TECHNICAL PAPER · VERSION 1.1

Orien: a quantum-proof cryptocurrency without blocks

Orien is a cryptocurrency whose coin is ORN. Each account orders its own payments, and a payment is final as soon as validators holding more than two thirds of the stake have vouched for it, with no blocks and no global queue. Validators agree on anything only once an hour, at a checkpoint. Every signature is a hash-based one-time signature built on BLAKE3 alone, so the system rests on a single assumption that quantum computers are not known to break. New coins, 100 million in all, are released on a fixed curve and earned by proof-of-work.

This paper states the rules exactly as version 1.1 of the **orien** program implements them. Where the two could ever differ, the program's source code is the reference.

CONTENTS

1. Introduction
2. Building blocks
3. Accounts and orders
4. How a payment becomes final
5. Validators and stake
6. Epochs and checkpoints
7. Fees
8. New coins
9. Genesis
10. The network
11. Wallets
12. Software releases
13. Security
14. Measured performance
15. Known limitations
16. Parameters
17. References

1 Introduction

Most cryptocurrencies settle payments by agreeing on a single, ever-growing sequence of blocks. Every payment in the world waits in the same queue, and finality is a matter of waiting for enough blocks on top of it. Orien starts from a different observation: a payment only needs to be ordered relative to other payments from the same account. Two payments from different people never conflict, so they never need to wait for each other.

Each Orien account therefore numbers its own orders 0, 1, 2 and so on. Validators check each order on arrival and sign it if it is the account's next number and the account can afford it. Once validators

holding more than two thirds of the stake have signed, and then confirmed that they hold that certificate, the payment is final. Validators never sign two different orders under the same number of the same account, so no two conflicting payments can both gather a quorum.

Some decisions do need agreement among all validators: which payments an epoch contained, who validates next, who cheated, how fees and new coins are shared. Orien makes them once an hour, at a checkpoint, with a classical agreement protocol. Between checkpoints, payments flow without any agreement round at all.

The design follows four goals:

- **Quantum-proof by construction.** Every signature is a hash-based one-time signature. The whole system assumes only that BLAKE3 behaves as a secure hash function.
- **Fast, independent finality.** A payment is final after two short round trips to the validators, and is never delayed by anyone else's payments.
- **Checkable by anyone.** A payment's receipt proves it final to anyone holding the network's genesis file and its checkpoints, offline. Balances are proven by checkpoints that wallets check themselves.
- **A fair start.** The network starts with no coins. Every coin is released on a fixed schedule to whoever does proof-of-work, beginning at a Bitcoin block named in advance.

The building blocks that make this work are known: per-account ordering with quorum certificates follows the line of work on consensusless payment systems such as FastPay [1], the hourly agreement is Tendermint [2], and the signatures are the Winternitz one-time signatures and Merkle key trees that underlie the SLH-DSA standard [3, 4]. Orien combines them into one small, complete system.

Throughout, amounts are counted in **units**: one ORN is 10^9 units, so balances have nine decimal places. Structures are encoded with Borsh [5], and integers are little-endian.

---

2 Building blocks

2.1 One hash function

Orien uses a single cryptographic primitive, the BLAKE3 hash function [6], with 32-byte outputs. Every use is separated by a one-byte domain tag, so that a value hashed for one purpose can never be mistaken for a value hashed for another:

$$\mathrm{hash}(\mathrm{tag}, \mathrm{part}_1, \ldots, \mathrm{part}_n) = \mathrm{BLAKE3}(\mathrm{tag} \parallel \mathrm{part}_1 \parallel \ldots \parallel \mathrm{part}_n)$$

Secret values are derived from seeds with BLAKE3 in keyed mode, as a pseudorandom function over a one-byte purpose and a 64-bit index:

$$\mathrm{prf}(\mathrm{key}, \mathrm{purpose}, i) = \mathrm{BLAKE3_keyed}(\mathrm{key}, \mathrm{purpose} \parallel \mathrm{u64_le}(i))$$

The tags are listed in the appendix.

