# TzEL: Tezos Encrypted Ledger

Post-Quantum Shielded Transactions on Tezos Smart Rollups

TzEL contributors

**Abstract**

TzEL, the Tezos Encrypted Ledger, is a shielded transaction protocol for Tezos smart rollups. It is designed for a setting in which shielded transaction data is public and durable, so encrypted note payloads may be archived long before anyone knows whether the cryptography protecting them will survive future attacks. TzEL adapts the note/commitment/nullifier architecture of shielded payments to that threat by making the shielded path post-quantum end to end: incoming notes are delivered with ML-KEM, spends are authorized with one-time hash-based signatures inside an XMSS-style tree, and validity is proved with recursive STARKs. The protocol also separates spend authority from proof generation, allowing wallets to authorize exact transaction effects locally while outsourcing proof generation to an untrusted prover. Shielded transactions can move value from bridge deposits into notes, transfer value between notes, and release value back to public recipients. The result is a rollup-based shielded ledger whose archived note ciphertexts, spend authorizations, and proof system do not depend on classical discrete-log assumptions.

## 1 Introduction

Shielded payment systems publish encrypted transaction data on a public ledger. Those ciphertexts can be copied and stored indefinitely. If the cryptography protecting them is later broken, old notes, values, and memos can be decrypted retroactively. For a private payment system, that is the natural form of harvest-now-decrypt-later.

That observation creates two distinct questions. The first is *when* to migrate: if note ciphertexts can be harvested now and decrypted later, then the privacy layer matters before a quantum-capable adversary arrives. The second is *how far* to migrate: it is possible to harden only the memo layer and still leave other parts of the shielded path on assumptions that are not meant to survive quantum attacks. TzEL takes the broader route. It aims to make the shielded transaction path itself post-quantum.

A second requirement is practical rather than cryptographic. STARK proving is heavy enough that wallets may want to outsource proof generation. That should not require outsourcing spend authority. TzEL therefore separates the right to authorize a spend from the work of producing the proof that validates it.

## 2 Threat Model and Design Goals

### 2.1 Threat model

TzEL is designed for the following setting:

- Shielded transaction data, including encrypted note payloads, is public and can be archived indefinitely.

- Proof generation may be delegated to an untrusted prover.
- Users may want watch-only services that can help find incoming notes or account for sent outputs without gaining spend authority.
- The rollup enforces public state-transition rules, while the proof system enforces private consistency of notes, nullifiers, and spends.

Like other practical shielded ledgers, TzEL does not make all observable structure disappear. Transaction type, timing, ordering, number of spent inputs, and some output-shape information remain visible at the public layer. The protocol separates receiver-side recovery from sender-side recovery. Recipients use incoming viewing material to detect and decrypt notes addressed to them. Senders can separately export outgoing viewing material that recovers the metadata needed to recognize notes they created, without granting spend authority and without adding another public key to payment addresses.

### 2.2 Design goals

The main design goals are:

- Preserve the core shielded-ledger structure of notes, commitments, nullifiers, and historical Merkle roots.
- Replace non-post-quantum assumptions in the shielded path with primitives chosen for long-lived public ciphertexts and durable note archives.
- Keep spend authority in the wallet even when proof generation is delegated.
- Support reduced-capability exports such as detection-only and incoming-view monitoring and sender-side outgoing accounting.
- Fit the execution and data model of Tezos smart rollups and the Tezos Data Availability Layer (DAL) [17, 18].

## 3 Protocol Overview

A *note* is the protocol's private value object. On chain it appears as a commitment together with encrypted note data. Off chain, the recipient uses the viewing path and local address metadata to recover the note plaintext. To spend a note, the wallet reveals a *nullifier*: a one-use serial derived from the note and the note owner's secret spending material. The rollup rejects any nullifier that has already appeared. This follows the note, commitment, and nullifier lineage introduced by Zerocash and refined in the Zcash Sapling and Orchard protocols [1, 2, 3].

The rollup state therefore contains:

- an append-only Merkle tree of note commitments;
- a global set of spent nullifiers;
- a historical set of Merkle roots usable as spend anchors;

- a per-pool aggregated bridge-deposit balance, keyed by a wallet- derived pubkey hash (a hash of the deployment's auth domain plus the recipient's auth tree root and a per-deposit blind), drained only by shield;
- verifier configuration, public-account bindings, and fee-policy state.

