



ZeroTrace

A private vault for a tradable token, paid by its trading fees

Technical whitepaper — Draft v0.3 — September 2026

ZeroTrace contributors

Abstract

ZERO is an ordinary ERC-20 launched on the Pons V2 launchpad on Robinhood Chain: it trades on its Pons bonding curve and, after graduation, on a Uniswap v4 pool, and every wallet, exchange and aggregator can handle it without special support. ZeroTrace is its private side. Shielding moves ZERO into a vault as an encrypted note; inside, notes are sent and sold with zero-knowledge proofs, so amounts, senders and recipients stay hidden. Notes hold shares of the vault, and our fee harvester, which Pons pays as ZERO's creator fee recipient, splits the creator share of every trade fee in half: one half goes to the team, the other is turned into ZERO for the vault, so the holder yield accrues to private holders only, without revealing any balance. Spends are authorised by a single Noir circuit, proved with UltraHonk on the user's device, which also verifies an EIP-712 signature by the owner's wallet key. The vault is its own ERC-4337 account: users pay a flat network fee in shares from the notes they spend, which is donated to the other notes, so no private transaction is signed or paid for by the user's address. Our contracts cannot be upgraded, and no role can move, freeze or block a note: the vault's guardian can only stop new deposits, and the harvester's owner can only propose, with a public 24-hour delay, a new recipient for the trading fees. This paper describes the construction, its security and privacy properties, and its limits.

Contents

1	Introduction	3
2	Design goals	3
3	System overview	4
4	Cryptographic construction	5
4.1	Primitives and notation	5
4.2	Notes, commitments and nullifiers	6
4.3	Keys	7
4.4	The spend circuit	7
4.5	The note tree	8
5	The vault and the holder fee	8
6	ZERO and shielding	9
7	The market and the fee stream	9
7.1	The Pons market	9
7.2	The holder fee	10
7.3	Price average	11
7.4	Private router	11
8	Gas abstraction with ERC-4337	11
9	ZERO on Pons	13
10	Security analysis	14
10.1	Trust assumptions	14
10.2	Attacks and defences	15
10.3	Accepted limits	15
11	Privacy analysis	16
12	Robinhood Chain	16
13	Implementation and measurements	17
14	Related work	17
15	Conclusion	17
A	Deployment	18
B	Constants	18

1 Introduction

On a public blockchain every balance and every payment is visible forever. Paying someone reveals your balance and history to them, and to everyone else. For ordinary users this has concrete consequences: visible wallets become targets, trading activity is copied or front-run, and counterparties can rebuild a person's finances.

Privacy on public ledgers is only as strong as the set of users one hides among, the *anonymity set*. Earlier designs made that set small in one of two ways:

- **Optional privacy.** When privacy is a mode users must opt into (for example transparent and shielded addresses in Zcash [1], or a separate shielded pool for existing tokens as in Railgun [3]), most value stays public for convenience. The few who opt in are easier to identify, and choosing privacy itself becomes a signal.
- **Fixed denominations.** Mixers such as Tornado Cash [2] split the crowd by deposit size and are used almost exclusively to break links, which concentrates misuse.

ZeroTrace takes a different position: it does not try to make every holder private by rule, it makes holding in private the only way to earn from trading. The asset is a single fungible token that trades like any other, and the private side pays. Three properties summarise the design:

1. **Trades anywhere.** ZERO is a plain ERC-20 created by Pons. Its market, a bonding curve and then a Uniswap v4 pool with locked liquidity, is already routed by aggregators; wallets and exchanges need nothing special.
2. **Private when shielded.** A note in the vault hides its amount and owner; private sends, sales and unshields reveal neither sender nor amount, and are never signed or paid for by the user's address. Buying through the ZeroTrace router lands straight in a note.
3. **Privacy is where holding pays.** Half of the creator share of every trade fee is harvested into the vault that backs the notes, and every private operation's network fee is donated to it. Public ZERO receives nothing.

Section 3 gives an overview. Section 4 defines notes, keys and the spend circuit. Sections 5 to 8 cover the vault, shielding, the market and the fee stream, and gas abstraction. Section 9 describes ZERO on Pons and the deployment, Sections 10 and 11 the security and privacy analysis, and Section 13 implementation and measurements.

2 Design goals

Six principles drive the design. When two conflict, the earlier one wins.

1. **Nobody loses funds, and anyone can always exit.** Contracts cannot be upgraded, and no role can move, freeze or block a note: the two roles that exist (Section 3) can only stop new deposits or propose a new fee recipient in public. Notes are always recoverable from chain data. Selling works without the protocol's gas deposit, its app or its bundler of choice.
2. **Private is where it pays.** Public ZERO works everywhere but earns nothing; only notes receive the holder yield and the network fees.
3. **Compatible with the outside world.** Any router or aggregator can buy the token, and token scanners should find nothing that resembles a honeypot.
4. **As few dependencies as possible, and none that can touch the notes.** No fully homomorphic

encryption, no oracle, no committee, no dedicated relayer; bundlers, indexers and interfaces can be run by anyone. Pons runs the market and the fee stream (Section 10).