2.2 One-time signatures

A one-time key signs exactly one message. Orien uses the Winternitz one-time signature (WOTS) with base $w = 16$ and 24-byte chain values, the same shape as the WOTS layer of SLH-DSA-192 [3]. A key is a 32-byte seed. From it the PRF derives a 24-byte public seed and 51 chain starts. Each chain is walked 15 steps:

```
step(pub_seed, chain, position, v) = BLAKE3(0x01 ‖ pub_seed ‖ chain ‖ position ‖ v)
[0..24]
endᵢ = 15 steps from startᵢ
commitment = hash(0x02, pub_seed ‖ end₀ ‖ … ‖ end₅₀)          (32 bytes)
```

The commitment is the public key. To sign a message, its digest D = hash(0x03, pub_seed ‖ message) is computed, and the first 24 bytes of D are cut into 48 base-16 digits $d_0 \ldots d_{47}$ (high nibble first), followed by three checksum digits of $c = \Sigma(15 - d_i)$. The signature reveals the public seed and, for each of the 51 digits, the chain value d_i steps from its start. A verifier walks each revealed value the rest of the way to step 15 and recomputes the commitment.

A signature is 24 × 52 = **1,248 bytes**. The checksum means that forging a signature on a second message would require walking some chain backwards, which is inverting BLAKE3. Signing two different messages with one key reveals enough to forge; the rules in sections 3 and 6 make sure no honest key ever does, and turn any key that does into evidence.

2.3 Key trees

Validators sign many messages, so they use a key tree: a Merkle tree [7] whose 2^h leaves are the commitments of one-time keys, with leaf *j* derived as prf(tree_seed, 0x04, j). Inner nodes are hash(0x10, left ‖ right), and the root is the public key. A tree signature names its leaf, carries that leaf's one-time signature, and the h sibling hashes from leaf to root:

```
TreeSignature { leaf: u32, ots: Signature, path: [Hash32; h] }     1,256 + 32·h bytes
```

Heights are at most 24. A validator key has height 16 by default (65,536 one-time keys), and signs whole batches at a time (section 4), so one leaf covers many payments.

2.4 Merkle trees of items

Lists that are signed or proven together (a batch of votes, the set of all accounts) form complete binary Merkle trees, padded to a power of two. Two kinds keep them apart: batches use tags 0x20, 0x21 and 0x22 for leaves, nodes and padding, and the state uses 0x40, 0x41 and 0x42. A path proves one item's place in a tree of depth at most 32.

---

3 Accounts and orders

3.1 Addresses

An account is controlled by a chain of one-time keys, one per order, plus 16 spare keys held in a key tree of height 4. Its address commits to the first key and to the spare tree:

```
address = hash(0x35, first_commitment ‖ spare_root)
```

An address is 32 bytes, written as 64 hexadecimal characters. No public key is revealed until the account's first order, and each order names the key that will sign the next one, so a key is exposed only for the short time between signing and finality.

3.2 Orders

Every change to an account is an order, signed by the account:

```
Order {
  kind:     Payment | Stake { validator, proof } | Unstake
  from:     AccountId          the paying account
  seq:      u64                its order number: 0, 1, 2, …
  to:       AccountId          recipient (the account itself for stake and unstake)
  amount:   u64                in units
  max_fee:  u64                the largest fee the owner accepts
  next_key: Hash32             commitment of the key that will sign order seq + 1
  spare:    Hash32             root of the account's spare keys
}
digest = hash(0x30, network_id ‖ borsh(order))
```

The digest covers the network's identity (section 9), so an order is valid on one network only. A payment order is 153 bytes and its signature 1,249 bytes.

An account's state records its balance, its next order number, and which key must sign next: its first key (until it has signed), the key named by its last order's `next_key`, or a numbered spare key after a cancellation (section 4.5). An order is accepted only under the account's current number and only with exactly the key the account expects.

3.3 What a validator checks

Before voting for an order, a validator first checks that its signature is well formed and that a stake order carries a valid proof from its validator (section 5.1). Then it checks, in this order, and refuses on the first failure:

1. the amount is not zero, and the paying account exists;
2. the order number is the account's current one, not an earlier or later one;
3. it is signed with the key the account expects;
4. this validator has not already voted for a different order under that number;
5. the recipient is not the mint or the rewards account (section 3.4);
6. a stake or unstake names the account itself, and a stake names a usable validator;
7. `max_fee` covers the fee the network charges (section 7);
8. the balance covers the amount plus the fee (for an unstake: the fee, and enough coins staked to release).