A *shielded transaction* is any transaction that creates or destroys notes. TzEL has three of them:

| Transaction | Consumes | Creates |
|---|---|---|
| Shield | some balance from a bridge-deposit pool keyed by a wallet-derived pubkey hash | a user note and a DAL-producer fee note |
| Transfer | 1–7 shielded input notes | a recipient note, a change note, and a DAL-producer fee note |
| Unshield | 1–7 shielded input notes | a public credit, an optional change note, and a DAL-producer fee note |

Every shielded transaction pays two distinct resource prices. The first is a burned rollup fee $fee$, which prices rollup execution and is recreated nowhere. In the current reference rollup it has a floor of 100000 mutez: the first two private transactions in an inbox level pay the floor, each later private transaction doubles the required fee, the doubling is capped after six steps, and the fee resets when the inbox level advances. The second is a strictly positive private producer fee $producer\_fee$, created as an ordinary shielded note intended for the DAL slot producer. The rollup proves that such a note exists and has positive value; the producer enforces that it is payable to them as an inclusion policy before publishing. The burned fee and the producer fee price different resources and are therefore modeled separately.

# 4 Cryptographic Design

## 4.1 Post-quantum profile

| Role | Instantiation | Purpose |
|---|---|---|
| Commitments and nullifiers | BLAKE2s-derived field hashes | bind notes and prevent double spends |
| Incoming-note delivery | ML-KEM-768 + ChaCha20-Poly1305 | encrypt note payloads and memos |
| Detection | ML-KEM-768 clue + short tag | watch-only candidate filtering |
| Outgoing recovery | BLAKE2s-derived key + ChaCha20-Poly1305 | sender-side accounting for created outputs |
| Spend authorization | hash-based one-time signatures in an XMSS-style tree | authorize each spent note once |
| Zero knowledge | Cairo AIR + recursive STARKs | prove validity without exposing witness data |

Where the protocol can be expressed with hash-based machinery, it does so: commitments, nullifiers, Merkle nodes, owner binding, transaction sighashes, and one-time-signature chains are all built from BLAKE2s [14, 15]. That leaves one place where hashing alone is not enough. A sender must take a public payment address and, without interaction, produce note ciphertext that only the recipient can open. That is a public-key delivery problem, so TzEL uses ML-KEM for the incoming-note path [9, 10].

This is why the system is not purely hash-based. If there were a practical, standardized hash-based KEM with the right ergonomics for per-address note delivery and watch-only detection, the protocol could have used it. In the current ecosystem, the pragmatic post-quantum choice for that role is ML-KEM. Everywhere else, the design stays with hashes.

That still leaves a design choice. One could adopt ML-KEM for note encryption and stop there, while keeping spend authorization or proof verification on classical assumptions. TzEL takes the stronger route: the shielded path as a whole is meant to survive without elliptic-curve or pairing-based cryptography.

### 4.2 Proof architecture

The private transaction logic is first expressed as Cairo AIR. That first proof establishes the arithmetic and Merkle constraints for shield, transfer, or unshield. A second proving layer then re-proves that Cairo execution using Stwo, yielding the recursive proof object that is posted to the rollup [8, 5, 6, 7].

This two-layer structure serves two purposes. First, it keeps the proving stack entirely in STARK territory. Second, it yields a proof that is small and verifier-friendly enough for rollup use while still carrying zero-knowledge blinding. On the current reference stack, recursive proofs are around 300 KB; single-level proving exists as a development aid, not as the intended privacy profile.

## 5 Keys, Addresses, and Capabilities

The key hierarchy is organized around capability separation. TzEL distinguishes the following reduced-authority capabilities:

| Capability | What it can do |
|---|---|
| Detection | flag candidate incoming notes, with false positives, without reading note contents |
| Incoming view | decrypt and validate incoming notes and memos, but not spend them |
| Outgoing view | recover sender-created output metadata and values, but not spend them |
| Full view | incoming view plus spent-state tracking for notes whose address metadata is known |
| Spend | full view plus transaction authorization |

Detection, incoming view, full view, and spend form a nested receiver-side chain. Outgoing view is orthogonal: it tracks outputs created by the sender but does not detect arbitrary incoming notes, compute nullifiers, or authorize spends. The detection capability is related to fuzzy message detection, but TzEL uses an ML-KEM-based clue and short tag rather than directly adopting the FMD constructions [4].