5. **Invisible to the user.** One signature on first use, no extra seed phrase, no gas paid from the user's address for private operations.
6. **Defensible.** Public supply, no team allocation. The team's only revenue is half of the creator fees, paid in the open by the harvester; it takes nothing from private operations.

One dependency cannot be removed: the Robinhood Chain sequencer decides which transactions enter blocks (Section 12).

3 System overview

The core is a single contract, PrivateVault. It holds ZERO for the notes, a registry of privacy keys, an append-only Merkle tree of note commitments, the set of spent nullifiers, the vault totals, the network fee and a price average. It verifies every spend, and it is itself an ERC-4337 [10] account. Two small contracts of ours surround it, next to the Pons launchpad [9] and Uniswap v4 [8] (Fig. 1 and Table 1).

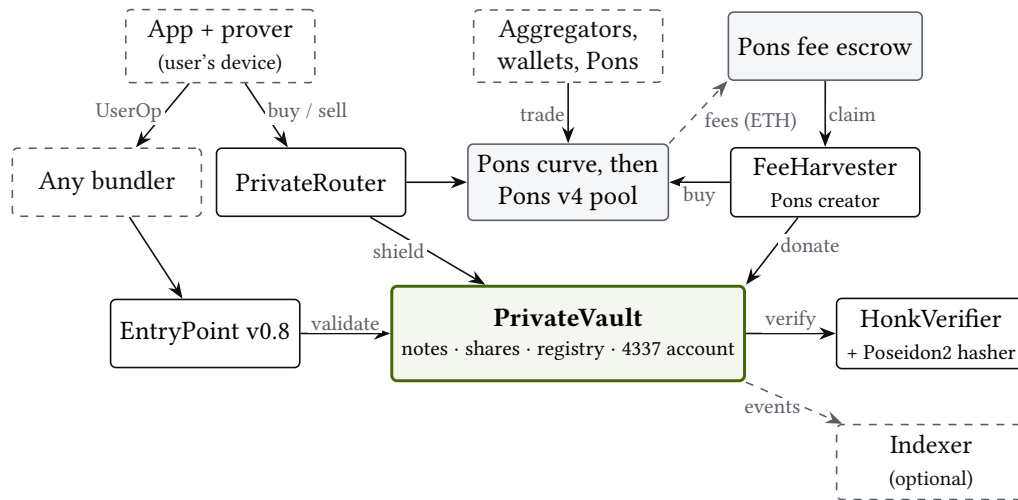


Figure 1: Our contracts (solid), Pons (grey) and off-chain parts anyone can run (dashed).

Component	Role
ZERO	Plain ERC-20 created by the Pons factory: OpenZeppelin ERC20 and ERC20Burnable, no owner, 10^9 supply.
Pons curve and pool	The bonding curve ZERO trades on until graduation, then a Uniswap v4 pool with the Pons hook and locked liquidity. Pons charges its fees on both.
PrivateVault	Holds ZERO for the notes. Key registry, note tree and nullifiers, vault totals B and S , network fee, price average. Verifies every spend. Acts as the ERC-4337 account for private transactions.
FeeHarvester	ZERO's Pons creator fee recipient. Sends half of the creator fees to the team; with the other half it tops up the vault's gas deposit and spends the rest on ZERO donated to the vault. Callable by anyone, who is refunded only the gas of useful work.
PrivateRouter	buy: buys ZERO and shields it in one transaction. sell: spends notes and sells for ETH. The exit target of private sales made through ERC-4337. quote: output of a trade, Pons fees included.
HonkVerifier	UltraHonk verifier of the transact circuit, generated by Barretenberg.
Poseidon2 hasher	Poseidon2 over BN254 in Yul, outside the vault for contract size.
Bundler, indexer, app	Off-chain; anyone can run them. The indexer serves the same event ranges to everyone and is checked against on-chain roots.

Table 1: Components.

There is no proxy, upgrade path or initializer: every address a contract works with is fixed at construction, and the addresses of our contracts are known before deployment (Section 9).

Roles. Two roles exist, both given to the deployer at deployment, both transferable (for instance to a multisig) and renounceable, after which they are gone for good:

- The vault's *guardian* can pause and resume shield, and so new deposits and private buys. Spends, transfers, sales, exits and the release of a stuck sale are never affected, so nobody can be locked in.
- The harvester's *owner* can change who receives the team's half of the creator fees, and propose a successor as ZERO's Pons creator fee recipient. The proposal is public and anyone can carry it out only after 24 hours; the owner can cancel it before then. It lets a future fix of the harvester take over the fee stream without anyone at Pons, and holders see it a day ahead.

Neither role can touch a note, the vault's ZERO, the holders' half of the fees, the network fee, the verifier or any parameter.

