

metaspace white paper

Quantum-resistant cold storage for Solana

A vault locked by a SHA-256 commitment and opened with a Winternitz one-time signature, so that no Ed25519 key stands between a balance and the world.

October 2026 metaspace.fi

Abstract

A Solana account is an Ed25519 keypair, and its address is the public key itself. A large enough quantum computer running Shor's algorithm derives the private key from the public one, so every address that has appeared on chain is exposed, and the exposure cannot be withdrawn. metaspace is a vault program for Solana that authorises spending without checking a keypair signature. A vault holds SOL behind a 32-byte SHA-256 commitment. To withdraw, the owner reveals a Winternitz one-time signature, 34 hash chains and 1,088 bytes, over a digest that binds the vault, its nonce, the amount, the destination and the next commitment. The program walks the chains to their endpoints, checks them against the commitment, pays the destination and rotates the lock to a fresh commitment in the same instruction. The one assumption is that SHA-256 cannot be inverted, which Grover's algorithm reduces to about 2^{128} work. This paper gives the construction, its parameters, the withdrawal protocol, the recovery model and the security argument, and places the design beside NIST's hash-based signature standards and its draft timeline for retiring EdDSA.

1. The problem

1.1 A wallet is one key

A Solana account is an Ed25519 keypair. The private key is a scalar a ; the public key is a point on Curve25519,

$$A = a \cdot B,$$

where B is the curve's base point. Recovering a from A is the elliptic-curve discrete logarithm problem. On classical hardware it costs about 2^{126} operations, which is why Ed25519 holds today. Everything a wallet holds, every token account it owns and every authority it has been granted rests on that single assumption.

1.2 The address is the public key

A Solana address is not a hash of the key. It is the 32-byte encoding of A , written in base58. The moment an address appears on chain, whether it signed a transaction or only received one, its public key is public, and it stays public. There is no later date at which an owner can decide to stop having published it. Moving funds to a fresh address publishes a fresh key as soon as that address is used.

1.3 Shor

Shor's algorithm solves the discrete logarithm problem in polynomial time on a large enough quantum computer. It reads a off A . It does not weaken Ed25519 by some number of bits. It removes it. Because the public key is the address, no Solana balance, token authority, upgrade authority or stake authority is out of reach of that computation once it exists.

1.4 The NIST timeline

NIST IR 8547, in its initial public draft of November 2024, sets out the transition away from the public-key algorithms Shor breaks. Its Table 2 lists EdDSA at 128 bits of security as disallowed after 2035. Ed25519, which every Solana wallet uses, is EdDSA at that strength. The same document points to the standards NIST has already published for what replaces it: SP 800-208 approves the stateful hash-based schemes LMS and XMSS, and FIPS 205 specifies the stateless SLH-DSA. All three are built from Winternitz-style one-time signatures, and NIST states that their security relies only on the hash function.

1.5 Why a multisig does not help

A multisig mitigates key theft, which is the common failure, and it is the responsible thing to do against that failure. It mitigates nothing here. Five Ed25519 keys fail the same way one does, at the same time, for the same reason. A hardware wallet protects the key from the computer, not from the mathematics. Every other Solana vault, multisig and timelock authorises spending by checking an Ed25519 signature, so all of them share the one assumption, and the assumption is the thing that breaks.

2. Design goals

metaspace replaces the assumption rather than adding to it. The design is held to the following.

- **No keypair on the path that moves value.** No instruction that pays out checks a signer against anything. The wallet that created a vault is recorded as a hint for the interface, never consulted for authorisation.
- **One assumption, and the right one.** Spending is authorised by revealing values that hash to a stored commitment. The assumption is that SHA-256 cannot be inverted. Grover's algorithm takes a 256-bit preimage search to roughly 2^{128} operations, which is not a threat anyone has a credible plan for.
- **A signature authorises exactly one withdrawal.** The revealed material must be useless to anyone who sees it in the mempool, because everything that could be substituted is bound into the signed digest. If this ever failed, the reveal would be a bearer instrument, and the mechanism would fail completely.
- **The lock rotates in the instruction that spends it.** A one-time key signs once. There is no path that pays without replacing the commitment.

- **Nothing an operator can do.** No admin, no upgrade path written into the program, no pause, no oracle, no fee inside the program, and no `close_vault`. A vault whose operator can intervene is a vault with a keypair again, just further away.
- **It fits Solana as it is.** A transaction carries 1,232 bytes and 1.4 million compute units. The signature size and the verification schedule are chosen to live inside those limits, with the work spread across transactions rather than squeezed into one.
- **Easy in, guarded out.** Anyone may fund any vault with a plain system transfer. The entire security budget is spent on the way out.
- **Reviewable rather than clever.** Plain hash chains, a checksum that is a subtraction, a commitment that is one hash. A one-time signature scheme has a small number of ways to be subtly wrong, and the construction is written so that each of them can be checked by reading it.
- **Recovery from a seed alone.** Twelve words derive every key the vault will ever use. There is no other path in, and the product says so before the first deposit.

