaf 1023; register the root of new words the owner rotates to (a box cannot call the factory, and creation needs that key's proof); finish a recovery that the owner cancels within 7 days
A3 Both	everything the owner can do	(no protection)
A4 Lost wallet	start a recovery to a new wallet, even after the last normal key is used (leaf 1022), and take over after 7 days	win against a phisher who also holds the words (Section 5.7)
A4 Lost words	nothing that moves assets; the box stays closed for good	
A5 Griefer (no secret)	send assets in; call <code>finishRecovery</code> once a recovery is due	forge a signature, burn a leaf whose signature they have not seen, burn a reserved leaf, register someone else's root, call <code>run</code> or <code>initialize</code> , block creation through the token
A6 Replayer	(nothing)	replay on another chain, box, key generation or function (all in the digest), after the deadline, or on a spent leaf
A7 Observer	see an owner action before inclusion and burn its leaf by submitting it first from another address, as often as they see one	change the action or use its signature for anything else

In Q-Day mode (Section 5.11) the sender check is gone: A1 loses the little it had (its submissions are relayed rather than refused), and A2 becomes A3, without the 7-day delay. Neither A1 nor A2 alone

can switch the mode on or off.

Row A7 is accepted by design (second review L-4). Burning on arrival is what keeps one-time keys one-time. Robinhood Chain has no public mempool, but the RPC that relays a transaction sees it first, so users should send through an RPC they trust. Section 9 describes a relayer design that would remove this lever.

6.2 One-time key reuse and the client rules

A WOTS+ key that signs two different messages reveals, on each chain, the lower of the two positions, and a forger can then sign any message whose digits are all at or above those minima. Our first security review computed the cost exactly (dynamic programming over the checksum, 400 random cases each). With two public signatures on one leaf, the median is $2^{59.5}$ SHA-256 trials, and 22.5% of cases need at most 2^{40} . With three the median is 2^{30} , and with four 2^{21} (see also [34] [35]). Key reuse is the practical risk, far more than the mathematics.

The contracts close most paths to reuse (Sections 4.9, 5.4, 5.6). What remains is a signature that leaves the device but never reaches the box, followed by a different signature at the same leaf.

`src/js/box.js` implements these rules:

- **R1.** A leaf is recorded as used in the browser's storage (per box and generation) before its signature leaves the page. Forgetting a box removes it from the list but keeps this record.
- **R2.** The leaf is `max(onchain nextLeaf, highest leaf recorded on this device + 1)`. A device with no record of the box adds a random skip of 1 to 16 leaves, as long as that stays below 1022, so two devices rarely pick the same leaf. `execute` and `setOwner` may use leaves up to 1021, `startRecovery` up to 1022, and `rekey` up to 1023.
- **R3.** No signature is sent to an RPC for a dry run:
 - `callGas` comes from `eth_estimateGas` on `run(calls)` sent from the box's own address, with no signature (`estimate × 1.4 + 40,000`), and if that simulation reverts, nothing is signed;
 - the L1 part of the gas is estimated with a random signature of the same size;
 - every signature is verified locally with `xmss.js` before it leaves.

A wallet may still simulate the transaction through its own provider before the user confirms; R1 already counts the leaf as used by then. The one real signature the app itself simulates is the creation proof. That is safe because the key is derived from the box address, the app draws a new salt and key whenever the connected wallet changes, and the box never accepts leaf 0. So leaf 0 signs one `createDigest` only.

- **R4.** A rotation signed on leaf 1022 or 1023 is stored. The box page then shows "A rotation is waiting", which resends the identical bytes without asking for the words. Any other failure moves to the next leaf.
- **R5.** After a leak, or while a recovery is pending, rotate to new words; the app disables same-words rotation during a pending recovery.

- **R6.** `execute` , `setOwner` , `startRecovery` and normal-leaf `rekey` are signed with a 30-minute deadline.

Two residual cases remain:

- **The record lives in one browser.** A restored or second device relies on `nextLeaf` and the random skip. These cover every signature that reached the chain, but not one still in flight elsewhere.
- **A recovery on leaf 1022.** The app signs it with no deadline and keeps the bytes, as it does for rotations on the reserved leaves, so a retry resends the identical transaction and never a second message on that leaf. Only a second device without that record, choosing a different new owner, could sign leaf 1022 again.

6.3 Why a forgery needs a SHA-256 break (sketch)

Suppose the box accepts a digest `D` at leaf `idx` that the owner never signed there. The verifier recomputed the stored root from the signature, so one of three things happened:

1. **The leaf was never signed.** Follow the recomputation down from the root, where it matches the honest tree, to the first place where the adversary's inputs differ from the honest ones. If there is such a place, the adversary has found a second preimage of keyed `H` (in the tree or the L-tree) or of keyed `F` (in a chain). If there is none, the adversary's chain values equal honest values at positions that were never published, which means it computed a preimage of keyed `F`. These are exactly the properties assumed in Section 3.2.
2. **The leaf signed $D_1 \neq D$ once.** The adversary may search over `(r, D)`. If the digits of `M'` are all at or above those of `M'_1` with one higher, the checksum sum is lower, so some checksum digit is lower than revealed and its chain must be run backwards: a preimage of keyed `F`. If all digits are equal, `M' = M'_1`: a second preimage of SHA-256 on a target that includes this root and this index, so the search cannot be shared across targets.
3. **Otherwise**, the keys and masks derived with `PRF` are distinguishable from random, which breaks the PRF assumption on SHA-256.