4 Cryptographic construction

4.1 Primitives and notation

Let p be the BN254 scalar field modulus,

$$p = 0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001,$$

and let H denote the Poseidon2 [4] sponge over $\mathbb{F}_p$ as computed by Poseidon2::hash of Noir's poseidon library. The definitions below are implemented in the Noir circuit and in the client, and the ones the contract needs (stub, commitment and tree nodes) also in the Solidity library NoteHash; all must agree. keccak256 is Keccak-256. Encryption uses X25519, HKDF-SHA256 and XChaCha20-Poly1305. Wallet keys are secp256k1.

4.2 Notes, commitments and nullifiers

A *note* is an amount of vault shares owned by one key. It consists of an owner key opk , a random value ρ , a blinding value r , and a share amount $s < 2^{120}$. The chain never stores a note, only its commitment:

$$\begin{aligned} \text{stub} &= H(opk, \rho, r), \\ \text{cm} &= H(1, \text{stub}, s), \\ \text{nf} &= H(2, nk, \text{cm}, i), \end{aligned}$$

where the constants 1 and 2 separate domains, nk is the owner's nullifier key and i is the leaf index of the commitment in its tree. Spending a note publishes nf , which marks it spent without identifying cm .

The *stub* lets a note be created before its share amount is known. `shield(stub, amount, ciphertext)` takes only the *stub*; the contract converts the amount to shares at execution time and completes cm itself. This is how a purchase swaps and shields in one transaction.

Positional nullifiers. Binding the leaf index into nf defeats the *faerie gold* attack [1], in which a sender creates two notes with identical contents that can only be spent once: two identical commitments at different leaves have different nullifiers.

Encryption. The client gives every note it creates a 286-byte ciphertext for its owner, published in the note's `NoteAdded` event; the contract does not read it:

$$\begin{aligned} k_{\text{sh}} &= X25519(x_e, vpk), & k &= \text{HKDF}(k_{\text{sh}}, \text{salt}=E, \text{"zerotrace/v1/note-key"}), \\ \text{tag} &= \text{SHA256}(k_{\text{sh}} \parallel \text{"zerotrace/v1/view-tag"})[0], \\ \text{ct} &= E \parallel \text{tag} \parallel n \parallel \text{XChaCha20Poly1305}_k(n, \rho \parallel r \parallel s \parallel m), \end{aligned}$$

with a fresh ephemeral key pair (x_e, E) per note and a 24-byte nonce n . The one-byte view tag lets a wallet skip 255 of 256 foreign notes without a full decryption. A shielded note's shares are only known on-chain, so its ciphertext carries $s = 0$ and the wallet reads the shares from the `Shielded` event; an empty output of a spend gets 286 random bytes, so every output looks alike. A wallet accepts a decrypted note only if it recomputes the commitment in the event, so a malformed ciphertext cannot fake a note; at worst the recipient cannot find it.

Spend memo and proof of payment. The 117-byte memo m is all zero except in a spend's change note, which goes back to the sender even when the change is zero. There it records whom the spend paid and, for a private send, the opening (ρ, r, s) of the recipient's note. Only the sender's viewing key reads it, so their history shows every recipient; the recipient's note says nothing about the sender, and every output ciphertext has the same length. The sender can reveal that opening as a proof of payment: anyone can check that the transaction created a leaf equal to $H(1, H(opk, \rho, r), s)$ for the recipient's registered opk . Spending the note or seeing when it is spent still needs the recipient's nk .

4.3 Keys

ZeroTrace has no separate seed phrase. The wallet signs one Sign-In with Ethereum [13] message that names the app's domain, the address, the chain and the vault. Every field is fixed, including the issue time and a nonce derived from the vault address, so the same wallet always signs the same text. From the signature σ :

$$\begin{aligned} seed &= \text{keccak256}(\sigma), \\ sk &= \text{HKDF}(seed, "zerotrace/v1/sk", 64 \text{ bytes}) \bmod p, \quad nk = H(sk), \\ (x, y) &= \text{secp256k1 public key recovered from } \sigma, \\ opk &= H(nk, x_{hi}, x_{lo}, y_{hi}, y_{lo}), \\ vsk &= \text{HKDF}(seed, "zerotrace/v1/view", 32 \text{ bytes}), \quad vpk = \text{X25519}(vsk), \end{aligned}$$

where x_{hi}, x_{lo} are the 128-bit halves of x , and likewise for y . The owner key therefore commits to both the nullifier key and the wallet's own public key.

The keys can be recovered on a new device only if the wallet signs deterministically (RFC 6979). On first use the app requests the signature twice and refuses to register if the two derivations differ. A user makes the keys usable for incoming payments by publishing (opk, vpk) in the vault's registry, directly with register or through an EIP-712 authorisation submitted by anyone (registerFor).

Two-factor spending. The activation signature alone does not allow spending. Every spend must carry a fresh EIP-712 [12] signature by the wallet key committed in opk :

$$\text{Spend}(\text{bytes32 } transaction), \quad transaction = H(root, nf_0, nf_1, cm_0, cm_1, exit, extDataHash),$$

under the vault's domain (the token name, version "1", chain ID, vault address). A leaked activation signature reveals the owner's notes and their spends, but cannot move them.