That separation is reflected in the wallet derivation tree. From a master secret, the wallet derives independent spending, incoming, and outgoing branches:

$$\begin{aligned} spend\_seed &= \mathsf{H}(\text{TAG\_SPEND}, master\_sk), \\ incoming\_seed &= \mathsf{H}(\text{TAG\_INCOMING}, master\_sk), \\ outgoing\_seed &= \mathsf{H}(\text{TAG\_OUTGOING}, master\_sk). \end{aligned}$$

The spending branch contains the secrets needed to derive nullifiers and to authorize spends. The incoming branch contains the material needed to receive, detect, and view notes. This makes reduced-capability export natural: a user can hand detection or viewing material to another process without handing over spend authority.

Incoming and full-view wallets still need the local address records containing $d_j$, $auth\_root_j$, $pub\_seed_j$, and $nk\_tag_j$ to validate recovered notes. The incoming and nullifier branches alone do not reconstruct that public address metadata.

The outgoing branch is sender-scoped. It is not part of payment addresses and does not require an additional ML-KEM public key on chain. Instead, when a wallet creates an output, it encrypts a compact recovery record under a symmetric key derived from *outgoing_seed* and the output commitment $\mathsf{cm}$. This lets the sender later recognize and account for notes it created while preserving the recipient-facing address format.

For address index $j$, the wallet derives:

- $d_j$, the address diversifier;
- $nk\_spend_j$, the per-address nullifier secret;
- $nk\_tag_j = \mathsf{H}_{\text{nktg}}(nk\_spend_j)$, a public binding tag that lets a sender bind a note to the correct nullifier lineage without learning the nullifier secret;
- $(auth\_root_j, pub\_seed_j)$, the public root and seed of that address's one-time authorization tree;
- $(ek_j^v, dk_j^v)$, the ML-KEM viewing keypair;
- $(ek_j^d, dk_j^d)$, the ML-KEM detection keypair.

The payment address is therefore

$$address_j = (d_j, auth\_root_j, pub\_seed_j, nk\_tag_j, ek_j^v, ek_j^d).$$

It contains enough public information for a sender to create a spendable note for that address, but not enough information to derive nullifiers or spend authorizations.

These addresses are larger than classical shielded addresses because they carry multiple post-quantum public components. In practice, such addresses are better handled as structured payment requests, QR encodings, or directory lookups than as strings to copy by hand.

## 6 Notes, Commitments, and Nullifiers

For note value $v$, per-note seed $rseed$, and the recipient's address metadata, the protocol derives commitment randomness and the note commitment as in a shielded-note design descended from Sapling and Orchard [2, 3]:

$$\mathsf{rcm} = \mathsf{H}(\mathsf{H}(\mathrm{TAG\_RCM}), rseed),$$

$$owner\_tag_j = \mathsf{H}_{\mathrm{owner}}(auth\_root_j, pub\_seed_j, nk\_tag_j),$$

$$\mathsf{cm} = \mathsf{H}_{\mathrm{commit}}(d_j, v, \mathsf{rcm}, owner\_tag_j).$$

The owner tag is what fuses the two sides of future spending authority into the note commitment. It binds the note both to the authorization tree and to the nullifier derivation path. Without that binding, a commitment could be paired with unrelated nullifier material. The dependency chain is

$$nk\_spend_j \to nk\_tag_j \to owner\_tag_j \to \mathsf{cm}.$$

Nullifiers are position-dependent:

$$\mathsf{nf} = \mathsf{H}_{\mathrm{nf}}(nk\_spend_j, \mathsf{H}_{\mathrm{nf}}(\mathsf{cm}, pos)).$$

Including the Merkle leaf position prevents two otherwise identical commitments inserted at different leaves from collapsing to the same nullifier. That is the same basic issue sometimes described as a faerie-gold problem: identical-looking notes at different positions must still spend distinctly.

# 7 Spend Authorization and Delegated Proving

## 7.1 Why one-time authorization is needed

If proof generation is delegated, the wallet needs a way to authorize the exact public effects of a transaction without handing the prover a reusable spending key. TzEL solves that with an XMSS-style authorization tree of one-time hash-signature keys. Each spent input consumes one unused leaf. The wallet signs the transaction locally; the prover only proves, inside the STARK, that the signature is valid under the address's authorization root [11, 12, 13].

This mechanism is still useful when proving is done locally. It keeps spend authorization inside the proof and avoids publishing public keys or signatures on chain.

## 7.2 Authorization tree and in-circuit verification

Each address owns a depth-16 authorization tree, giving $2^{16}$ one-time authorization leaves. The per-leaf signature is WOTS-like, with the current parameterization using base $w = 4$, 133 chains, and chain length $w - 1$.