Under the RFC 8391 analysis each case costs about 2^{256} classical or 2^{128} quantum SHA-256 evaluations [22]. Separately, a keccak-256 collision between two call lists would let one signature authorise both; finding one costs about 2^{128} classical work and only helps someone who can get the owner to sign one of the two lists.

6.4 What QAT does not protect

- **The 24 words.** With the owner wallet, they open the box. Alone, they start a recovery that the owner must cancel within 7 days. Lost words cannot be recovered by anyone.
- **The device.** Browser malware can read the phrase as it is typed or change the calls before signing. A digest is not human-readable, so the user relies on the app to show what is signed; a tampered build of the app amounts to malware. The app zeroes the secret seed copies it controls, but the phrase string itself stays in browser memory until garbage collection.
- **The RPC.** A dishonest RPC can hide transactions, misreport `nextLeaf` (R2 defends against this), censor, or burn every signed transaction it relays by submitting it first from another address. It cannot forge a signature. Use a trusted RPC.

- **Our code.** It has had two internal reviews (Section 8), not a third-party audit. A bug in the verifier, the vault, the factory or the key derivation could defeat everything above.
- **The chain and the tokens.** See Section 1.4, and the token behaviours in Sections 5.5 and 5.8.

6.5 The web app

The page where people type their words is part of the attack surface. A review of the web app on 9 October 2026 found no cross-site scripting and no path for the words to leave the browser. It reported two high, three medium and seven low findings, all fixed the same day. The app now works as follows:

- **Real boxes only.** Before it shows a box, the app reads `isVault(box)` from the factory in the same call as the box's state. A contract that merely copies the box's view functions is shown as "Not a QAT box" and dropped from the device's list; the verify page does the same. Without this check, a look-alike contract could be presented on the official site as "your box", and its victim would fill it.
- **Strict CSP.** The `Content-Security-Policy` allows scripts, connections and images from the site itself only (`script-src 'self'` , `connect-src 'self'` , `img-src 'self' data: blob:`), with `frame-ancestors 'none'` , `base-uri 'none'` and `object-src 'none'` . Chain reads go through the site's own RPC relay, token logos are served by the site, and the page makes no third-party requests. Wallet extensions run outside the page and use their own providers. Pages also send `Cross-Origin-Opener-Policy: same-origin` and a restrictive `Permissions-Policy` .
- **Word fields.** Every field that takes the 24 words turns off autocomplete, autocorrect, spellcheck and capitalisation, carries opt-out attributes for Grammarly, 1Password and LastPass, and is cleared on `pagehide` . After "Copy anyway" on new words, the app clears the clipboard 60 seconds later if the page still has focus.
- **The words stay on the page.** They are read from the field and passed to the page's own Web Worker, which builds the key. No network request carries them.
- **Whose box is it?** Owning the wallet lock is not enough: whoever made a box keeps its words, and can hand the box over with `setOwner` while keeping them. The app calls a box "yours" only when the connected wallet owns it **and** this device has seen its words open it (at creation, or with "Check my words", a local check that sends nothing). Otherwise it shows "Check this box before you fill it", hides "Put all my bags in", and never shows "Your box is ready". Every viewer of a box in Q-Day mode sees a red banner. A second internal review of the web app found the hand-over attack (its H1); `scripts/e2e-app.mjs` now plays it.
- **Proof means it happened.** The proof page reads the transaction receipt and the box's events: it shows "Done" only when the box acted, and otherwise "Valid signature, but nothing happened" with the reason (reverted, refused after its deadline or from the wrong wallet, failed calls). It decodes Uniswap batches and says "output back into the box" only when every output really goes there.
- **Review before signing.** Before the words sign a withdrawal or a swap, a dialog states the whole action in one sentence ("Send 0.5 ETH to 0xAb...", or "Swap 5 USDG for at least 0.02 NVDA on Uniswap"), then "Nothing else. One one-time key. Valid 30 minutes." For a withdrawal it also checks the destination: another QAT box (it stays quantum-safe), a wallet that has never signed (its key is still hidden), a contract, or a wallet whose public key is already out, in which case it warns that what is sent there is open on Q-Day and offers to make another box. The default

destination is the connected wallet, which is usually the most exposed address the user has. The proof page decodes the same calls for any past transaction.

- **Passkey copy (optional).** A user may keep the words on one device behind a passkey. The app checks that the words open the box, creates a passkey with the WebAuthn `prf` extension and a random 32-byte salt, derives an AES-256-GCM key from the passkey's PRF output with HKDF-SHA-256, and stores only the ciphertext, the salt, the IV and the credential id in `localStorage`, with the box address as associated data. Unlocking asks for user verification (Face ID, Touch ID or a PIN) and fills the words field. Authenticators without PRF are refused. The copy is deleted when the user removes it, forgets the box, or rotates to new words. It changes nothing onchain, and the paper copy stays the backup. It trades typing (and keyloggers) for trust in the device and its passkey.
- **Rate limits.** The site's API endpoints count requests per visitor in memory (an IPv6 /64 counts as one visitor) before they touch the database, and the RPC relay accepts only JSON requests.