4.4 The spend circuit

Every private spend (send, sale or unshield) is one proof of the transact circuit, written in Noir [5]. It consumes two input notes and creates two output notes; missing ones are zero-value dummies.

Public inputs. Nine field elements, in order: a Merkle root; two nullifiers; two output commitments; the exiting shares e ; $extDataHash = \text{keccak256}(\text{abi.encode}(ext)) \bmod p$; and the two 128-bit halves of the vault's EIP-712 domain separator. The contract computes the last three from calldata and its own state; the prover does not choose them.

Constraints. For each input $j \in \{0, 1\}$ with private witness $(nk_j, \rho_j, r_j, s_j, i_j, \pi_j)$ and the shared wallet key (x, y) :

$$\begin{aligned} s_j &< 2^{120}, \quad i_j < 2^{24}, \\ cm_j &= H(1, H(H(nk_j, x_{hi}, x_{lo}, y_{hi}, y_{lo}), \rho_j, r_j), s_j), \\ s_j \neq 0 &\Rightarrow \text{MerkleRoot}(cm_j, i_j, \pi_j) = root, \\ nf_j &= H(2, nk_j, cm_j, i_j). \end{aligned}$$

For each output k : $s'_k < 2^{120}$ and $cm'_k = H(1, stub'_k, s'_k)$. Conservation and range:

$$e < 2^{120}, \quad s_0 + s_1 = s'_0 + s'_1 + e.$$

The range checks prevent the sum from wrapping the field. Finally the circuit recomputes the EIP-712 digest $\text{keccak256}(0x1901 \parallel domain \parallel \text{keccak256}(\text{SPEND_TYPEHASH} \parallel transaction))$ and verifies an ECDSA secp256k1 signature on it against (x, y) , without revealing the key.

External data. The structure *ext* carries the exit recipient, an optional required submitter (caller), the network fee in shares, caller-specific parameters and the two output ciphertexts. For a spend through the vault’s ERC-4337 account those parameters are a minimum call gas limit and, for an exit to a contract, its *exit target*, the recipient of last resort and the target’s own data (for a sale, the minimum ETH out; [Section 8](#)). Because its hash is a public input covered by the wallet’s signature, no relayer or bundler can redirect an exit, change its target, raise the fee, loosen the sale terms or replace a ciphertext.

Checks in the contract. The vault additionally requires that the root is known ([Section 4.5](#)), that the two nullifiers differ and are unspent, that all public values are below p , and that the network fee does not exceed the exit.

Proof system. Proofs use UltraHonk [6] with a Keccak transcript and the zero-knowledge variant, verified by a generated Solidity verifier. UltraHonk needs no circuit-specific trusted setup; it uses a universal reference string, which the app serves itself; its files are checked against pinned hashes when they are added to the app. Groth16 [7] would verify for less gas but requires a per-circuit ceremony and was not adopted.

4.5 The note tree

Commitments are leaves of an append-only Poseidon2 Merkle tree of depth 24 ($2^{24} \approx 16.8$ million leaves). Leaves are inserted in pairs at even indices: a transact inserts its two outputs, and a single note is padded with an empty leaf, which leaves the root as if the note were inserted alone. A pair costs 24 hash evaluations instead of 48. The last 64 roots are accepted, so proofs in flight survive concurrent insertions. When a tree fills up, its final root is recorded as valid forever and a new tree starts; notes in a full tree remain spendable.

5 The vault and the holder fee

Notes hold *shares* of the vault, in the style of ERC-4626 [15]. Let B (totalBacking) be the ZERO backing all notes, S (totalShares) the shares held by unspent notes, and $V = 10^6$ a constant of virtual shares. Conversions round down, in favour of the vault:

$$\text{toShares}(a) = \left\lfloor \frac{a(S + V)}{B + 1} \right\rfloor, \quad \text{toAssets}(s) = \left\lfloor \frac{s(B + 1)}{S + V} \right\rfloor.$$

Shielding a tokens adds a to B and $\text{toShares}(a)$ to S . Exiting e shares, g of which pay the network fee, removes $\text{toAssets}(e) - \text{toAssets}(g)$ from B and e from S : the fee’s ZERO stays in B , a donation to every remaining note ([Section 8](#)).

Donations. `donate(amount)` adds ZERO to B and leaves S unchanged, so the value of every share rises. Value reaches the notes in two ways: the harvester donates the ZERO it buys with the holders’ half of the Pons creator fees ([Section 7](#)), and every private operation leaves its network fee in B ([Section 8](#)); both emit `Donation`, the second with the vault itself as donor. Because donations raise the share price uniformly, every private balance grows in proportion to its shares without any transaction and without revealing amounts. Public ZERO holds tokens, not shares, and receives nothing.