A validator records which order it voted for under each (account, number) *before* its vote leaves the machine, writing it durably to disk, so that not even a crash can make it vote twice.

3.4 Reserved accounts

Four addresses belong to no key and are used by the system itself:

Account	Address	Holds
Forfeited	00…00	stake taken from validators caught cheating; never spendable
Fees	01…01	fees collected during the current epoch, paid out at the checkpoint
Mint	02…02	minus the number of coins released so far, so the total of all accounts stays zero
Rewards	03…03	released coins waiting to be paid to miners

Payments to the mint and rewards accounts are refused. Coins paid to the fees account are shared among validators at the next checkpoint, and coins paid to the forfeited account can never be spent again.

---

4 How a payment becomes final

4.1 Votes, signed in batches

A validator signs its votes in batches. It gathers the items to sign, builds a Merkle tree over them, and signs the root once with one leaf of its key tree:

```
batch_message = hash(0x31, network_id ‖ u64_le(epoch) ‖ batch_root)
Vote { validator: u16, batch_path: Path, signature: TreeSignature }
```

A vote proves its item through the batch path, so one tree signature serves every payment in the batch. The busier the network, the more payments each one-time key covers.

4.2 Certificates

The paying wallet sends its signed order to every validator and collects their votes. Votes from validators holding more than two thirds of the epoch's total stake form a certificate:

```
Certificate { epoch, order, signature, votes }        valid when 3 · stake(votes) > 2 ·
total_stake
```

Votes are listed by increasing validator number, and each is checked against that validator's key and the certificate's epoch.

4.3 Confirmations and finality

The wallet then hands the certificate to the validators. Each applies the order to its copy of the ledger and signs a confirmation of the item hash(0x34, order_digest), again in batches. A certificate together with confirmations from more than two thirds of the stake is a receipt, and the payment is **final**:

```
Receipt { certificate, confirmations }        valid when both carry more than two
thirds of the stake
```

Anyone with the genesis file can check a receipt without contacting the network: it is checked against the validators of its own epoch, which the genesis file names for the first epoch and the checkpoints name after that. The second step matters at checkpoints: a payment with a receipt is known to more than two thirds of the validators, so it cannot be left out of the epoch's record (section 6).

4.4 Applying a certified order

A validator applies a certificate even if it did not vote for it, as long as the order is the account's next one. It moves the amount, charges the fee into the fees account, records the next key, and increases the account's order number. Balances are signed numbers internally, because certificates for payments into an account can arrive before or after payments out of it.

4.5 Two orders under one number

An honest wallet signs one order per number. If the owner, or a thief with a copy of the wallet, signs two different orders under the same number, honest validators vote for whichever they see first. Neither may then reach a quorum, and the account is stuck at that number. The two signed orders are themselves the evidence: any validator includes them in its statement at the checkpoint, and the checkpoint cancels that number unless one of the two orders already became final. Nothing moves, and from then on the account must sign with its next spare key. After all 16 spare keys are used, the account stays locked.

Two signatures from one one-time key reveal enough of its chains that a third order can sometimes be forged from them, and such an order could gather votes before the checkpoint. Signing twice is therefore never safe, and an honest wallet never does it. Spare keys protect an account against the forgery of a burned key; they do not protect it from a thief who holds a copy of the whole wallet, since that copy holds the spare keys too.

4.6 Why conflicting payments cannot both be final

Two certificates for different orders under the same account and number would each need votes from more than two thirds of the stake. Any two such sets overlap in more than one third of the stake. If validators holding at most one third misbehave, at least one honest validator lies in the overlap, and it never votes twice. So at most one of the two orders can ever be certified, and no coin can be spent twice.