For each spent input, the wallet signs a transaction-specific sighash with the corresponding one-time secret. Inside the STARK, the circuit

1. recomputes the sighash from the public outputs;
2. decomposes it into base-4 WOTS digits;
3. hashes the signature chains forward to recover the corresponding public key endpoints;
4. compresses those endpoints through the XMSS-style L-tree;
5. proves membership of the recovered leaf under $auth\_root_j$.

No authorization leaf, one-time public key, or one-time signature appears in the public transaction outputs. The proof itself is the public authorization object.

That matters for privacy as well as correctness. Publishing authorization leaves or signatures would create another public handle linking spends to address material. Instead, the public layer sees only the commitments, nullifiers, note-data hashes, and other public effects that the rollup actually needs.

### 7.3 What the prover can and cannot do

The wallet, not the prover, chooses the transaction effects. It selects the inputs, computes the output commitments, derives the note-data hashes, forms the public statement, and signs the sighash locally. These hashes bind the posted recipient-delivery ciphertexts and outgoing-recovery ciphertexts as bytes; the circuit does not prove that those ciphertexts decrypt to the commitment preimages. The prover receives only the witness needed to prove that statement.

Because spend authorization is checked inside the circuit over the public outputs, an untrusted prover cannot change recipients, fees, nullifiers, commitments, withdrawal targets, or note-data hashes without invalidating the proof. This is the core separation between delegated proving and delegated spending.

### 7.4 Replay binding

For transfer and unshield, the signature covers a folded transaction sighash that includes

- a circuit-type tag;
- the deployment authorization domain $auth\_domain$;
- the input root;
- all nullifiers;
- public values such as fees, withdrawal amount, and the canonical L1 recipient hash;
- output commitments and note-data hashes, including the hashes of recipient-delivery and outgoing-recovery ciphertexts.

The type tag prevents cross-circuit replay. The authorization domain prevents replay across mirrored deployments, forks, or verifier migrations that might otherwise share root history. The current base protocol does not yet include an expiry or per-transaction nonce in that authorization, so a delegated prover can withhold a completed authorization until one of its nullifiers is consumed elsewhere.

## 8 Shielded Transaction Types

### 8.1 Shield

Shield takes value that has already crossed from L1 into the rollup and moves it into the shielded pool. The public rollup object being consumed is a *deposit pool* keyed by a wallet-derived pubkey hash:

$$pubkey\_hash = \mathsf{H}_{pubkey}(auth\_domain,\ auth\_root,\ auth\_pub\_seed,\ blind).$$

$\mathsf{H}_{pubkey}$ is the sequential left-fold using the `sighSP__` personalization with leading type tag $0x04$ (domain-separated from transfer / unshield / shield sighashes, which use tags $0x01$, $0x02$, $0x03$). The L1 deposit transaction is addressed to the canonical recipient string `deposit:<hex(pubkey_hash)>`; the kernel maintains a per-pool aggregated balance and credits it on each L1 ticket. *blind* is a per-deposit randomness the wallet derives deterministically from *master_sk*, so distinct deposits to the same auth tree get distinct pool ids by default. The blind derivation is reproducible from the seed, but the (address_index, deposit_nonce) pairs the wallet actually used live only in the wallet's local state — the rollup kernel intentionally stores no enumerable pool index. A wallet that loses its local file before shielding therefore cannot discover its pools without a bounded brute-force scan over candidate $(i, j)$ pairs followed by per-candidate kernel balance probes.

**Aggregation and dust resistance.** Multiple L1 tickets to the same `deposit:<hex(pubkey_hash)>` recipient string aggregate into one balance. The `deposit:<hex(pubkey_hash)>` string is public on L1: an attacker observing a victim's deposit and dust-depositing the same recipient string just adds to the victim's pool balance. Mirror- depositing is therefore a donation rather than a brick — the attacker pays mutez that only the victim (sole holder of the auth tree's signing material) can later shield-and-spend.