6.6 Alerts

A device can subscribe to web-push alerts for a box. The browser creates a push subscription with the site's public VAPID key. The site stores it only if the box passes `isVault` and the subscription's endpoint belongs to an allowlisted browser push service (Google, Mozilla, Apple or Microsoft).

Every 2 minutes a scheduled job:

1. reads new `RecoveryStarted`, `Executed`, `Burned`, `OwnerChanged`, `Rekeyed`, `RecoveryCancelled` and `OpenSet` events with a topic query, in windows of at most 29,000 blocks;
2. advances its cursor before sending;
3. pushes a notification to every device that watches the box, with a 5-second timeout per send.

A `RecoveryStarted` alert goes out with high urgency and tells the owner to rotate to new words. This is the off-site alert that both contract reviews recommended for M-3: an owner no longer has to open the page to learn of a recovery. It depends on the site, its scheduled job and the browser's push service. The contracts do not depend on it.

6.7 The canary

How would anyone know that ECDSA has fallen? QAT watches an address that nobody can sign for. Its public key is built from a public sentence: $x = \text{SHA-256}(\text{"QAT Q-Day canary v1 \#0"}) \bmod p$, which is the x-coordinate of a point on secp256k1 (the first try, `#0`, already lands on the curve), with the even `y`. The address is the last 20 bytes of `keccak256(x || y)`:

```
0x15AD483771d2e4A667679012617a6a613b29C80F
```

No private key was involved in making it, so the only way to send a transaction from it is to solve the discrete logarithm of that point, which is what Shor's algorithm does. Anyone can redo the derivation: the home page has a button that recomputes it in the browser, and `scripts/canary.mjs` does the same in Node. The scheduled job of Section 6.6 reads the canary's nonce every 2 minutes. If it is

ever above zero, the canary section of the home page turns red and every device that watches a box gets a high-urgency alert telling the owner to switch on Q-Day mode while the owner wallet still works. The words need no rotation: breaking ECDSA reveals nothing about them. Anything sent to the canary is a public bounty that only a discrete-log break can collect.

The canary is a signal, not a guarantee. A capable attacker could leave it alone and go after richer keys first, and the alert depends on the site like the others. It costs nothing to watch, and a nonce that moves would be hard to explain any other way.

7 Implementation and verification

7.1 Code

File	Role
<code>contracts/src/XMSS.sol</code> (132 lines)	onchain XMSS-SHA2_10_256 verifier, internal library, Yul
<code>contracts/src/QatVault.sol</code> (364 lines)	the box
<code>contracts/src/QatFactory.sol</code> (182 lines)	CREATE2 deployment, proof of possession, root registry, fee rules
<code>src/js/xmss.js</code>	browser and Node XMSS: key generation, signing, verification
<code>src/js/phrase.js</code> , <code>src/js/keys.js</code>	24 words to a per-box seed; key worker; local check of every signature
<code>src/js/box.js</code>	client rules R1–R6: leaf choice, digests, simulation, sending
<code>scripts/xmss/vectors.c</code>	vector generator linked against the reference implementation
<code>contracts/test/XMSS.t.sol</code> , <code>contracts/test/QatVault.t.sol</code>	Foundry tests

Compiler settings: solc 0.8.26, optimiser at 1,000 runs, EVM version `cancun` , no via-IR. `QatVault` has 9,133 bytes of runtime code (9,197 bytes of initcode). `QatFactory` has 14,473 bytes, since it carries the vault's creation code; both include the verifier.

7.2 Conformance with the reference implementation

`scripts/xmss/vectors.c` links against the official reference implementation [24] at commit `171ccbd` (16 March 2021). It builds a key with `xmssmt_core_seed_keypair` from the 96-byte seed `seed[i] =`

$(7i + 3) \bmod 256$ (root `0xdd83baf7...b29100e3`), signs six messages $m_k[i] = (31k + 13i + 1) \bmod 256$ at leaves 0, 1, 2, 7, 500 and 1022, and checks each with the reference verifier. `xmss.js` reproduces the root and all six signatures byte for byte and accepts them; `XMSS.sol` accepts all six and returns the right leaf index.

A second known-answer test from the same reference (seed `00 01 ... 5f`, $D = 32 \times 0xab$, leaf 0) is also reproduced exactly by our JavaScript: root `0xd898033...690c3483`, $r = 0x11c3e8f9...77071cb2$, $\sigma_0 = 0xab658aa7...b9ebf8dd$, and SHA-256 of the whole signature `dc6b0603...2467b1f5`. The vault tests use this seed.

7.3 Negative tests and fuzzing

The verifier rejects a message with one bit flipped, a wrong root, a wrong `PUB_SEED`, a 2,499-byte signature, index 1024, and leaf 1's signature relabelled as leaf 0. A fuzz test that flips one random bit anywhere in the 2,500-byte signature was rejected in all 256 runs. The JavaScript test also rejects a tampered signature and a tampered message for each vector.