---

5 Validators and stake

5.1 Staking

Anyone holding ORN can lock coins behind a validator with a stake order. The order names the validator's key (its root and height) and the address where it can be reached, and carries a proof that the validator agrees, signed with leaf 0 of the validator's key tree, which is set aside for this purpose:

```
stake_proof = sign(leaf 0, hash(0x45, network_id ‖ account ‖ address))
```

An unstake order releases staked coins. They stay locked, and can still be taken for cheating, for 168 epochs (about a week, at one epoch an hour), then return to the account's spendable balance.

5.2 Choosing the validators

At each checkpoint the next epoch's validators are chosen deterministically from the state. Accounts with at least **10,000 ORN** staked behind a usable validator are sorted by stake, largest first, ties broken by account address, and the first **33** distinct validator keys are taken. Each votes with a weight equal to its stake. Validators proven to have cheated during the epoch are left out of the next epoch's set, and so is any key they used.

5.3 Launch seats

A network that starts with no coins has no stake. So it starts with launch seats: validators named in the genesis file, which together carry a fixed weight (3 million ORN of weight in the launch configuration), shared equally. Seats give way as real stake arrives. With *real* the total stake of chosen stakers and *seats* the seats' combined starting weight, each seat's weight at a checkpoint is:

```
weight(seat) = max( ⌊(seats − real) · start_weight(seat) / seats⌋, 1 ) + backing(seat)
```

Stake placed behind a seat's own key counts as that seat's backing, not as a separate validator. Once real stake reaches the seats' starting weight, the seats leave for good and only stakers remain. Seats never earn fees.

5.4 Penalties

Two kinds of cheating are provable from signatures alone:

- **Two votes**: one validator voting for two different orders under the same account and number.
- **Key reuse**: one leaf of a validator's key tree signing two different messages (other than the stake proof leaf).

Proof travels in validators' statements. At the checkpoint, a convicted staked validator loses everything its account holds in stake (staked, unstaking and waiting to be released), moved to the forfeited account; a convicted seat loses its seat for good. Either way its key is left out of the next epoch's validators.

---

6 Epochs and checkpoints

6.1 Epochs

Time is divided into epochs of one hour. Each epoch has a fixed set of validators, and every vote and certificate names its epoch. A validator starts closing an epoch when its hour is up, after 50,000 payments, or as soon as it sees validators holding more than a third of the stake closing it.

6.2 Statements

To close an epoch, each validator publishes a signed statement of what it saw: a running hash, or pin, of every payment it applied, in the order it applied them; evidence of orders signed twice; proof of cheating validators; the proof-of-work shares it received; and optionally a new key for the next epoch.

```
pin₀ = hash(0x37, u64_le(epoch)),   pinₖ₊₁ = hash(0x37, pinₖ ‖ order_digest)
Statement { epoch, validator, log: Pin, evidence, next_key, cheats, shares }
```

6.3 The checkpoint

A checkpoint merges statements from validators holding more than two thirds of the stake. The epoch's payments must include every payment in the logs those statements pin. Because a final payment was confirmed by more than two thirds of the stake, and any two such sets overlap in an honest validator, every final payment is in some merged log.

```
Checkpoint {
  epoch, previous,          this epoch, and the digest of the previous checkpoint
  time,                     the proposer's clock
  state,                    Merkle root of every account after closing
  payments, merged,         how many payments, and which statements
  validators,               the next epoch's validators and their weights
  released, target          new coins released, and the next proof-of-work target
}
```

Closing applies, in this order:

1. every payment, sorted by account and order number, each of which must be certified;
2. a check that no account other than the mint is overdrawn;
3. cancellations for orders signed twice (section 4.5);
4. forfeits for convicted validators (section 5.4);
5. payout of the epoch's fees (section 7);
6. release of new coins and their payout to miners (section 8);
7. new keys announced by staked validators;
8. unstaked coins that have waited long enough, returned to their owners.

6.4 Agreement

Validators agree on the checkpoint with Tendermint [2], rule for rule. In each round a proposer, drawn by weight from a seed taken from the previous checkpoint, proposes a checkpoint together with the statements it merges. Validators check it by recomputing it from the statements, prevote, then precommit; a checkpoint precommitted by more than two thirds of the weight is decided, and the next epoch begins. The rule of locking on a precommitted value keeps two different checkpoints from ever being decided for one epoch, as long as no more than a third of the weight is faulty.