The public outputs of shield are

$$[auth\_domain,\ pubkey\_hash,\ v_{pub},\ fee,\ producer\_fee,\ \mathsf{cm}_{user},\ \mathsf{cm}_{producer},\ mh_{user},\ mh_{producer}].$$

The circuit proves the user commitment, the producer-fee commitment, the condition $producer\_fee > 0$, and *verifies an in-circuit WOTS+ signature* from the recipient's auth tree (the same scheme used by transfer and unshield). The signed message is the shield sighash:

$$\mathsf{H}_{sigh}(0x03,\ auth\_domain,\ pubkey\_hash,\ v_{pub},\ fee,\ producer\_fee,\ \mathsf{cm}_{user},\ \mathsf{cm}_{producer},\ mh_{user},\ mh_{producer}).$$

The kernel reads $balance = \text{pool}[pubkey\_hash]$, rejects unless $balance \geq v_{pub} + fee + producer\_fee$, debits the pool, and appends both notes. Pool overfunding is fine — the surplus stays available for future shields. The user picks $(v_{pub}, fee, producer\_fee)$ at shield time, so fee revisions between deposit and shield are a wallet-side retry, not a stranded slot.

**Untrusted-prover safety.** The witness exposes the recipient's address fields, value, randomness, and the encrypted-note bytes. None of these grants spend authority because (a) the in-circuit WOTS+ signature binds the entire request payload, and (b) the prover does not have the wallet's auth-tree signing material. Delegated proving requires the wallet to sign each request — the same operational model as transfer / unshield.

**Verifier configuration is one-shot.** Because *pubkey_hash* commits to *auth_domain*, any reconfiguration between the wallet computing *pubkey_hash* and the L1 ticket landing would silently strand the deposit. The kernel therefore makes the entire verifier configuration one-shot for the lifetime of the deployment: once a signed `configure_verifier` message is accepted, every subsequent `configure_verifier` is rejected — same fields, different *auth_domain*, different program hashes, anything. Wallets that read *auth_domain* once therefore never have to worry about it changing under them.

**Producer-fee receiver is a publisher-side concern.** The producer-fee note recipient is a private input; the circuit only proves that $\mathsf{cm}_{producer}$ commits to some $producer\_otag = \mathsf{H}_{owner}(auth\_root, pub\_seed, nk\_tag)$ and that the note has positive value. The producer fee is paid to whichever DAL slot publisher includes the transaction. DAL slot publishing is a permissionless market: any participant can pay bakers to publish a slot and keep the spread between producer fees they collect and the L1 inclusion costs they pay. Routing of $producer\_otag$ is enforced at the publisher, who only bundles transactions whose producer note is payable to an address they control — a wallet that routes fees elsewhere simply does not get bundled.

**Top-ups and partial drains.** A user can send multiple L1 tickets to the same `deposit:<hex(pubkey_hash)>` recipient; each ticket adds to the pool's balance. Each shield then debits some $(v_{pub} + fee + producer\_fee)$ at the user's discretion. Shielding is not exclusive of further deposits or further shields: pools live as long as their balance is positive.

**Ownership binding accounting.** $v_{pub}$ becomes the user's shielded note, $producer\_fee$ becomes the DAL-producer fee note, and $fee$ is burned. There is no separate direct-withdraw path for the deposit pool; only Shield can drain it. The pool's signing material lives in the wallet's auth tree; a third party who observes the L1 deposit on chain cannot construct a valid shield without the auth tree.

### 8.2 Transfer

Transfer consumes $N$ shielded notes, where $1 \leq N \leq 7$, and creates three new shielded notes: recipient, change, and DAL-producer fee. Its public outputs are

$$[auth\_domain, root, \mathsf{nf}_0, \ldots, \mathsf{nf}_{N-1}, fee, \mathsf{cm}_1, \mathsf{cm}_2, \mathsf{cm}_3, mh_1, mh_2, mh_3].$$

For each input, the circuit proves commitment reconstruction, Merkle membership, nullifier computation, and one-time authorization under the corresponding authorization root. For the outputs, it proves all three commitments and enforces

$$\sum_{i=0}^{N-1} v_i = v_1 + v_2 + v_3 + fee,$$

with $v_3 > 0$ for the producer-fee note.

The rollup state transition checks the public nullifier list before applying the transfer. After validating and stripping the bootloader's Cairo array length prefix, the verified public-output vector must have exactly the length implied by $N$, the published nullifiers must be pairwise distinct, and none of them may already appear in the global nullifier set.

### 8.3 Unshield

Unshield consumes $N$ shielded notes, releases public value to a recipient, and creates a producer-fee note plus an optional shielded change note. Its public outputs are

$$[auth\_domain, root, \mathsf{nf}_0, \ldots, \mathsf{nf}_{N-1}, v_{pub}, fee, recipient\_id, \mathsf{cm}_{change}, mh_{change}, \mathsf{cm}_{fee}, mh_{fee}].$$

The circuit proves the same input validity and authorization conditions as transfer, then enforces

$$\sum_{i=0}^{N-1} v_i = v_{pub} + v_{change} + v_{fee} + fee.$$

If there is no change note, the change commitment, change note-data hash, and all corresponding witness values are constrained to zero. That removes a source of prover malleability: the prover cannot hide nonzero private change behind a zero public commitment.