7.4 Vault and factory tests

The 37 vault and factory tests, all passing, sign with `xmss.js` through Foundry's FFI and cover:

- **Creation:** initial state (1,021 keys), bad proofs, a copied creation, a reused key.
- **Withdrawals:** a two-transfer withdrawal, replay, expired and stranger submissions burnt, the Q-Day attacker, `NotEnoughGas` then an identical resend, a failed batch, a 1 MB revert, a 60 KB batch that still gets its signed gas, the self-only `run`, cross-box replay, a batch calling the factory or the box itself.
- **Rotation:** a rekey, no return to a used key, a bad proof, the rekey deadline (burnt on a normal leaf, reverted on leaf 1023), the reserved leaf.
- **Owner and recovery:** `setOwner`, recovery after 7 days, cancellation by `rekey`, the owner's last word with leaf 1023 after a phisher used 1022, a lost wallet after the last normal key recovering with leaf 1022, an expired recovery.
- **Factory:** stipend ETH sends, single initialisation, fees and holders, a code-less token, two-step admin, `claimRoot` restricted to boxes.
- **Q-Day mode:** a stranger's withdrawal refused when closed and landed when open, both locks needed to switch either way (a phisher's attempt burns its leaf), the owner key alone still useless, a rotation relayed by a stranger.

The app's end-to-end test (`scripts/e2e-app.mjs`, 36 checks on a fork of the chain) drives the real interface through every flow above, including a 7-day recovery with the chain clock moved forward and the hand-over attack of Section 6.5.

The reviewers' own proof-of-concept tests (`Review3Qat.t.sol`, 29 tests, including six for Q-Day mode) run with them, for 73 tests in all.

7.5 Gas and cost

Gas grows with the chain-step count `S` of each signature (Section 4.4), so Table 5 gives each measured figure with its `S`, and the same figure adjusted to the mean $S = 506$ at 1,136 gas per

step. Costs use the adjusted figure, at 0.02 gwei and \$2,490 per ETH.

Table 5. Measured gas (Foundry; gas of the call itself)

Operation	S	Measured gas	At S = 506	Cost
verify, leaf 0	480	702,715		\$0.035 (measured)
verify, leaf 1022	510	736,705		\$0.037 (measured)
verify, leaf 7	525	753,805		\$0.038 (measured)
verify, leaves 1, 2, 500	555	787,905 / 787,905 / 787,855		\$0.039 (measured)
execute , 2 transfers (ETH and token)	570	951,311	878,607	\$0.044
same, full transaction: + 21,000 base + 43,064 calldata (3,140 bytes) *			942,671	\$0.047
execute whose call reverts with 1 MB of data	600	1,211,123	1,104,339	\$0.055
execute refused and burnt (not sent by the owner) *	495	796,911	809,407	\$0.040
createVault (deploys the box, checks one XMSS proof) *	465	2,722,875	2,769,451	\$0.138
same, full transaction: + 21,000 + 41,568 calldata (2,724 bytes) *			2,832,019	\$0.141
setOwner *	495	792,227	804,723	\$0.040
startRecovery *	510	831,392	826,848	\$0.041
rekey (two XMSS checks, root registration, cancels a recovery) *	570 + 525	1,697,442	1,603,154	\$0.080

Operation	S	Measured gas	At S = 506	Cost
finishRecovery *		34,096	34,096	\$0.002

Rows without * come from the test suite's logs. Rows marked * come from a separate gas script run against the same contracts, one run each; that script's own two-transfer `execute` (`S = 495` , 866,132 gas) adjusts to 878,628, within 21 gas of the suite's figure.

The six verification points fit $\text{gas} \approx 157,407 + 1,136 \times S$ with a maximum error of 57 gas; leaves 1, 2 and 500 cost the same because they share `S = 555` (`S` moves in steps of 15). With the distribution of `S` from Section 4.4, a verification costs about 732,000 gas at the mean, 839,000 at the 99th percentile and 1.28 million in the theoretical worst case (`S = 990`), far below the chain's per-transaction limit of 32 million gas. A chain step makes three precompile calls at 100 gas for the warm `STATICCALL` plus 96 for the precompile, so about half of its 1,136 gas is call overhead.

Prices: on 9 October 2026 the chain's `eth_gasPrice` was 20,114,000 wei (0.0201 gwei) and its minimum gas price 0.02 gwei. At 0.02 gwei and \$2,490 per ETH, one million gas costs $1e6 \times 2e-11 \text{ ETH} = 2e-5 \text{ ETH} \approx \0.0498 . An Arbitrum Orbit chain also charges for the data it posts to its parent chain. That L1 part comes on top of the figures above. It depends on the calldata (2.7 KB for a creation, 3.1 KB for a two-transfer `execute` , 5.3 KB for a `rekey`), not on the verification work. On 9 October 2026 the chain reported it as zero: `ArbGasInfo` gave an L1 calldata price of 0, and `NodeInterface.gasEstimateL1Component` returned 0 for a 2.6 KB payload. The app still asks `NodeInterface` for it before each transaction.