3. The vault

3.1 Accounts

The program has two account types. A `Vault` holds the balance and the lock. A `WithdrawalRequest` holds one in-flight withdrawal. Both are Anchor accounts, so each begins with an 8-byte discriminator; the sizes below are the Borsh encoding that follows it.

Vault seeds

```
seeds = [b"vault", owner_hint.as_ref(), label.as_ref()]
```

Field	Type	Bytes	Meaning
commitment	<code>[u8; 32]</code>	32	Where the lock is. The only thing between the balance and the world.
nonce	<code>u64</code>	8	Increments on every completed withdrawal. Part of the signed digest.
owner_hint	<code>Pubkey</code>	32	Lets a wallet find its vaults. Never consulted when authorising anything.
label	<code>[u8; 16]</code>	16	A seed, so one owner can hold several vaults.
created_at	<code>i64</code>	8	Unix time the vault was opened.
rotations	<code>u64</code>	8	How many times the lock has rotated.
bump	<code>u8</code>	1	PDA bump.

A vault is $8 + 105 = 113$ bytes. What it does not hold is the point: there is no authority key on the withdrawal path. The vault stores a 32-byte commitment rather than the 1,088-byte public key it commits to, which is why a vault costs 32 bytes of lock rather than a kilobyte.

WithdrawalRequest seeds

```
seeds = [b"request", vault.key().as_ref(), &vault.nonce.to_le_bytes()]
```

Field	Type	Bytes	Meaning
<code>vault</code>	<code>Pubkey</code>	32	The vault being withdrawn from.
<code>nonce</code>	<code>u64</code>	8	The vault nonce the request was opened against.
<code>amount</code>	<code>u64</code>	8	Lamports to pay.
<code>destination</code>	<code>Pubkey</code>	32	Who is paid. A wallet, a PDA or a program-owned account.
<code>next_commitment</code>	<code>[u8; 32]</code>	32	What the lock becomes once this withdrawal lands. Signed over.
<code>chain_endpoints</code>	<code>Vec<[u8; 32]></code>	$4 + 34 \times 32 = 1,092$	Up to 34 endpoints, filled as chunks arrive.
<code>chains_filled</code>	<code>u8</code>	1	How many chains have been accepted. Doubles as the cursor.
<code>opened_at</code>	<code>i64</code>	8	Unix time the request was opened. Expiry is measured from it.
<code>payer</code>	<code>Pubkey</code>	32	Who paid the rent, and who gets it back.
<code>bump</code>	<code>u8</code>	1	PDA bump.

A request is $8 + 1,246 = 1,254$ bytes. It is seeded on the vault's current nonce, so exactly one request can exist per vault state, and it is closed, with its rent returned to the payer, when the withdrawal finalises or the request is cancelled.

3.2 Instructions

Six instructions, in two groups: getting value in, and getting value out.

Instruction	What it does	Who may call it
<code>init_vault</code>	Opens a vault behind a commitment.	Anyone. The payer is recorded as a hint, not an authority.
<code>deposit</code>	Transfers lamports into a vault.	Anyone, into any vault.
<code>open_withdrawal</code>	Writes down the amount, destination and next commitment.	Anyone who pays the rent.
<code>push_signature</code>	Delivers up to 8 chains of the signature and verifies each walk.	Anyone.
<code>finalize_withdrawal</code>	Checks the commitment, pays out, rotates the lock.	Anyone. No signer is required.
<code>cancel_withdrawal</code>	Reclaims an expired or superseded request.	Anyone, once the request has expired or been superseded.

Deposits are deliberately permissive. A deposit is a system transfer into a program-derived address and needs no commitment, no signature over anything and no permission. There is no `close_vault`: a vault whose commitment nobody holds the preimage for is indistinguishable, on chain, from one whose owner is away, and an instruction that could retire the first could steal from the second. Emptying a vault is a withdrawal like any other, and the rent-exempt minimum stays behind.

3.3 Vault types

One lock, pointed at three different things a key can hold.

Personal. SOL behind a commitment. Withdraw any amount at any time. The lock rotates on every withdrawal, so nothing an observer sees in a landed transaction helps them with the next one. Token support follows the SOL path with the same account surface, a vault-owned token account per mint; the signed digest then binds the mint and the token amount as well, which moves its version tag from `metaspace:withdraw:v1` to `v2`.