The rollup applies the same public nullifier checks as transfer: after validating and stripping the bootloader's Cairo array length prefix, the verified public-output vector length must match $N$, the published nullifiers must be pairwise distinct, and every published nullifier must be fresh with respect to the global nullifier set.

## 9 Encrypted Note Delivery and Detection

Each output note carries

- the commitment $\mathsf{cm}$;
- a detection ciphertext $ct_d$;
- a short detection tag;
- a viewing ciphertext $ct_v$;
- a derived AEAD nonce;
- encrypted note plaintext $(v \parallel rseed \parallel memo)$;
- an outgoing recovery ciphertext $ct_o$.

For recipient address $(ek_j^d, ek_j^v)$, the sender creates two ML-KEM encapsulations. The viewing path is the strong one: encapsulate to $ek_j^v$, derive a symmetric key from the shared secret, and encrypt $(v \parallel rseed \parallel memo)$ with ChaCha20-Poly1305 [9, 10, 16]. The detection path is weaker on purpose. It gives a watch-only service just enough material to flag candidate incoming notes, while leaving note contents encrypted and spend authority untouched.

This is the operational role of detection. Without it, a wallet or service would have to attempt full incoming-note recovery against every output on the chain. With detection, a lightweight indexer can scan the chain, report likely matches by address index, and leave the wallet to do full note recovery only on that narrowed candidate set [4].

Detection leaks very little compared with viewing. A detector can test whether a note is a candidate for one of the monitored addresses. It cannot decrypt note contents, derive nullifiers, authorize spends, or directly infer spent status. False positives are built in by design.

The outgoing recovery path is separate from recipient delivery. For each output, the sender encrypts

$$(role \parallel v \parallel rseed \parallel d_j \parallel auth\_root_j \parallel pub\_seed_j \parallel nk\_tag_j)$$

under a key derived from $outgoing\_seed$ and $\mathsf{cm}$. The plaintext omits spending secrets. A wallet using outgoing viewing material accepts a recovered record only after recomputing

$$\mathsf{cm}_{expected} = \mathsf{H}_{\text{commit}}(d_j, v, \mathsf{H}(\mathsf{H}(\text{TAG\_RCM}), rseed), \mathsf{H}_{\text{owner}}(auth\_root_j, pub\_seed_j, nk\_tag_j))$$

and checking $\mathsf{cm}_{expected} = \mathsf{cm}$. Thus outgoing viewing is an accounting and recovery feature for the sender, not a new authorization path.

Detection is only a filtering aid, not a correctness condition. A wallet accepts an incoming note only after it

1. decrypts the note plaintext;
2. selects the matching local address record containing $(d_j, auth\_root_j, pub\_seed_j, nk\_tag_j)$;
3. recomputes $\mathsf{rcm} = \mathsf{H}(\mathsf{H}(\text{TAG\_RCM}), rseed)$;
4. recomputes $owner\_tag_j = \mathsf{H}_{\text{owner}}(auth\_root_j, pub\_seed_j, nk\_tag_j)$;
5. recomputes $\mathsf{cm}_{expected} = \mathsf{H}_{\text{commit}}(d_j, v, \mathsf{rcm}, owner\_tag_j)$;
6. accepts the note only if $\mathsf{cm}_{expected}$ equals the posted commitment.

These acceptance rules matter because the proof binds the posted ciphertext bytes through note-data hashes, but does not prove that the viewing or outgoing ciphertexts decrypt to the same commitment preimage used inside the circuit.

## 10 Rollup Verification and DAL

The STARK proves private consistency. The rollup still has to enforce public state-transition rules that are not handled inside the proof. These include:

- proof verification for the intended executable;
- *auth_domain* matching the configured deployment domain for spending transactions;
- the input root being a recognized historical root;
- exact agreement between the verified public-output length and the number of published nullifiers;
- pairwise distinctness and global freshness of every published nullifier;
- equality between posted note commitments and the commitments in the proof outputs;
- equality between posted note data and the note-data hashes in the proof outputs;
- correct binding of shield to the deposit-pool balance being drained (the kernel pins the proof's *pubkey_hash* to the request and verifies the pool's balance is at least $v_{pub} + fee + producer\_fee$);
- correct binding of unshield to the intended canonical L1 recipient string;
- enough remaining commitment-tree capacity for every output note before any balance, nullifier, or outbox mutation is applied;
- satisfaction of the current burned-fee policy.