7.6 Reproducing

```
cd contracts && forge test -vv          # 73 tests; logs per-leaf verification gas,
node scripts/xmss/test-js.mjs          # JS against the reference vectors; prints
./scripts/xmss/vectors > vectors.json
```

The last command regenerates the vectors. `vectors` is `scripts/xmss/vectors.c` compiled with the reference sources at commit `171ccbd` (every `.c` file except `xmss_core_fast.c`) and OpenSSL.

7.7 Deployment

The v4 `QatFactory` is deployed on Robinhood Chain at `0x27B710753B82b62DB11C16d5aA9BBa7be4a30cB2` , in block 83,956,629. Its runtime code is 14,473 bytes, the size of the v4 build in Section 7.1. Its source is verified on Sourcify, with an exact match of both creation and runtime code. On 9 October 2026 its creation fee was 0 and no \$QAT token was set.

A first end-to-end run used it:

1. Box `0x2688DDc96FD4f1Ca5276fc5F6166Fd8115f831f0` was created in transaction `0x93490565792b6790c5dd6b447293dde28fbceae67663ad5c8516681cac270db4` (block 83,956,907;

2,827,605 gas, about \$0.14).

2. The box was filled.

3. It was opened with one-time key #1 in transaction

`0xda7bdca84fb1166a8ba50d0f5a4d2b8043468746c85d407628648a75d698ebaa` (block 83,956,926; 804,539 gas, about \$0.04). The box emitted `Executed` for leaf 1 with `success = true`.

The v3 factory (`0x949D90C26DC5A325630AAd113Ce13dDAAEfe6e44` , block 83,904,659) made one box for the same kind of test, opened in `0x427e4918...f2254a24` . It stays on chain, but the app creates boxes with v4 only.

These are whole-transaction figures from the receipts, with the base cost and calldata included. They differ from Table 5 because every signature has its own `S` , and this batch was not the two-transfer one. Anyone can check them through the chain's RPC or explorer.

8 Security reviews

Two internal adversarial reviews of the contracts ran on 9 October 2026, followed by a focused check of the v4 change. Two reviews of the web app (`research/webaudit.md` , `research/webaudit2.md`) are summarised in Section 6.5. None is a third-party audit. The first two reviews' proof-of-concept tests target the versions they reviewed and are archived in `contracts/test/v1/` ; the later ones run against v4 in `Review3Qat.t.sol` . Every change they led to has a regression test in `QatVault.t.sol` .

8.1 First review (v1 to v2)

The first review covered `XMSS.sol` , `QatVault.sol` , `QatFactory.sol` and the key and signing code in `xmss.js` , `phrase.js` and `keys.js` . It found no critical issue and no deviation of the verifier from RFC 8391 on any point checked. It reported one high, three medium and five low findings, plus nine informational notes. All findings except L-3 and L-5, which were found by inspection, came with proof-of-concept tests.

Table 6. First review: findings and fixes

ID	Finding	Fix in v2
H-1	A valid signature could become public while its leaf stayed live (expired, wrong sender, out of gas, dry runs), and one key could serve several boxes, so a leaf could end up signing two digests	a valid signature burns its leaf on arrival (Section 5.4); per-box, per-chain keys (4.9); factory root registry; client rules R1–R3
M-1	Under the 63/64 gas rule a transaction could succeed with its calls starved, spending the leaf	<code>callGas</code> is signed; <code>NotEnoughGas</code> reverts and the same bytes can be resent (5.5)
M-2	Digests lacked the key generation, and <code>rekey</code> could return to a used root, so old signatures replayed	<code>epoch</code> in every digest; current and registered roots refused (5.6)
M-3	The 24 words alone take the box if the owner does not react within the delay	delay raised from 3 to 7 days; reserved leaf keeps the owner able to cancel; in-app banner; no wallet-only veto, by design (5.7)
L-1	Unbounded revert data could make a batch impossible to run	revert data capped at 256 bytes; <code>execute</code> copies none (5.5)
L-2	A broken or hostile token could block paid box creation	balance read with a 50,000-gas <code>staticcall</code> and a fixed 32-byte return (5.9)
L-3	No proof that anyone can sign for a new root, so a wrong root would lock the funds	leaf-0 proof of possession at creation and rekey (5.6, 5.9)
L-4	ETH sends with the 2,300-gas stipend failed on EIP-1167 clones [36]	full deployments, and <code>receive()</code> emits no event (5.8)
L-5	<code>setFee</code> accepted a zero recipient, and admin changes were single-step	zero recipient refused; <code>proposeAdmin</code> / <code>acceptAdmin</code> (5.9)

8.2 Second review (v2 to v3)

The second review checked v2, every first-review fix, and the browser signing path in `box.js`, `phrase.js`, `keys.js` and the parts of `app.js` that drive them. It ran 16 proof-of-concept tests that signed with `xmss.js` and derived seeds with the real `phrase.js`. A fuzz test that runs the digest code sliced out of `box.js` found the client's digests equal to the contract's for all five actions.

It found no critical or high issue, and the verifier was unchanged. It judged the first review's H-1 fixed except for one regression (its M-1 below), first-review M-1 fixed except for very large batches (L-1), and M-3 mitigated, with two residual paths to theft (M-1, M-2). It reported two medium and six low findings, plus eight informational notes.