Authority. A project moves its upgrade authority, mint authority and treasury authority into a vault instead of a keypair or a multisig. The vault then signs those operations as a PDA, under a commitment: `set_upgrade_authority`, an SPL mint authority transfer, a treasury spend. The asset at risk is not a balance but the program itself. Whoever holds an upgrade authority can replace the program and take everything every user has ever approved, and a multisig of Ed25519 keys does not change that.

Staking. A stake account has two authorities, staker and withdrawer, and the withdrawer is the one that matters. Putting it behind a commitment protects the principal while delegation continues and yield keeps accruing. It is the same vault with a different authority type.

4. The lock

The lock is a Winternitz one-time signature over SHA-256, implemented in `hashlock.rs`. Write H for SHA-256 and H^k for k applications of it:

$$H^0(x) = x, \quad H^k(x) = H(H^{k-1}(x)).$$

4.1 Parameters

Name	Value	In the source
Winternitz parameter w	256	<code>WINTERNITZ_W</code>
Message chains ℓ_1	32	<code>MESSAGE_CHAINS</code>
Checksum chains ℓ_2	2	<code>CHECKSUM_CHAINS</code>
Total chains ℓ	34	<code>TOTAL_CHAINS</code>
Chain length $w - 1$	255	<code>CHAIN_LENGTH</code>
Signature size	$34 \times 32 = 1,088$ bytes	<code>SIGNATURE_BYTES</code>
Commitment size	32 bytes	<code>Vault.commitment</code>
Chains per push	8	<code>CHAINS_PER_PUSH</code>
Request expiry	86,400 seconds	<code>REQUEST_EXPIRY_SECONDS</code>

$w = 256$ means one chain per byte of a 32-byte digest. The choice is forced by Solana's 1,232-byte transaction limit, not by cryptography. A Lamport signature over 256 bits is $256 \times 32 = 8,192$ bytes, six times what a transaction can carry. At $w = 16$ there are 64 message chains and 3 checksum chains, 2,144 bytes, still too large. At $w = 256$ the signature is 1,088 bytes and fits, with nothing to spare, and only because the verification is moved out of the transaction that carries it and spread over several.

4.2 Keys and the commitment

A private key is 34 random 32-byte values, one per chain, derived off chain from the seed. The public key is each value hashed 255 times.

$$sk = (x_1, \dots, x_{34}), \quad x_i \in \{0, 1\}^{256}$$

$$pk_i = H^{255}(x_i), \quad i = 1, \dots, 34$$

The vault does not store the 1,088-byte public key. It stores one hash of it, under a domain tag, so a vault costs 32 bytes rather than 1,088. This is the commitment.

$$C = H(\text{metaspace:commitment:v1} \parallel pk_1 \parallel \dots \parallel pk_{34})$$

The tag is the 23-byte ASCII string. $\parallel$ is concatenation. Computed by `commitment_of`.

Commitment preimage, $23 + 34 \times 32 = 1,111$ bytes

```
"metaspace:commitment:v1" || pk[0] || pk[1] || ... || pk[33]
commitment = SHA-256(preimage)
```

The tag is there because a bare concatenation hash is the kind of thing that becomes a problem the moment a second version of the format exists.

4.3 Encoding a digest

The thing signed is a 32-byte digest $d = (d_1, \dots, d_{32})$, one byte per message chain. The first 32 chain lengths are the bytes themselves.

$$m_i = d_i, \quad 0 \leq m_i \leq 255, \quad i = 1, \dots, 32$$

The checksum is a plain subtraction, so the argument in 4.6 can be checked by eye.

$$c = 32 \cdot 255 - \sum_{i=1}^{32} m_i = 8160 - \sum_{i=1}^{32} m_i$$

c ranges from 0 to 8,160. Computed by `chain_steps`.

It is split big-endian across the two checksum chains.

$$m_{33} = \lfloor c/256 \rfloor, \quad m_{34} = c \bmod 256$$

Since $8160 = 31 \cdot 256 + 224$, m_{33} never exceeds 31.

4.4 Signing

To sign, release each chain value hashed m_i times. A byte of 0 releases the private value itself. A byte of 255 releases the public value, which reveals nothing further.

$$\sigma_i = H^{m_i}(x_i), \quad \sigma = (\sigma_1, \dots, \sigma_{34})$$

The signature is 34 values of 32 bytes, 1,088 bytes. Signing costs $\sum_i m_i$ hashes.

4.5 Verification

The verifier recomputes the chain lengths from the digest, walks each released value the remaining distance to its endpoint, and hashes the endpoints together.

$$e_i = H^{255-m_i}(\sigma_i), \quad i = 1, \dots, 34$$

$$\text{accept} \iff H(\text{metaspace:commitment:v1} \parallel e_1 \parallel \dots \parallel e_{34}) = C$$

Compared against `Vault.commitment` in `finalize_withdrawal`.