These checks must be implemented and should be treated as part of the protocol. Without historical-root validation, a prover could spend against a fabricated local tree. Without executable binding, a verifier could accept the wrong Cairo task. Without note-data binding, a relayer could swap ciphertext fields after the proof was produced.

Verifier parameters are not supplied by the transaction. The rollup infers the expected circuit from the operation type, checks the corresponding configured executable hash, and derives the verifier inputs from the proof output preimage.

The large proof and note payloads are why the Tezos DAL is structurally part of the design. The DAL is a companion data-availability network for rollups: it lets rollups publish large blobs without forcing them into L1 block bandwidth. That matters here because recursive proofs are roughly 300 KB and each encrypted output note is roughly 3 KB. The intended path is therefore to publish the heavy payload through DAL, inject only a small pointer into the rollup inbox, and let the kernel resolve that pointer before applying the state transition [17, 18].

TzEL's post-quantum claims concern the shielded transaction layer. Tezos L1 consensus, bridge mechanics, and DAL attestation are separate systems with their own security assumptions. Even before those layers reach a fully post-quantum profile, protecting note ciphertexts, spend authorizations, and proof objects at the shielded layer still blocks the harvest-now-decrypt-later attack against archived shielded data.

# 11 Security Properties and Limitations

## 11.1 What the design gives you

- **Balance conservation.** Values are range-checked and the value equations for shield, transfer, and unshield are enforced inside the circuit.
- **Double-spend resistance.** Nullifiers are unique per note-position pair. The circuit binds each public nullifier to the corresponding private input witness, and the rollup enforces that the public nullifier list has no duplicates and is disjoint from the global spent-nullifier set before applying the transaction.
- **Commitment binding.** The chain $nk\_spend \to nk\_tag \to owner\_tag \to \mathsf{cm}$ ties each note to the nullifier lineage that will later be used to spend it.
- **Local spend control.** A valid spend proof implies both note ownership and a valid one-time authorization over the exact public effects of the transaction.
- **No public authorization artifacts.** Authorization leaves, one-time public keys, and one-time signatures do not appear in the public transaction outputs.
- **Post-quantum shielded path.** Incoming-note delivery, spend authorization, and zero-knowledge proving do not depend on classical discrete-log assumptions.

## 11.2 What still remains visible

- Transaction type, ordering, and timing are public.
- The number of published nullifiers reveals the number of spent inputs.
- Observers can tell whether an unshield carries a change note.

- A delegated prover sees per-address witness material such as $nk\_spend_j$ and $auth\_root_j$. Combined with public commitments, positions, and the global nullifier set, that is enough to infer spent-state for notes associated with that address.

Users who want delegated proving with weaker witness exposure can treat TEE-based provers or equivalent isolation as an operational mitigation, but that is not a protocol guarantee.

### 11.3 Current caveats

- **Detection is honest-sender.** A malicious sender can post bogus detection data for a note. That does not let them steal funds or read wallet state; it simply forces the recipient or its watch service to fall back to broader viewing-key scanning. Detection should therefore be understood as a performance optimization, not as the source of note correctness.
- **Ciphertext correctness is not proved in-circuit.** Wallets must recompute commitments before treating incoming or outgoing recovered records as valid.
- **Address fields are not self-authenticating to the sender.** Shield and transfer outputs can be formed with malformed public address metadata, producing unspendable notes. This is sender self-griefing, not theft.
- **One-time keys must still be treated as one-time.** Reuse does not create a public on-chain linking issue by itself because signatures are not published. But once the same external prover sees two authorizations under the same one-time key, it may be able to forge further authorizations. Wallets should therefore allocate one leaf per spend and persist that state durably.
- **Shield is signed by the auth tree and aggregates per pool.** The shield circuit verifies an in-circuit WOTS+ signature from the recipient's auth tree, binding every prover-rewritable field — value, fees, recipient and producer commitments, ciphertext memo hashes — under the same scheme used for transfer and unshield. Cross-deployment replay fails because $auth\_domain$ is in $pubkey\_hash$; prover-side redirection fails because the prover does not have the wallet's signing material. The cost is that shielding is no longer stateless on the wallet side: each shield consumes one WOTS+ key from the recipient's auth tree. Top-ups and partial drains are first-class — multiple L1 tickets to the same pool aggregate, and one pool can be drained by multiple shields.