---

7 Fees

Every payment pays a fee of 0.1% of its amount, between a floor and a ceiling:

```
fee(payment) = clamp( ⌊amount · rate / 1,000,000⌋, min, max )
rate = 1,000 (0.1%)    min = 1,000 units (0.000001 ORN)    max = 10⁹ units (1 ORN)
```

```
fee(stake) = fee(unstake) = min
```

A share of the amount keeps the fee about the same in real terms whatever a coin is worth. The floor stops floods of tiny payments, and the ceiling keeps large payments from paying far more than they cost the network. The fee is paid on top of the amount, and an order states the largest fee its owner accepts.

Fees collect in the fees account during the epoch. At the checkpoint they are shared among that epoch's staked validators that were not caught cheating, in proportion to stake: each receives ⌊pot · stake / total⌋, and the remainder waits for the next checkpoint.

---

8 New coins

8.1 The release curve

ORN are created only by release, on a fixed curve that approaches 100 million coins and halves what remains every four years. With t the seconds since the network's start time:

```
released(t) = CAP · (1 − 2^(−t / HALF_LIFE))
CAP = 100,000,000 ORN = 10¹⁷ units       HALF_LIFE = 126,230,400 s  (4 × 365.25 days)
```

Exact arithmetic matters, since every validator must compute the same number. The curve is evaluated in 64-bit fixed point: PER_SECOND = 18,446,743,972,416,134,102, which is $2^{-1/\text{HALF_LIFE}} \times 2^{64}$ rounded to nearest, is raised to the power (t mod HALF_LIFE) by square-and-multiply with every product rounded to nearest, then halved once per whole half-life:

```
h = ⌊t / HALF_LIFE⌋,   u = PER_SECOND^(t mod HALF_LIFE) >> h        (u = 0 when h ≥ 64)
released(t) = CAP − ⌊CAP · u / 2⁶⁴⌋
```

This releases about 1,977 ORN in the first hour, 47,432 in the first day, 50 million in the first four years and 75 million in the first eight. At each checkpoint the coins released since the previous one move from the mint to the rewards account. The amount depends on elapsed time, not on the number of epochs.

8.2 Proof-of-work shares

Miners earn released coins with shares: numbers that, hashed with the previous checkpoint and the miner's account, fall below the epoch's target:

```
work = u64_le( hash(0x46, seed ‖ account ‖ u64_le(nonce))[0..8] )
a share counts when work < target          seed = digest of the previous checkpoint
```

Because the seed is the previous checkpoint, no share can be computed before that checkpoint exists. Miners send shares to validators, which list them in their statements. At the checkpoint, the shares from the merged statements are deduplicated and counted per account, and the rewards account is shared out in proportion: each account receives ⌊pot · its_shares / all_shares⌋, and the remainder waits.

Work only creates coins. It never decides which payments happen or in what order, so a miner with a great deal of computing power can earn more coins but cannot rewrite anyone's payments.

8.3 The target

The target aims for 1,000 shares per epoch, so that pay is spread finely however much work is done. After each epoch with *n* shares under target *T*:

```
T′ = 4 · T                                  if n = 0
T′ = clamp( T · 1,000 / n,  ⌊T / 4⌋,  4 · T )  otherwise
```

The first target is $2^{40} - 1$: about one share for every 16.8 million hashes. The target moves by at most a factor of four per epoch, which limits how far a sudden burst of work can push it.

---

9 Genesis

A network is defined by its genesis file: its name, its launch validators (each with a key root, height, weight and address), any starting balances, and every rule this paper describes as a number: epoch length, payments per epoch, start time, minimum stake, number of validators, unstaking time, the fee rule, the cap, the shares wanted per epoch and the first target. The network's identity is a hash of all of them:

```
network_id = hash(0x32, name, validators (root, height, weight), allocations, every
rule, bitcoin block)
```

Every order, vote, statement and stake proof is bound to this identity, and checkpoints are chained from it, so two networks with different rules can never accept each other's signatures. Validator names and addresses are not part of it, so a validator can move without changing the network.