Verification costs $\sum_i (255 - m_i)$ hashes. Signer and verifier together always do exactly $34 \cdot 255 = 8670$. For a uniformly random digest the verifier's share is about 4,300. Solana's compute limit is 1.4 million units per transaction, which is why the verifier's walk is split across several transactions. One push of 8 chains is at most $8 \cdot 255 = 2040$ hashes, and the final check is a single hash over 34 endpoints and the tag.

4.6 Why the checksum stops forgery

An attacker holds a valid signature σ on m and wants a signature on some other digest m' . For a single chain, moving forward is free and moving backward is not:

$$m'_i \geq m_i : \quad \sigma'_i = H^{m'_i-m_i}(\sigma_i) \quad (\text{computable})$$

$$m'_i < m_i : \quad \sigma'_i \text{ is a preimage of } \sigma_i \text{ under } H^{m_i-m'_i}$$

So a forgery is only cheap if every chain moves forward or stays. Suppose the message chains all satisfy $m'_i \geq m_i$ and at least one is strictly larger. Then the sum rises, so the checksum falls:

$$\sum_{i=1}^{32} m'_i > \sum_{i=1}^{32} m_i \implies c' < c.$$

Because c is split big-endian, $c' < c$ means either $m'_{33} < m_{33}$, or $m'_{33} = m_{33}$ and $m'_{34} < m_{34}$. Either way a checksum chain must move backward. There is no $m' \neq m$ in which every one of the 34 chains moves forward or stays still. Every forgery is therefore a preimage of SHA-256, which is the 2^{128} search of section 8. This is the entire argument, and it is why the checksum chains are not optional.

4.7 The digest and what it binds

The digest is not chosen freely. It is the hash of the withdrawal request, as written on chain before any chain value is revealed. Everything that could be substituted is in it.

$$d = H(\text{metaspace:withdraw:v1} \parallel \text{vault} \parallel \text{nonce} \parallel \text{amount} \parallel \text{dest} \parallel C_{\text{next}})$$

Computed by `WithdrawalRequest::digest`. Integers are little-endian 64-bit.

Digest preimage, 133 bytes

```
"metaspace:withdraw:v1" // 21 bytes, ASCII
|| vault                // 32 bytes, the vault's address
|| nonce                // 8 bytes, u64 little-endian
|| amount               // 8 bytes, u64 little-endian, lamports
|| destination          // 32 bytes
|| next_commitment      // 32 bytes
digest = SHA-256(preimage)
```

Field	Bytes	What it stops
<code>metaspace:withdraw:v1</code>	21	A signature for one message format being read as another.
<code>vault</code>	32	Replaying a signature against a different vault.
<code>nonce</code>	8	Replaying a signature against a later state of the same vault.
<code>amount</code>	8	Changing how much is paid.
<code>destination</code>	32	Changing who is paid. The finaliser cannot choose this.
<code>next_commitment</code>	32	Substituting the lock the vault rotates to.

What is absent on purpose: the submitter. Binding the finaliser's key would mean only one party could land the withdrawal. That turns a failed transaction into a stuck vault and buys nothing, because the destination is already fixed.

5. Rotation

The key is one-time. That is the price of the construction, and it is the reason the lock rotates in the same instruction that spends it.

5.1 Why a key must never sign twice

Suppose one key signs two digests, m and m' . For every chain the attacker now holds the lower of the two released values:

$$H^{\min(m_i, m'_i)}(x_i), \quad i = 1, \dots, 34.$$

They can sign any digest m'' whose chain lengths all sit at or above that floor:

$$m''_i \geq \min(m_i, m'_i) \quad \text{for every } i = 1, \dots, 34.$$

With one signature the checksum rules this set down to m itself. With two, the floor on the checksum chains is the lower of two checksums, and digests that satisfy the condition exist. The attacker needs no preimage. The program cannot detect that a key was used twice, so rotation is not a feature. It is a requirement.

5.2 What the program enforces

- `finalize_withdrawal` sets `Vault.commitment` to C_{next} and increments `Vault.nonce` and `Vault.rotations` in the same instruction that pays out. The rotation is unconditional. There is no path that pays without rotating, and the spent signature is public from that point on.
- `open_withdrawal` rejects $C_{\text{next}} = C$ with `CommitmentReused`. Rotating to the same commitment would mean signing twice with one key.
- `init_vault` and `open_withdrawal` both reject a zero commitment with `ZeroCommitment`. A zero commitment is a vault that can be funded and never emptied.
- `push_signature` accepts chains in order only, never skipped and never re-sent, enforced on `chains_filled`. Allowing a chain to be replaced after it was accepted would let two different signatures be mixed into one commitment check.