Table 7. Second review: findings and what was done in v3

ID	Finding	What was done
M-1	Any box could call <code>factory.claimRoot</code> from a batch and register any root without a proof, squatting creations and rotations and defeating the reserved-leaf rotation during a phisher's recovery	<code>run</code> refuses calls to the factory and to the box itself (<code>ForbiddenTarget</code>) (5.5)
M-2	The app offered a same-words rotation as the answer to a leak, but whoever has the words derives the next generation	same-words rotation disabled while a recovery is pending; banner and copy point to new words (5.6)
L-1	The gas check ran before the batch was copied, so batches above about 60 KB got less than <code>callGas</code>	batch encoded before the check; <code>callGas</code> capped at <code>2^64 - 1</code> (5.5)
L-2	A rotation signed on the reserved leaf could not be resumed from the app	stored bytes resent from "A rotation is waiting", without the words (R4)
L-3	The leaf record lived in one browser, "Forget this box" erased it, and an unknown device started at <code>nextLeaf</code>	record kept when a box is forgotten; random skip of 1 to 16 leaves on a device with no record (R1, R2)
L-4	Any observer can burn owner actions before inclusion	accepted and documented: use a trusted RPC; relay with an owner signature as future work (6.1, 9)
L-5	<code>rekey</code> had no deadline, so an abandoned rotation could land months later	signed deadline; the app uses 30 minutes on normal leaves and the maximum on leaves 1022–1023 (5.6)
L-6	An owner who used leaf 1022 and then lost the wallet was locked out	leaf 1022 reserved for <code>startRecovery</code> (and <code>rekey</code>) (5.7)
I-1	More errors than claimed revert after a valid signature, and a huge <code>callGas</code> panicked	paper corrected (5.4); <code>callGas</code> above <code>2^64 - 1</code> gives <code>NotEnoughGas</code>

ID	Finding	What was done
I-2	Leaf 0 could sign two creation digests if the user switched wallets between attempts	the app draws a new salt and key when the wallet changes (R3)
I-3	Client gas budgets are tight in the theoretical worst case	unchanged; a shortfall reverts with the leaf live, and R1–R2 apply
I-4	<code>run</code> does not check ERC-20 return values; there is no <code>fallback()</code>	documented (5.5, 5.8)
I-5	The worker's copy of the secret seed stayed in memory	zeroed after use; the phrase string still lives until garbage collection (6.4)
I-6	<code>vaults0f</code> lists boxes by creator, so a new owner may not see a box	unchanged, interface only; the app also remembers boxes per device
I-7	A reverting fee recipient blocks paid creation; borrowed \$QAT waives the fee	documented (5.9, 10)
I-8	Five claims in this paper were wrong for v2	corrected in this version (5.4, 5.9, 6.1, 6.2)

First-review M-3 remains mitigated rather than eliminated: a phished set of words is a 7-day race that the owner must notice, and must answer with new words.

8.3 v4 check (Q-Day mode)

A focused check of the v4 change (Section 5.11) re-ran every earlier test, added six proof-of-concept tests and compared the app's `setOpen` digest with the contract's. It found nothing of medium severity or above. It confirmed that switching the mode needs both locks in both directions, that `setOpen` can never spend leaf 1022 or 1023, and that keeping the mode through a rotation is correct. Its one low finding is about the user: in Q-Day mode a phisher with the words drains the box at once, can take ownership and can switch the mode off, with no 7-day window. No contract change can tell the user from the phisher, since both hold the same secret, so the app's warning says to switch only when wallet keys are really being broken.

9 Limits and future work

- **Audit.** No third-party audit yet; until then, only put in a box what you can afford to lose. Also planned: differential testing against xmss-solidity [17] and a symbolic check against the RFC pseudocode.