Invariants. Every code path that moves tokens subtracts from one bucket and adds to another, so

$$\text{zero.balanceOf(vault)} = B + \sum \text{pendingSales},$$

except for ZERO sent by a plain transfer, which the permissionless absorb, called on every harvest, adds to B as a donation. Because both conversions round down, the share value $(B + 1)/(S + V)$ never decreases. B and S are public and every change emits `VaultUpdated`, so anyone can monitor them.

First-depositor attack. In an empty share vault an attacker can inflate the share price so that the next depositor's shares round to almost nothing. The virtual shares make such an attack cost more than it can take. In addition, the deployer buys a first amount of ZERO and shields it into a note whose stub is `keccak256("zerotrace/locked-seed") mod p`. No preimage is known, so the note can never be spent. The seed also absorbs fees donated before the first shield, which would otherwise dilute the first holder.

6 ZERO and shielding

ZERO has no transfer logic of its own: it is an OpenZeppelin ERC-20, and anyone holds, sends and trades it. Privacy starts only inside the vault.

Action	Effect
Buy through <code>PrivateRouter.buy</code>	ZERO bought on the market and shielded into a note in the same transaction.
Buy anywhere else	Public ZERO in the buyer's wallet.
<code>shield(stub, amount, ciphertext)</code>	Pulls public ZERO (after an approval) into a note.
Private send, sale, unshield	A spend proof; the vault sends ZERO out only on an unshield or a sale.

Pausing entries. The vault's guardian ([Section 3](#)) can pause shield, which also stops private buys, since the router shields what it buys. It is meant for stopping new money while a problem is examined; it never stops a spend, so every note can still be sent, sold or unshielded.

Public ZERO earns nothing, so it is an economic place to pass through rather than to hold. The price of that simplicity is that privacy is a choice: a holder who never shields stays public, and the anonymity set is made of those who choose it. The fee stream ([Section 7](#)) is what makes the choice worth making.

7 The market and the fee stream

7.1 The Pons market

Pons V2 mints the whole supply into a bonding curve for ZERO. Buyers take 5/7 of the supply from the curve. Once the curve has collected its graduation threshold (4.2 ETH for a native-ETH launch), Pons graduates the token in two permissionless steps: graduate sweeps the curve, and `createGraduatedPool` seeds a full-range Uniswap v4 pool with locked liquidity. Our router and harvester perform whichever step is pending before they trade.

The graduated pool's key is native ETH as `currency0`, ZERO as `currency1`, the launch's pool fee (0) and tick spacing (200), and the factory's own hook. The hook's `beforeInitialize` accepts only the Pons factory, so nobody can open a pool with that key first; any other ETH/ZERO pool has another id and is never traded on or priced by our contracts.

7.2 The holder fee

Pons charges a standard fee of 1% on every trade, the same on the curve and on the pool, in the quote asset (ETH). A launch may add a creator tax on top; ZERO has none, so trading it costs the same as any other Pons token. Pons keeps 30% of the standard fee; the rest goes to the creator fee recipient, our harvester, through the Pons fee escrow, and the harvester splits it in half between the team and the private holders:

Share of each trade	With the defaults
Paid by the trader	1.0%
Kept by Pons	0.3%
To the harvester	0.7%
of which to the team	0.35%
of which to private holders	0.35%

`FeeHarvester.harvest`, callable by anyone, sweeps the fees Pons holds for ZERO, claims them from the escrow and sends half of what it claimed to the team. With the holders' half it keeps the vault's gas deposit at its target ([Section 8](#)) and spends the rest on ZERO, which it donates to the vault. The team's half is never spent on gas or ZERO; if its recipient refuses the ETH, it stays owed and the harvest goes on.

Small buys. The holders' half buys ZERO in small steps: at most one buy per L1 block, each sized from the market's depth to move the price by no more than 0.5%, and never more than 15% dearer than the vault's price average, after fees. When the market does not take a buy within these limits, the harvester halves it, trying up to eight sizes, and what does not fit waits for the next block. A buy that small is not worth sandwiching: pushing the price up before it and selling after costs the Pons fees on both legs, about 2%, for at most the 0.5% the buy moves the price. Someone who buys ahead of the harvester and holds through several blocks of its buys can still sell higher, like anyone who buys ahead of a buyback, at the risk of holding a pumped token in the open market meanwhile.

The caller's refund. The caller is refunded only the gas of a call that put ETH to work, by topping up the deposit or buying ZERO: the gas the call measured plus a fixed 25,000 for the transaction's base cost, priced at no more than twice the base fee, and never more than 0.5% of the ETH put to work. A call that does nothing pays nothing. So calling harvest in a loop, for instance a second time in the same block, or while a buy is skipped because the market is too far from the average, earns nobody anything, and the fees cannot be drained by repeated calls.

Successor. The harvester's owner ([Section 3](#)) can propose a successor; after 24 hours anyone can execute the proposal, which calls Pons' `transferCreatorFeeRecipient` so the successor receives the fees from then on. What Pons had already credited to the harvester stays claimable by harvest.