5.3 What the client must do

The client derives chain secrets deterministically from the seed, so the key for state $n + 1$ is computable before the key for state n is revealed. The next commitment is derived from the next secret before the current one is spent, and the ordering is the client's to keep.

$$C^{(n)} = H\left(\text{metaspace:commitment:v1} \parallel H^{255}(x_1^{(n)}) \parallel \dots \parallel H^{255}(x_{34}^{(n)})\right)$$

One signature per vault state, ever. The client must never sign two different digests against the same commitment, including across retries. A failed push is resent with the same signature over the same digest. If a request is abandoned, the next request is signed only after the vault's state has moved or the request has been cancelled. The program cannot check this, which is why it is stated here as a requirement on the client rather than left to the implementation.

6. The withdrawal protocol

A withdrawal cannot be one transaction. The signature is 1,088 bytes and verifying it is a few thousand hashes. So it is three instructions across at least seven transactions, and the order is the security property: the intent is written down before any signature material is revealed.

One withdrawal, seven transactions

1	open_withdrawal	amount, destination, next_commitment
2	push_signature	chains 1 .. 8
3	push_signature	chains 9 .. 16
4	push_signature	chains 17 .. 24
5	push_signature	chains 25 .. 32
6	push_signature	chains 33 .. 34
7	finalize_withdrawal	pays destination, rotates the lock, closes the request

Quantity	Value
Transactions, minimum	7
Signature bytes delivered	1,088
Instruction data per full push	256 bytes
Hashes per push, worst case	2,040
Hashes to verify, average	about 4,300
Hashes in finalise	1, over 34 endpoints and the tag

6.1 open_withdrawal

Creates a `WithdrawalRequest` at the address derived from `["request", vault, nonce]`, so exactly one request can exist per vault state. It records the amount, the destination, the next commitment, who paid the rent, and the time. Nothing about the signature is revealed yet. It is rejected with `ZeroAmount` if the amount is 0, `ZeroCommitment` if the next commitment is zero, and `CommitmentReused` if it equals the current one. The signer on this instruction pays rent and proves nothing about the vault.

6.2 push_signature

Each push carries between 1 and 8 chain values, in order. The program recomputes the digest from the request, computes the chain lengths, and walks each value to its endpoint:

$$e_j = H^{255-m_j}(\sigma_j), \quad j = \text{chains_filled} + 1, \dots$$

The endpoints are stored on the request and `chains_filled` advances. Chains cannot arrive out of order, cannot be skipped and cannot be re-sent. With 8 per push, 34 chains take pushes of 8, 8, 8, 8 and 2. The digest is read from the request, not from the instruction, so there is nothing for the submitter to vary. A push against a request whose nonce no longer matches the vault fails with `StaleNonce`.

6.3 finalize_withdrawal

Requires all 34 endpoints and a nonce that still matches the vault. Then the check the program exists for:

$$H(\text{metaspace:commitment:v1} \parallel e_1 \parallel \dots \parallel e_{34}) \stackrel{?}{=} \text{Vault.commitment}$$

If it holds, in this order:

- The vault pays the destination. It keeps at least its rent-exempt minimum, computed from `Rent` and the account's own length, or the instruction fails with `InsufficientFunds`. Letting the runtime reclaim a vault account would destroy a live commitment.
- `Vault.commitment` becomes C_{next} . `Vault.nonce` and `Vault.rotations` each increase by one.
- The request is closed and its rent returned to whoever paid it.

No signer is required. The destination account is constrained to `request.destination`, and the rent recipient to `request.payer`. The destination is checked rather than typed because a withdrawal may legitimately target a wallet, a PDA or a program-owned account.

6.4 Who may finalise

Anyone. Someone watching the mempool can take the revealed chains and submit `finalize_withdrawal` themselves. Every field they could want to change is inside the digest the chains sign. The destination account is further constrained to the one in the request, and the rent goes to the request's payer, not to the finaliser. So all a stranger can do is pay the owner's chosen destination, slightly sooner, at their own expense. There is nothing to protect, which is why no signer is asked for. Permissionlessness here is not generosity; it is the absence of anything to protect.

6.5 cancel_withdrawal and the 24-hour expiry

Reclaims a request that will never finalise. It is permissionless, and it has to be: the party best placed to abandon a request is the only party who could have authorised cancelling it. There are two ways in. A request whose nonce no longer matches the vault is already dead, because another withdrawal landed, and closes at once. A current request must have expired:

$$t_{\text{now}} - t_{\text{opened}} \geq 86,400 \text{ seconds}$$

`REQUEST_EXPIRY_SECONDS` is $60 \cdot 60 \cdot 24$. Rent returns to the original payer. Otherwise `NotExpired`.