- **More keys.** A tree holds 1,021 general-purpose keys; each rekey adds a tree. XMSS^{MT} would give far more per public key (one verifier reports 1.85 million gas for a 20-level, 2-layer tree [17]).
- **Device state.** The leaf record of R1–R2 lives in one browser. Syncing it, or reading the box's reverted transactions from an explorer to find leaves already published, would narrow the residual window of Section 6.2. A recovery on leaf 1022 could be signed without a deadline and stored, as rotations are.
- **Relayer and owner signature.** Q-Day mode (5.11) already lets any wallet relay, at the price of making the words the only lock. Before Q-Day, authorising the owner with an EIP-712 ECDSA signature over the XMSS digest, instead of `msg.sender`, would let anyone relay the owner's transaction with both locks kept: a front-runner would land the owner's own action, and burning would stop being a lever (6.1). It would also let an owner wallet act without holding ETH.
- **Withdrawal delay.** A delay of 24 to 48 hours on new destinations, with an allowlist that goes out at once and a lock button, would protect against malware in the one browser that holds both the wallet and the words.
- **Canary in the contract.** A canary contract that pays its bounty to whoever shows a valid ECDSA signature for the canary key could expose a `tripped` flag that boxes read, and switch Q-Day mode on by themselves.
- **ERC-4337 and EIP-7702.** A smart account whose validation step is the XMSS check, along the lines of Kohaku's `pq-account` and ETHFALCON's 4337/7702 demonstrations [37] [21]. After Q-Day the ECDSA lock adds no protection; Q-Day mode already lets the XMSS signature alone authorise, opt-in, and an account would make that the default.
- **Guardian phrases.** Recovery or execution gated by k-of-n XMSS keys, so that losing one phrase is not fatal.
- **Alerts.** Push alerts (Section 6.6) depend on the site and on a browser that keeps its subscription. An e-mail or messaging channel would add a path that does not.
- **Tokens.** Checking ERC-20 return values in `run`, and a `fallback()` for ETH with `calldata`.
- **SLH-DSA option.** Stateless boxes with no key-reuse risk, at about 2.0 million gas and 7,856-byte signatures [18].
- **Native verification.** EIP-7932, EIP-8051 and EIP-8052 propose precompiles for ML-DSA and Falcon [38]; none targets XMSS or is live on Robinhood Chain. A hash-based verification precompile, or a native code path in the Arbitrum stack, would remove most of the call overhead.
- **Gas grinding.** The verifier accepts any `r`, so a signer could try deterministic candidates and keep the smallest `s`; with 1,024 candidates the mean `s` falls from 506 to 392. This departs from the RFC's generation of `r` (verification is unchanged), so QAT does not do it.
- **Hardware signers.** Signing inside a module that never exports the key is what SP 800-208 requires, and would remove the browser from the trusted base.

10 The \$QAT token

\$QAT is a separate ERC-20 token. Boxes do not depend on it: signatures, recovery, rotation and custody work the same without it, and holding it gives no right over any box. Its only role is in `QatFactory.feeFor`: if a creation fee is switched on, an address holding at least `holderMin` \$QAT creates boxes for free. The balance is read for the creating address only, at creation time, so a

borrowed balance also qualifies. If the read fails, the fee applies. The token address can be set once; the threshold is a public parameter that the admin can change. Creation fees are 0 by default, can never exceed 0.01 ETH, and go to the non-zero fee recipient named onchain. Like any ERC-20 balance in an ordinary wallet, \$QAT held outside a box has no protection against Q-Day.

References

- [1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", SIAM Journal on Computing, 1997. arXiv:quant-ph/9508027. <https://arxiv.org/abs/quant-ph/9508027>
- [2] L. K. Grover, "A fast quantum mechanical algorithm for database search", STOC 1996. arXiv:quant-ph/9605043. <https://arxiv.org/abs/quant-ph/9605043>
- [3] V. Buterin, "How to hard-fork to save most users' funds in a quantum emergency", ethresear.ch, 9 March 2024. <https://ethresear.ch/t/how-to-hard-fork-to-save-most-users-funds-in-a-quantum-emergency/18901>
- [4] R. Babbush, C. Gidney, A. Zalcman, M. Broughton, T. Khattar, H. Neven, T. Bergamaschi, J. Drake, D. Boneh, "Securing Elliptic Curve Cryptocurrencies against Quantum Vulnerabilities: Resource Estimates and Mitigations", arXiv:2603.28846, 31 March 2026. <https://quantumai.google/static/site-assets/downloads/cryptocurrency-whitepaper.pdf>
- [5] PostQuantum.com, summary of the Google Quantum AI resource estimates for secp256k1. <https://postquantum.com/security-pqc/google-quantum-bitcoin-ecdlp/>
- [6] M. Mosca, M. Piani, "Quantum Threat Timeline Report 2025", Global Risk Institute, published 9 March 2026. <https://globalriskinstitute.org/publication/quantum-threat-timeline-report-2025b/>
- [7] K. Torpey, Gizmodo, 8 October 2026, on J. Drake's post of 7 October 2026 and the responses to it. <https://gizmodo.com/cryptographers-urgently-debating-whether-ai-could-break-bitcoins-security-in-months-2000823670>
- [8] Europol, "Europol urges early action to protect cryptocurrencies and sensitive data from quantum threats", press release, 7 October 2026. <https://www.europol.europa.eu/media-press/newsroom/news/europol-urges-early-action-to-protect-cryptocurrencies-and-sensitive-data-quantum-threats>
- [9] Forklog, coverage of the two Europol reports, 7 October 2026. <https://forklog.com/en/europol-outlines-quantum-threats-to-cryptocurrencies-and-encrypted-data/>
- [10] J. Van Straten, "Over 6 million bitcoin sit behind exposed public keys as AI warnings mount", CoinDesk, 8 October 2026 (Glassnode data). <https://coindesk.com/markets/2026/10/08/over-6-million-bitcoin-sit-behind-exposed-public-keys-as-ai-warnings-mount>
- [11] "Quantum Horizon", arXiv:2606.14484. <https://arxiv.org/html/2606.14484>
- [12] Deloitte Netherlands, perspective on quantum risk to the Ethereum blockchain (citing Project Eleven). <https://www.deloitte.com/nl/en/services/consulting-risk/perspectives/quantum-risk-to-the-ethereum-blockchain.html>
- [13] NIST IR 8547 (Initial Public Draft), "Transition to Post-Quantum Cryptography Standards", 12 November 2024. <https://csrc.nist.gov/pubs/ir/8547/ipd>
- [14] NSA, Commercial National Security Algorithm Suite 2.0, as summarised by Entrust (May 2024) and Encryption Consulting. <https://www.entrust.com/it/blog/2024/05/nsa-announces-update-to-commercial-national-security-algorithm-suite-2-0-and-quantum-computing-faq> ; <https://www.encryptionconsulting.com/education-center/what-is-cnsa-2-0/>
- [15] Executive Order 14412, "Securing the Nation Against Advanced Cryptographic Attacks", 22 June