7.3 Price average

The vault keeps an exponential moving average of the price in ZERO per ETH, used to set the network fee (Section 8) and to bound the harvester’s trades. A permissionless poke, which the router and the harvester call on every trade, reads the live price (the curve’s reserves before graduation, the pool’s price after) and updates it:

Algorithm 1: Price average update, on every poke

Input: live price x ; state ema , $lastSpot$, $lastBlock$

```

1 if block.number  $\neq$  lastBlock then
2    $c \leftarrow \text{clamp}(lastSpot, 0.9\ ema, 1.1\ ema)$ ; ▷ previous update
3    $ema \leftarrow (7\ ema + c)/8$ ;
4  $lastSpot \leftarrow x$ ;  $lastBlock \leftarrow$  block.number;
```

With a clamp of $\pm 10\%$ and weight $1/8$ the average moves at most 1.25% per block, however far the price was pushed. On Robinhood Chain block.number is the Ethereum L1 block number, roughly 12 seconds apart. The first poke after the launch seeds the average.

The price folded for a block is the one recorded at its last poke. Since anyone can poke, a price pushed, poked and pushed back within one transaction still counts, within the clamp. Lowering the average by 20% therefore takes about 18 consecutive L1 blocks, and raising it by 20% about 15, each paying the Pons fees on both legs of a push large enough to move the price by 10%. What such a move can win is bounded in Section 8.

7.4 Private router

PrivateRouter holds no funds between transactions. buy buys ZERO with ETH and shields it into a note from a user-supplied stub, so a purchase through the app lands in a note whose owner is not visible on-chain. A private sale reaches the router in one of two ways, and both sell the exit for ETH sent to a recipient bound in the proof:

- sell takes a proof whose external data names the router as both caller and exit recipient, with the ETH recipient and the minimum ETH out as its parameters, and is paid for by whoever submits it;
- through the vault’s ERC-4337 account the router is the sale’s *exit target*: the vault calls its onPrivateExit, and the router pulls the ZERO it was approved for and sells it (Section 8).

quote computes the curve’s output from its reserves and fee rates, or runs the pool swap and reverts, so its result includes the Pons fees. The app sets minimum outputs to the quote less 1%; for sales the minimum is part of the proof’s external data.

8 Gas abstraction with ERC-4337

A private transaction must not be signed or paid for by the user’s address, which would link the address to the spend. PrivateVault is therefore an ERC-4337 account. Any bundler submits a UserOperation whose sender is the vault; the canonical EntryPoint v0.8 pays the bundler from the vault’s deposit; and the user pays a flat network fee in shares out of the notes being spent.

Validation performs the transition. validateUserOp decodes the transaction and proof, checks the fixed operation fields (no init code, no paymaster, call data naming the transaction’s first nullifier), enforces the minimum call gas limit bound in the proof, checks the cost cap and the fee, and then applies the entire private state transition: root, nullifiers, proof, insertion of both outputs, the network fee left in B , and the exit either paid out at once or held in escrow for its target. Execution only settles exits

to a target. Because every state change happens in validation, a failing execution cannot undo it, and nobody can make the deposit pay for an operation that did nothing. The nonce key is derived from the first nullifier, so operations from different users never queue behind each other.

The network fee. Every operation pays the same fee, worth a fixed amount of ETH set at deployment, $c_{\max} = \text{maxOpCostWei}$ (0.0005 ETH). The vault turns it into ZERO with the price average,

$$\text{networkFee} = c_{\max} \cdot \text{ema} / 10^{18},$$

and poke updates it at most once an hour (NETWORK_FEE_INTERVAL). The fee it replaces stays accepted until the next update, and validation requires

$$\text{toAssets}(\text{gasFee}) \geq \min(\text{networkFee}, \text{previousNetworkFee}).$$

A wallet prices the fee once, from the current value, and signs it into the proof. However the market moves before inclusion, the operation stays valid at least until the second update after signing, an hour or more later. Validation reads only the vault's own storage, as ERC-7562 [11] requires; the market is never read there.

The cost cap. The proof does not fix the gas fields, so bundlers can estimate and price the operation. Validation caps what it may cost instead:

$$\text{maxCost} = (\text{verificationGas} + \text{callGas} + \text{preVerificationGas}) \cdot \text{maxFeePerGas} \leq c_{\max}.$$

The cap and the fee have the same value on purpose: the deposit never pays more for an operation than the operation pays in ZERO at the average. A bundler that fills bundles with its own operations at an inflated gas price cannot draw more out of the deposit than it puts into the vault. When gas costs more than the cap allows, the vault refuses operations and the app falls back to a direct exit (below).

Who pays what. The fee's shares are burned and its ZERO stays in B , so the whole fee is a donation to the remaining notes (Section 5). The gas itself is paid in ETH by the deposit, which the harvester refills with the holders' half of the creator fees before it buys anything (Section 7). Since an operation's gas never exceeds the cap and its fee is worth the cap, the holders never pay more for an operation than it pays them.