The expiry gate exists so that a live request cannot be cancelled by a stranger. Cancelling a request mid-push would be a denial of service against the owner.

6.6 Liveness

One vault, one pending request. A request that is opened and abandoned holds the vault's nonce until it expires, 24 hours later. That is a deliberate trade. The failure mode it leaves is liveness, not theft.

It follows that anyone can open a request. `open_withdrawal` only requires a fee payer, so a stranger can open a request against any vault with their own destination. They cannot finalise it; that needs the owner's signature over that digest. It does occupy the vault's nonce until it expires, 24 hours

after it was opened, after which anyone may cancel it and the owner proceeds.

7. Recovery

7.1 The seed

The twelve words are the seed. They are generated in the browser when the vault is created, shown once, and never stored or sent anywhere. At withdrawal they are typed in, used to derive the current one-time key and the next commitment, and discarded. The wallet that created the vault pays fees and rent and is never an authority over it.

7.2 Key derivation

Every one-time key the vault will ever use comes from the seed: one set of 34 chain secrets per rotation. Writing n for the vault's rotation count and KDF for the client's derivation function,

$$x_i^{(n)} = \text{KDF}(\text{seed}, n, i), \quad i = 1, \dots, 34.$$

The derivation is deterministic, so the key for the next state is always computable before the key for the current state is revealed.

The commitment for state n is the hash of those secrets' chain endpoints, as in section 4.2. A client that holds the seed can therefore recompute any past or future commitment, which is what makes the seed sufficient for recovery. A client that generated random keys instead, and lost the next one before finalising, would leave the vault locked behind a commitment nobody can open; deriving from a seed is what rules that out.

7.3 There is no other path

A seed is the only recovery path. There is no admin, no social recovery, no timelock escape and no `close_vault`. A lost seed is lost funds, permanently. The product says this before the first deposit, in those words, and asks for the words to be written down and kept off any device that is online.

8. Security analysis

8.1 Assumptions

A vault authorises spending by checking that revealed values hash to a stored commitment. No signature over any keypair is checked anywhere on the path that moves value. Everything else, including the wallet that pays fees and the `owner_hint` a vault records, is outside the trust

boundary. The construction rests on two properties of SHA-256.

- **Preimage resistance.** Section 4.6 shows that every forgery against a revealed signature requires walking at least one chain backward, which is finding a preimage of SHA-256. This is the assumption the whole design rests on.
- **Second-preimage resistance.** The 34 endpoints are bound to the vault by a single hash. An attacker who could find a different set of endpoints hashing to the same commitment could pass the final check without the signature, so the commitment hash is relied on to admit no second preimage.

Grover's algorithm is the quantum attack on a hash. For an n -bit hash it finds a preimage in about $2^{n/2}$ evaluations instead of 2^n . For SHA-256:

$$2^{256} \longrightarrow 2^{128}.$$

A 2^{128} search is out of reach of anything physical, and there is no known attack that does better. That asymmetry is the whole design. metaspac moves the one assumption from the first row of this table to the second.

Primitive	Problem	Classical	Quantum
Ed25519	Discrete logarithm on Curve25519	$\approx 2^{126}$	Polynomial time (Shor)
SHA-256	Preimage	2^{256}	$\approx 2^{128}$ (Grover)

The lock is plain Winternitz: 34 unkeyed SHA-256 chains of length 255, two of them carrying a checksum, with the endpoints hashed under a domain tag into one 32-byte commitment. It is the same family as the hash-based schemes NIST has standardised, which are built from Winternitz-style one-time signatures (LM-OTS, WOTS+). It is not WOTS+ itself: WOTS+ adds per-step masks so that its proof rests on second-preimage resistance rather than collision resistance. metaspac keeps the chains plain and the checksum a subtraction so that the construction can be checked by reading it. The paper claims lineage, not certification.

8.2 Properties the program enforces

Property	Where it is enforced
A signature authorises exactly one withdrawal.	The digest covers the vault, nonce, amount, destination and next commitment. <code>WithdrawalRequest::digest</code> .
The checksum chains constrain the walk direction.	<code>chain_steps</code> : raising a message byte lowers the checksum.
A commitment is never spent twice.	<code>finalize_withdrawal</code> rotates and increments the nonce in the instruction that pays.
A vault cannot rotate to a commitment it already has.	<code>open_withdrawal</code> , <code>CommitmentReused</code> .
A vault cannot rotate to zero.	<code>init_vault</code> and <code>open_withdrawal</code> , <code>ZeroCommitment</code> .
No instruction that moves value checks a signer.	<code>finalize_withdrawal</code> has no signer at all.
Finalisation is permissionless and safe.	Destination constrained to <code>request.destination</code> ; rent to <code>request.payer</code> .
The vault stays rent exempt.	<code>finalize_withdrawal</code> computes the minimum from <code>Rent</code> and refuses to go below it.
One pending withdrawal per vault.	The request PDA is seeded on the vault's current nonce.
A live request cannot be cancelled by a stranger.	<code>cancel_withdrawal</code> requires expiry while the nonce still matches.
Chains arrive in order and are never replaced.	<code>push_signature</code> enforces the cursor on <code>chains_filled</code> .