The launch configuration has no starting balances at all: every coin is earned. It names a Bitcoin block, by height and hash, that will be mined after the genesis file is published. The block's hash is part of the

network's identity, and the identity is the seed of the first epoch's mining, so nobody can mine before that block exists.

---

10 The network

10.1 Messages

Wallets, miners and validators talk over TCP. Each message is a 4-byte length followed by a Borsh-encoded request or response, at most 16 MiB. A connection carries many request and response pairs in turn. Every request can safely be sent twice, so a client that loses a connection simply asks again.

10.2 Between validators

Validators find each other through the addresses in the genesis file and in each checkpoint's list of validators. Each validator keeps two links to every other one: it pushes its own closing messages and the proof-of-work shares it receives, and once a second (every 40 ms while an epoch is closing) it pulls the certificates and closing messages it has not yet seen. A validator that missed payments, or a whole epoch, catches up from its peers this way.

Connections are not encrypted. They do not need to be for safety: every vote, certificate, statement and checkpoint is signed and checked on arrival, so a message cannot be forged or altered in transit, only read or dropped. To stay open to the internet, a validator accepts at most 1,024 connections in all and 64 from any one address, closes a connection that stays silent for 30 seconds, and closes one that sends anything it cannot decode. Other validators of the current epoch, when listed by IP address, are exempt from these limits.

10.3 Durability

A validator writes each promise to disk before it is kept. The signed order behind every vote is written and synced before the vote leaves; a certificate is synced before its confirmation leaves; each one-time key number is reserved on disk before it signs; and every closing message is journaled before it is sent. After a crash, a validator replays its records and cannot contradict anything it said before.

10.4 Storage that stays small

A validator keeps the ledger state of the last three epochs and the certificates of the current one. Older certificates, closing journals and states are deleted at each checkpoint, so the storage a validator needs depends on the number of accounts, not on the network's age. Checkpoints themselves, about 40 KB an hour with 33 validators, are kept. A validator that falls more than an epoch behind catches up in one jump: it fetches the checkpoints it missed, checks each one against the validators of its epoch, then fetches the latest state and checks it against the last checkpoint's state root.

10.5 Validators that join

A validator outside the genesis file runs in watching mode until a checkpoint chooses it. It follows decided checkpoints and state like any other node, but neither votes nor closes epochs. When a checkpoint lists its key, it takes its place at the start of the next epoch.

---

11 Wallets

11.1 Trusting only what can be checked

A wallet trusts no single validator. It starts from the genesis file and follows checkpoints, accepting each only if it continues the previous one and carries precommits from more than two thirds of that epoch's weight. It remembers how far it got, so it does not start from genesis again. A balance is proven by a Merkle path from the account to the last checkpoint's state root. For what happened since, a wallet counts only what validators holding more than two thirds of the stake all report, and a validator claiming the account has made more payments than the checkpoint shows must produce the certificate and confirmations for them. Miners take the seed and target for their work from checked checkpoints in the same way.

11.2 Keys

A wallet holds one 32-byte account seed. One-time key *n*, which signs the account's order *n* and nothing else, is derived as prf(seed, 0x03, n), and the 16 spare keys come from prf(seed, 0x05, 0). Before a signed order leaves the wallet, the wallet writes it to its own file and syncs it. From then on it will only ever send that same order under that number. It also records what each of its keys has signed, and keeps that record even after letting go of an order a checkpoint cancelled, so no key of the wallet ever signs two different orders, whatever it is asked and even across crashes. If another copy of the same wallet signed an order, any copy can find it at the validators and finish it.

11.3 Recovery phrase

A wallet is made from a 256-bit master secret, written down as 24 words from the standard list of 2,048 English words that most cryptocurrency wallets use. The 24 words carry 264 bits: the secret, and an 8-bit check taken from hash(0x48, master) rather than from SHA-256, so that an Orien phrase is almost always refused by wallets for other cryptocurrencies, and theirs by Orien. A word can be typed by its first four letters.

One phrase makes many accounts, one per computer: account *n* has the seed hash(0x49, master ‖ u32_le(n)). Each computer should use its own account number: two computers using the same one could sign two different orders under one number.