Moving the average. Lowering the average lowers the fee in ZERO, while the cap stays in ETH. The most an attacker can win, running bundles of its own operations at the cap, is the gap between the two on each operation, for the up to two hours the lowered fee is accepted. Lowering the average by 20% takes about 18 consecutive L1 blocks of manipulation (Section 7.3), each paying Pons' fees on both legs of the push, and every operation must spend a note of the attacker's own; the gain is then at most 20% of $c_{\max}$, 0.0001 ETH, per operation, against a deposit of 0.1 ETH. Raising the average only makes operations dearer, and users may still pay the previous, lower fee. We accept this bound.

Estimation. An invalid proof does not revert: `validateUserOp` returns `SIG_VALIDATION_FAILED` and writes nothing, as ERC-4337 expects of a bad signature. A wallet can therefore obtain gas estimates with a placeholder proof of the right size, and then sign once and prove once with the final gas values.

Exits to a target. A swap cannot run in validation, nor can any call to another contract. An exit through the ERC-4337 account may therefore name an *exit target*, in external data bound by the proof and the wallet’s signature:

$$ext.data = abi.encode(minCallGasLimit, target, recipient, targetData).$$

Validation holds the exit’s ZERO in escrow, keyed by the first nullifier. Execution approves exactly that amount to the target and calls its `onPrivateExit(amount, recipient, targetData)`; the target pulls what it uses. The vault then reads what the target left from the remaining allowance, clears the approval, and sends the rest to the recipient as ZERO; if the target reverts, the recipient receives everything. If execution runs out of gas, anyone can release the escrow to the recipient (`claimSale`). A target can take at most the exit, and never anything else.

The router is the target of private sales: it sells what it pulls and sends the ETH to the recipient, with the minimum out as its target data (Section 7). Any contract implementing `IExitTarget` can be a target, and only a proof signed by the note’s owner can name one. This is how a migration would work, should one ever be needed: a new vault exposing `onPrivateExit` could take a note’s ZERO and shield it there, one owner at a time and only at their request, with no change to this vault. Without a target, the exit (an unshield) is paid to the recipient in validation, as in `transact`.

Gas deposit. The deployer funded the `EntryPoint` deposit with 0.1 ETH at deployment, and the harvester keeps it at that target with the first ETH of every harvest. Nobody can withdraw the deposit or the `EntryPoint` stake.

Guaranteed exits. If no bundler serves the vault, the deposit is empty, or gas costs more than the cap allows, a spend can be submitted directly to `transact`, or a sale to `PrivateRouter.sell`, with gas paid by an address of the user’s choice and no network fee. If the router is unusable, notes can be unshielded and the ZERO sold on any venue. None of these paths depends on the guardian, who can only pause shield.

9 ZERO on Pons

Parameter	Value
Token	Plain ERC-20 created by Pons V2, no owner
Creator fee recipient	<code>FeeHarvester</code>
Total supply	10^9 ZERO (18 decimals), fixed, minted into the Pons curve
Team allocation, treasury, reserve	None
Team revenue	Half of the creator fees, through the harvester
Graduation	At 4.2 ETH collected: a locked Uniswap v4 pool
Fees	1% standard (30% to Pons), no creator tax, from the first block
Buybacks	Off
Snipe protection	Pons’ opening tax, 99% decaying to 0 over 5 seconds
Locked seed	A purchase by the deployer, in an unspendable note

Table 2: ZERO’s terms on Pons.

Every ZeroTrace contract is deployed through CreateX’s `CREATE3` with a salt guarded by the deployer’s address and the chain ID, so its address is fixed before deployment and nobody else can deploy at it. ZERO was therefore launched first, naming the harvester’s planned address as its creator fee recipient.

Pons' record of the token, `getLaunchedToken(zero)`, names the harvester as creator fee recipient, with ETH as the pair, no creator tax and buybacks off; anyone can read it. The deployment script refuses to deploy unless that record matches, names the launch wallet and finds the harvester's address still empty, and the harvester's constructor itself reverts for a token that is not a Pons launch paired with ETH. The wallet that made the launch has no special rights over ZERO: only Pons' owner can redirect the creator fees, after a 3-day timelock, and the harvester's own successor procedure is public and delayed (Section 3). The deployer bought the locked seed once the snipe tax was over. The team can hold ZERO only by buying it.

ZERO launched on 2 October 2026. Its curve filled within seconds and graduated to the Uniswap v4 pool before our contracts were deployed; the creator fees earned in between were credited to the harvester's address by Pons and collected by its first harvest, so nothing was lost.

10 Security analysis

10.1 Trust assumptions

Soundness rests on (i) the circuit enforcing membership, ownership, correct nullifiers, conservation of shares, range checks and a valid wallet signature; (ii) the generated verifier matching the circuit, with no soundness bug in Noir or Barretenberg; (iii) collision resistance of Poseidon2 and Keccak-256 and unforgeability of secp256k1 ECDSA; and (iv) honest generation of the universal reference string. A failure of any of these could allow shares to be created from nothing, which would dilute every holder. This is the most critical assumption in the system.

Both roles (Section 3) are held by the deployer at launch, until they are handed to a multisig or renounced.