8.3 What depends on the client

Two properties depend on the client rather than the program, and the program cannot check them.

- The next secret is derived before the current one is revealed. The client derives chain secrets deterministically from the seed (section 7.2).
- One signature per vault state, ever. A client must never sign two different digests against the same commitment, including across retries (section 5.3).

8.4 What is protected, and what is not

The vault protects a balance against the derivation of a key from a published address, which is the attack Shor's algorithm makes possible and which no Ed25519 wallet, multisig, hardware wallet or timelock protects against. It protects against a watcher in the mempool, because the revealed chains

authorise exactly one payment to exactly one destination. It protects against replay, because the digest binds the vault and its nonce. It protects against an operator, because there is none.

It does not protect against the following.

- Theft of the seed. Whoever holds the twelve words holds the vault.
- Loss of the seed. There is no recovery path that bypasses the commitment.
- A compromised client that signs a digest the owner did not intend.
- A client that reuses a one-time key. Section 5.1 shows what two signatures leak.
- A bug in the verifier. Every property in 8.2 is conditional on `hashlock.rs` implementing the scheme as described, which is why it is written to be read rather than to be clever, and why `INVARIANTS.md` states each property so that a reviewer has something to disagree with.
- Denial of service against liveness. A stranger can open a request against any vault and hold its nonce for 24 hours. They cannot take anything, and the request expires.

8.5 Errors

Eleven errors, in the order the program declares them. Anchor numbers custom errors from 6000.

Code	Error	Raised by
6000	<code>CommitmentMismatch</code>	<code>finalize_withdrawal</code>
6001	<code>ZeroCommitment</code>	<code>init_vault</code> , <code>open_withdrawal</code>
6002	<code>CommitmentReused</code>	<code>open_withdrawal</code>
6003	<code>ChainOutOfOrder</code>	<code>push_signature</code>
6004	<code>SignatureComplete</code>	<code>push_signature</code>
6005	<code>SignatureIncomplete</code>	<code>finalize_withdrawal</code>
6006	<code>StaleNonce</code>	<code>push_signature</code> , <code>finalize_withdrawal</code>
6007	<code>InsufficientFunds</code>	<code>finalize_withdrawal</code>
6008	<code>ZeroAmount</code>	<code>deposit</code> , <code>open_withdrawal</code>
6009	<code>NotExpired</code>	<code>cancel_withdrawal</code>
6010	<code>Overflow</code>	<code>finalize_withdrawal</code>

9. The scanner

The scanner makes the problem concrete. Paste a wallet address or a program id and see what one Ed25519 key is holding. It is free, read-only and needs no wallet connection and no signature; it is pure read-only RPC, and it works with no database, no program and no wallet connected.

9.1 What it measures

Six independent checks, run in parallel and reported as each one settles.

Check	What it reports	How it is obtained
Signing history	Whether the address has ever sent a transaction.	<code>getSignaturesForAddress</code> , limit 1.
SOL balance	SOL sitting directly under the key.	<code>getBalance</code>
Token accounts	Token positions owned by the key, Token and Token-2022.	<code>getTokenAccountsByOwner</code>
Mint and freeze authority	Mints the key can inflate, and mints it can freeze.	<code>getProgramAccounts</code> on the Token programs, filtered on the authority offsets.
Program upgrade authority	Programs the key can replace.	<code>getProgramAccounts</code> on the upgradeable loader.
Stake authority	Stake accounts the key can withdraw from.	<code>getProgramAccounts</code> on the Stake program, staker and withdrawer offsets.

Findings are ordered by what an attacker gets, not by what is easiest to explain. An upgrade authority outranks any balance: taking a treasury costs the treasury, while replacing a program costs every user who has ever approved it. A mint authority outranks a balance for the same reason, because it is unbounded. Freeze authority and withdrawable stake follow, then the balance and the token positions.

9.2 Why it is built the way it is

The last finding is always the same, and it is deliberately last: the public key is the address itself. A Solana address is its Ed25519 public key, base58 encoded, visible since the address existed, whether or not it has ever signed. The signing-history check is activity, not exposure, and the scanner never reads it as the key being private. It reads better after the user has seen what the funds are, because it is the finding that cannot be fixed by moving them.