11.4 Passwords

A wallet file can be sealed with a password. The password is stretched with Argon2id [8] (64 MiB of memory, 3 passes), and the secret is encrypted with XChaCha20-Poly1305, with the account's address as associated data so that a sealed secret cannot be moved into another wallet file. The address stays readable, so a balance can be checked without the password.

11.5 Unlocking for a while

Its owner can unlock a sealed wallet for a set time and up to a set amount, fees included, so that programs they trust, such as an AI assistant, can send without ever learning the password. A small helper process holds the opened wallet in memory only and listens on the computer's loopback address. It answers only requests that carry a random 32-byte token, kept in a file only the owner can read. It signs through the same wallet rules, refuses anything beyond the remaining amount, and ends when the time is up, when its owner locks it, or when the computer restarts.

---

12 Software releases

The **orien** program is published for macOS and Linux on x86-64 and ARM, and for Windows on x86-64. Each release lists the SHA-256 checksum of every build in a file that is signed with the Orien release key: a key tree of height 10, quantum-proof like every other key in the system, whose root is built into the program. The signature covers hash(0x47, checksums).

`orien update` installs a newer version only after checking that signature against the built-in root and the download against its signed checksum. If the website that serves releases were ever taken over, it could not make `orien update` install anything the release key did not sign.

---

13 Security

13.1 Assumptions

- **BLAKE3 is a secure hash function**: preimages and second preimages cannot be found faster than by generic search, and keyed BLAKE3 is a pseudorandom function. The protocol assumes nothing else. There is no elliptic curve, no factoring and no lattice anywhere in the protocol.
- **Safety** holds as long as validators holding at most one third of an epoch's weight misbehave in any way.
- **Progress** needs validators holding more than two thirds of the weight to be honest and reachable. Payments need nothing more. Checkpoints, like any agreement protocol, also need the network to deliver messages within some bound eventually (partial synchrony [2]).

13.2 What is guaranteed

- **No double spending.** Two certificates for conflicting orders would need two quorums, whose overlap of more than a third of the stake would have to vote twice (section 4.6).
- **Nothing final is lost.** A final payment was confirmed by more than two thirds of the stake, so at least one honest validator in any checkpoint's merged statements has it in its pinned log.
- **One checkpoint per epoch.** Tendermint's locking rule keeps two different checkpoints from being decided for the same epoch.
- **Cheating is punished.** Voting twice and reusing a one-time key leave signed evidence that anyone can check, and a convicted validator loses its stake and its place in the next epoch.

13.3 Quantum computers

Shor's algorithm breaks the elliptic-curve signatures most cryptocurrencies use, but not hash functions. Against a hash function, a quantum computer can at best search faster with Grover's algorithm, which halves the effective number of bits. Orien's one-time signatures use the parameters of the one-time layer of SLH-DSA-192, NIST's security category 3, with BLAKE3 in place of SHA-2 or SHAKE: 192-bit chain values, leaving about 96 bits against a quantum search. One-time keys add a second layer: an account's next key is revealed only when it signs, and is used for exactly one order, so there is never a long-lived public key for an attacker to work on.

13.4 Attacks considered

- **A stolen copy of a wallet** can spend. The owner's best defence is a password on the file and the recovery phrase kept on paper. If thief and owner both sign under one number, the account stalls until the next checkpoint cancels both. Since the thief's copy holds the spare keys too, the owner should then move what is left to a new wallet.
- **Flooding** is limited by the minimum fee on every payment, by per-address connection limits and timeouts, and by a cap of four times the wanted shares per epoch on each validator.
- **Mining before the start** is impossible: the first seed depends on a Bitcoin block that does not exist when the genesis file is published.
- **A burst of mining power** can earn more coins, but cannot touch payments, and moves the target by at most a factor of four per epoch.

---

14 Measured performance

These figures come from validators that sync every vote to disk before answering, with four validators in total.