## 12 Size and Data Path

Each encrypted output note is roughly 3487 bytes on chain:

- 32 bytes for the commitment;
- 1088 bytes for the detection ciphertext;
- 2 bytes for the detection tag;
- 1088 bytes for the viewing ciphertext;
- 12 bytes for the AEAD nonce;

- 1080 bytes for the encrypted $(v, rseed, memo)$ payload;
- 185 bytes for the outgoing recovery ciphertext.

Typical transaction sizes are therefore dominated by the recursive proof rather than by authorization artifacts:

| Transaction | Output note data | Approximate total |
|---|---|---|
| Shield | 2 notes | about 302 KB |
| Transfer | 3 notes | about 305 KB plus 32 bytes per input nullifier |
| Unshield | 1–2 notes | about 299–302 KB plus 32 bytes per input nullifier |

Those numbers are why DAL is not an optional add-on for this design. A rollup that wants shielded transactions with recursive proofs needs a high-bandwidth data path, not only a small inbox.

## 13 Conclusion

TzEL preserves the core shielded-ledger logic of notes, commitments, nullifiers, and Merkle-root-anchored spends, but retools the privacy layer for a post-quantum setting and a rollup execution model. Hash-based constructions cover commitments, nullifiers, and one-time spend authorization wherever they can. ML-KEM is used where the protocol genuinely needs public-key delivery of incoming notes and watch-only detection. Recursive STARKs replace pairing-based proof systems while keeping spend authorization inside the proof itself.

The result is a shielded transaction protocol for Tezos smart rollups that aims to protect the data most exposed to harvest-now-decrypt-later pressure: durable note ciphertexts, spend authorizations, and the proof path that validates private state transitions.

## References


[1] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. IEEE Symposium on Security and Privacy, 2014. https://eprint.iacr.org/2014/349.

[2] Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification. Electric Coin Company. https://zcash.readthedocs.io/en/master/rtd_pages/protocol.html.

[3] Daira-Emma Hopwood, Jack Grigg, Sean Bowe, Kris Nuttycombe, and Ying Tong Lai. ZIP 224: Orchard Shielded Protocol. https://zips.z.cash/zip-0224.

[4] Gabrielle Beck, Julia Len, Ian Miers, and Matthew Green. Fuzzy Message Detection. ACM CCS, 2021. https://eprint.iacr.org/2021/089.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptology ePrint Archive, Report 2018/046. https://eprint.iacr.org/2018/046.

[6] Ulrich Haböck, David Levit, and Shahar Papini. Circle STARKs. IACR Cryptology ePrint Archive, Report 2024/278. https://eprint.iacr.org/2024/278.

[7] StarkWare. Stwo: a next-generation Circle STARK prover and verifier. https://github.com/starkware-libs/stwo.

[8] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. IACR Cryptology ePrint Archive, Report 2021/1063. https://eprint.iacr.org/2021/1063.

[9] National Institute of Standards and Technology. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. 2024. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.203.pdf.

[10] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber: a CCA-secure module-lattice-based KEM. IACR Cryptology ePrint Archive, Report 2017/634. https://eprint.iacr.org/2017/634.

[11] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. RFC 8391: XMSS: eXtended Merkle Signature Scheme. Internet Engineering Task Force, 2018. https://datatracker.ietf.org/doc/html/rfc8391.

[12] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS – A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. PQCrypto, 2011. https://eprint.iacr.org/2011/484.

[13] Andreas Hülsing. WOTS+ – Shorter Signatures for Hash-Based Signature Schemes. IACR Cryptology ePrint Archive, Report 2017/965. https://eprint.iacr.org/2017/965.

[14] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. RFC 7693: The BLAKE2 Cryptographic Hash and Message Authentication Code. Internet Engineering Task Force, 2015. https://datatracker.ietf.org/doc/html/rfc7693.

[15] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. IACR Cryptology ePrint Archive, Report 2013/322. https://eprint.iacr.org/2013/322.

[16] Y. Nir and A. Langley. RFC 8439: ChaCha20 and Poly1305 for IETF Protocols. Internet Engineering Task Force, 2018. https://datatracker.ietf.org/doc/html/rfc8439.

[17] Tezos documentation. Smart Rollups. https://docs.tezos.com/architecture/smart-rollups.

[18] Tezos documentation. The Data Availability Layer. https://docs.tezos.com/architecture/data-availability-layer.