A section of a scan is never a bare value. Each is either an answer or a statement that no answer could be had and why. The expensive questions, which mints this address can inflate and which programs it can upgrade, are `getProgramAccounts` calls that an endpoint will rate-limit, refuse or time out on. A scanner that renders a rate-limited reply as “no mint authorities found” has told the

user their project is safe when it has not looked, which is worse than no scanner at all. So a check that could not be read is rendered as “could not check” and never as an all-clear, and the type system does not allow the interface to show a section without handling the gap.

10. Roadmap

The work is sequenced by how fast it ships, not by how much it earns. The scanner comes first because it makes the problem concrete in a reader’s own wallet, with their own numbers, and because nothing else is worth building until it has. The vault is the fix.

1. **Exposure scanner.** Section 9. Free, read-only, nothing to deploy.
2. **Personal vault.** SOL behind a hash commitment, withdraw any amount at any time, the lock rotating on every use. Token support follows the SOL path, with the digest binding the mint and the token amount under a `v2` tag.
3. **Authority vault.** The same lock pointed at upgrade, mint and treasury authority, with the vault signing as a PDA. The intent is a public verification page, so that the claim that an authority is held behind a commitment can be checked against the chain by anyone.
4. **Staking vault.** The withdrawer authority of a stake account behind a commitment, so that delegation and yield continue while the principal is protected.

The roadmap intends to fund the work with a small fee on the personal vault, charged on deposit rather than on withdrawal, and a setup plus monthly fee on the authority vault. The program itself carries no fee; a fee belongs at the edge, on deposit, where it cannot sit on the withdrawal path. The roadmap ties a token to fee buybacks from those two vaults and a stake requirement on the authority tier, and to nothing else: it is not a yield source, not a governance surface over vault parameters, and never something a vault’s safety depends on. A security product whose guarantee weakens when its token falls is not a security product.

Two measurements decide the shape of what follows. The compute cost of a push is to be measured rather than estimated, and if it moves, the number of transactions per withdrawal moves with it. And the message-binding property of section 4.7 is the one a reviewer is asked to attack first, because a vault that is front-runnable in the mempool would be worse than the wallet it replaced.

11. References

-
- [1] NIST IR 8547 (initial public draft), Transition to Post-Quantum Cryptography Standards, November 2024. Table 2 lists EdDSA at 128 bits of security as disallowed after 2035.
<https://csrc.nist.gov/pubs/ir/8547/ipd>

- [2] NIST SP 800-208, Recommendation for Stateful Hash-Based Signature Schemes, October 2020. Approves LMS and XMSS, both built on Winternitz one-time signatures. <https://csrc.nist.gov/pubs/sp/800/208/final>
- [3] FIPS 205, Stateless Hash-Based Digital Signature Standard, August 2024. Specifies SLH-DSA, built on WOTS+. <https://csrc.nist.gov/pubs/fips/205/final>
- [4] R. C. Merkle, A certified digital signature, CRYPTO 1989. Introduces the one-time signature improvement attributed to Winternitz and the hash tree that builds many-time schemes from it. https://link.springer.com/chapter/10.1007/0-387-34805-0_21
- [5] A. Hülsing, W-OTS+: shorter signatures for hash-based signature schemes, AFRICACRYPT 2013. The masked variant used by XMSS and SLH-DSA. <https://eprint.iacr.org/2017/965>
- [6] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM Journal on Computing 26(5), 1997. <https://arxiv.org/abs/quant-ph/9508027>
- [7] L. K. Grover, A fast quantum mechanical algorithm for database search, STOC 1996. <https://arxiv.org/abs/quant-ph/9605043>
- [8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, B.-Y. Yang, High-speed high-security signatures, CHES 2011. The Ed25519 signature scheme. <https://ed25519.cr.yp.to/ed25519-20110926.pdf>
- [9] RFC 8032, Edwards-Curve Digital Signature Algorithm (EdDSA), January 2017. <https://www.rfc-editor.org/rfc/rfc8032>
- [10] Solana documentation, Core concepts: Accounts. An account's address is its 32-byte Ed25519 public key. <https://solana.com/docs/core/accounts>
- [11] Solana documentation, Core concepts: Transactions. The 1,232-byte transaction size limit. <https://solana.com/docs/core/transactions>
- [12] Solana documentation, Core concepts: Fees. The compute budget, with a maximum of 1.4 million compute units per transaction. <https://solana.com/docs/core/fees>
- [13] FIPS 180-4, Secure Hash Standard (SHS), August 2015. SHA-256. <https://csrc.nist.gov/pubs/fips/180-4/upd1/final>