Measure	Result	Conditions
Time to finality	162 ms median (138 to 240 ms)	40 payments; four validators on three machines, one of them 57 ms away

Measure	Result	Conditions
Payments per second	950 to 1,500	four validators on one laptop, 100 to 250 clients at once
Payments covered by one validator signature	10 to 28	under load, on one laptop
A payment that runs into a checkpoint	about 1.1 s	on the three machines
One-time signature	1,248 bytes	
Receipt	12,086 bytes	3 of 4 votes and 3 of 4 confirmations
Checking one signature	about 25 µs (40,000 a second)	one processor core

About 115 ms of the 162 ms is the network's own travel time: a payment needs two round trips to the validators, and the rest is signing, checking and syncing to disk.

---

15 Known limitations

- **Nothing is private.** Balances, payments and addresses are public, as in most cryptocurrencies, and connections are not encrypted.
- **Old validators.** Like every system secured by stake, a wallet starting from genesis must trust that no group holding more than two thirds of some past epoch's weight later signed a different history. Wallets that keep following the network are not affected.
- **Restarts during an epoch.** Proof-of-work shares and proofs of cheating are held in memory until the checkpoint, so a validator that restarts mid-epoch forgets the ones it alone received.
- **Spare keys run out.** An account can be cancelled 16 times. After that it stays locked.
- **Large signatures.** Hash-based signatures are about 20 times larger than elliptic-curve ones. Batching keeps validators' share of this small, but each payment carries its own 1,249-byte signature.

---

16 Parameters

Parameter	Value
Unit	1 ORN = 10^9 units
Most coins ever	100,000,000 ORN
Half-life of the release	126,230,400 s (4 × 365.25 days)

Parameter	Value
Coins at the start	0
Epoch	3,600 s, or 50,000 payments
Quorum	more than two thirds of the weight ($3w > 2 \cdot total$)
Validators	at most 33
Minimum stake	10,000 ORN
Unstaking	168 epochs (about a week)
Launch seats	3,000,000 ORN of weight, shared equally, fading as stake arrives
Fee	0.1% of the amount, at least 0.000001 ORN, at most 1 ORN; stake and unstake 0.000001 ORN
Shares wanted per epoch	1,000
First target	$2^{40} - 1$
Target change per epoch	at most × 4 or ÷ 4
Hash function	BLAKE3, 32-byte output
One-time signature	WOTS, w = 16, n = 24 bytes, 51 chains, 1,248 bytes
Validator key tree	height 16 by default, at most 24
Spare keys per account	16 (key tree of height 4)
Recovery phrase	24 words: 256-bit secret, 8-bit check
Wallet password	Argon2id (64 MiB, 3 passes), XChaCha20-Poly1305
Connection limits	1,024 in all, 64 per address, 30 s idle
States a validator keeps	the last 3 epochs

The domain tags that separate every use of the hash function:

Tag	Use	Tag	Use
0x01	one-time chain step	0x34	confirmation
0x02	one-time commitment	0x35	address

Tag	Use	Tag	Use
0x03	one-time message	0x36	proposer draw
0x10	key tree node	0x37	payment log pin
0x20–0x22	batch tree leaf, node, padding	0x38	statement
0x30	order	0x39	checkpoint
0x31	batch message	0x40–0x42	state tree leaf, node, padding
0x32	genesis (network id)	0x43	ballot
0x33	account	0x44	proposal
0x45	stake proof	0x47	software release
0x46	proof-of-work share	0x48, 0x49	recovery phrase check, account seed

17 References

1. M. Baudet, G. Danezis, A. Sonnino. FastPay: High-performance Byzantine fault tolerant settlement. ACM Advances in Financial Technologies, 2020.
2. E. Buchman, J. Kwon, Z. Milosevic. The latest gossip on BFT consensus. arXiv:1807.04938, 2018.
3. NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard (SLH-DSA). 2024.
4. R. Merkle. A certified digital signature. CRYPTO 1989. (The Winternitz one-time signature is described there.)
5. Borsh: Binary Object Representation Serializer for Hashing. borsh.io.
6. J. O'Connor, J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn. BLAKE3: one function, fast everywhere. 2020.
7. R. Merkle. Protocols for public key cryptosystems. IEEE Symposium on Security and Privacy, 1980.
8. A. Biryukov, D. Dinu, D. Khovratovich, S. Josefsson. RFC 9106: Argon2 memory-hard function for password hashing and proof-of-work applications. 2021.