2026. <https://www.presidency.ucsb.edu/documents/executive-order-14412-securing-the-nation-against-advanced-cryptographic-attacks>
- [16] Arbitrum Foundation forum, "Post-Quantum Cryptography Risk in Arbitrum Smart Contracts: A Technical Overview". <https://forum.arbitrum.foundation/t/post-quantum-cryptography-risk-in-arbitrum-smart-contracts-a-technical-overview/31027>
- [17] SKALE Network, xmss-solidity v1.1.0 (MIT), 8 October 2026. <https://github.com/skalenetwork/xmss-solidity>
- [18] SKALE Network, slhdsa-solidity (MIT). <https://github.com/skalenetwork/slhdsa-solidity>
- [19] Fireblocks, post-quantum signature verification on Ethereum (ML-DSA-44 verifier). <https://www.fireblocks.com/blog/post-quantum-signatures-ethereum-cheaper-gas>
- [20] ZKNox, ETHDILITHIUM (MIT). <https://github.com/ZKNoxHQ/ETHDILITHIUM>
- [21] ZKNox, ETHFALCON (MIT). <https://github.com/ZKNoxHQ/ETHFALCON>
- [22] A. Hülsing, D. Butin, S. Gazdag, J. Rijneveld, A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, May 2018. <https://www.rfc-editor.org/rfc/rfc8391> ; errata: <https://errata.rfc-editor.org/rfc8391>
- [23] D. Cooper, D. Apon, Q. Dang, M. Davidson, M. Dworkin, C. Miller, "Recommendation for Stateful Hash-Based Signature Schemes", NIST SP 800-208, October 2020. <https://doi.org/10.6028/NIST.SP.800-208>
- [24] XMSS reference implementation (CC0), commit 171ccbd, 16 March 2021. <https://github.com/XMSS/xmss-reference>
- [25] ethereum.org, "Quantum resistance" roadmap page, updated 8 September 2026. <https://ethereum.org/roadmap/security/quantum-resistance/>
- [26] Ethereum Foundation, "Introducing leanSig", <https://pq.ethereum.org/blog/introducing-leansig/> ; IACR ePrint 2025/1332, <https://eprint.iacr.org/2025/1332>
- [27] The Quantum Insider, on the Solana Winternitz Vault, 4 January 2025. <https://thequantuminsider.com/2025/01/04/solana-takes-a-step-toward-pqc-era-with-quantum-resistant-vault/>
- [28] The QRL, press release on Testnet V2. <https://www.theqrl.org/press/qrl-launches-testnet-v2-for-its-postquantum-evmfriendly-blockchain/>
- [29] "poqeth: Efficient post-quantum signature verification on Ethereum", ACM AsiaCCS 2025, IACR ePrint 2025/091. <https://eprint.iacr.org/2025/091>
- [30] BunkerERC, bunker (BunkerVault). <https://github.com/BunkerERC/bunker>
- [31] Qanary. <https://github.com/RaYYeR220/qanary>
- [32] M. Palatinus, P. Rusnak, A. Voisine, S. Bowe, "BIP-39: Mnemonic code for generating deterministic keys". <https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki>
- [33] H. Krawczyk, P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010. <https://www.rfc-editor.org/rfc/rfc5869>
- [34] L. Groot Bruinderink, A. Hülsing, "'Oops, I did it again': Security of One-Time Signatures under Two-Message Attacks", IACR ePrint 2016/1042. <https://eprint.iacr.org/2016/1042>
- [35] S. Fluhrer, IACR ePrint 2023/1905 (two-message attacks on Winternitz signatures, revisited). <https://eprint.iacr.org/2023/1905>
- [36] P. Murray, N. Welch, J. Messerman, "EIP-1167: Minimal Proxy Contract". <https://eips.ethereum.org/EIPS/eip-1167>
- [37] Ethereum Foundation, Kohaku wallet SDK, including @kohaku-eth/pq-account. <https://github.com/ethereum/kohaku>
- [38] EIP-7932 (Secondary Signature Algorithms), <https://eips.ethereum.org/EIPS/eip-7932> ; EIP-8051 (ML-DSA verification precompiles), <https://eips.ethereum.org/EIPS/eip-8051> ; EIP-8052 (Falcon-512

precompile), <https://eips.ethereum.org/EIPS/eip-8052>

[39] Robinhood Markets, Inc., Form 10-Q for the quarterly period ended June 30, 2026, risk factors.
<https://www.sec.gov/Archives/edgar/data/0001783879/000178387926000114/hood-20260630.htm>