Party	Can	Cannot
Guardian	Pause and resume shield: new deposits and private buys	Stop a spend, transfer, sale or exit; move or freeze a note; change the fee or any parameter; upgrade
Harvester owner	Change the team's recipient; propose a new creator fee recipient, which anyone can carry out after 24 hours, and cancel it before	Redirect the fees at once; touch the holders' half, the vault, the notes or their ZERO
Pons owner	Redirect the creator fee recipient after a 3-day public timelock, which would stop the holder yield; use its rescue functions on curves and graduations	Touch the vault, the notes or their ZERO
Holder of the activation signature	See all the owner's notes and history	Spend them
Compromised frontend	Read notes; ask the user to sign a harmful spend	Spend without the wallet's signature
Bundler	Censor; see IP and timing	Change recipient, fee or amounts; starve execution
Indexer	Withhold data; see IP and timing	Learn which notes are whose; alter the tree undetected
RPC node	Censor; lie about reads	Make an invalid transaction succeed
Sequencer	Censor or delay	Forge, steal or alter a transaction

Table 3: What each party can and cannot do.

10.2 Attacks and defences

Malicious bundler. All external data is hashed into a public input covered by the wallet's signature, and a minimum call gas limit is bound in the proof.

Deposit draining. State changes happen in validation; invalid proofs write nothing and are not executed; an operation's maximum gas cost may not exceed the value of the network fee it pays.

Price manipulation. The average is clamped and weighted and moves at most 1.25% per L1 block, and the network fee follows it at most once an hour. Lowering the fee by 20% takes about 18 consecutive manipulated blocks, each paying Pons' fees on both legs of the push, and wins at most 20% of the cap per operation (Section 8). The harvester buys at most once per block, moving the price by at most 0.5%, and never more than 15% above the average.

Malicious exit target. A target is named only by a proof the note's owner signed. It is approved for exactly the exit, and whatever it does not take, or everything if it reverts, goes to the owner's recipient.

Fee stream takeover. Moving ZERO's creator fees to another contract takes a public proposal by the harvester's owner and 24 hours before anyone can carry it out; Pons' own redirection needs a 3-day timelock.

Sandwiching. Minimum outputs from an exact on-chain quote; for sales the minimum is bound in the proof. Robinhood Chain orders transactions by arrival, which makes sandwiches harder.

Decoy pools. Only the Pons factory can initialize a pool with its hook, so the graduated pool's key cannot be taken first; any other pool has another id and is ignored.

Tree exhaustion. Full trees roll over and their final roots stay valid.

Lying indexer. The client rebuilds the tree, checks the root with `isKnownRoot`, and rereads the chain on a mismatch. Funds are never at risk from the indexer.

Phishing for the activation signature. The SIWE message names the app's domain, so wallets warn on other sites; and the activation signature cannot spend.

10.3 Accepted limits

Immutability means a bug cannot be patched; the only remedy is a voluntary migration, one note at a time through an exit target (Section 8), with new deposits paused by the guardian meanwhile. The fee stream depends on Pons: its owner can redirect it after a timelock, and Pons keeps 30% of the standard fee. The fees Pons collects in ZERO on the graduated pool are converted to ETH by a sweep that only Pons' operator can run, which sells that ZERO on the pool; the harvester buys it back in small steps over the following blocks. The network fee is flat: a private send pays as much as a sale, and more than the gas it uses; the difference goes to holders. While gas costs more than the cap allows, the ERC-4337 path is closed and exits go direct. A sale can be released to its recipient's public balance by another operation in the same bundle before it executes; the caller gains nothing. Contract-based multisigs cannot own notes, because the circuit verifies one EOA signature; EIP-7702 [14] accounts and MPC wallets that output one ECDSA signature are supported.

11 Privacy analysis

Inside the system, proofs hide who pays whom and how much. Entries into and exits from the private side are public, and their protection comes from the number of users.

Information	Public
Total supply, B , S , network fee, price average	yes
Trades on the curve and the pool: amounts, price, time	yes
Public ZERO balances and transfers	yes
Registered addresses and their public keys	yes
Buyer's address and amount of a purchase	yes
Shields: address and amount	yes
Recipient and amount of an exit; its target; ETH recipient of a sale	yes
Which notes a spend consumed	no
Sender, recipient and amount of a private send	no
A holder's private balance	no
Whether a given note has been spent	no

Table 4: What is visible on-chain.

Anonymity set. Every note ever created lives in the same tree, and a spend proves membership without saying which leaf. Because the yield accrues only to notes, the design aims to make the private set the natural place to hold ZERO, not a destination for a minority. Early on the set is small, and a distinctive amount bought and sold soon after is easy to match by amount and timing. Splitting amounts, waiting, and selling to fresh addresses mitigate this.

Metadata. The RPC node, bundler and indexer can observe IP addresses and request timing. The indexer serves the same event ranges to everyone, with no per-user queries, so it does not learn which notes a wallet owns. The proving reference string is served by the app, so preparing a proof does not fetch it from a third party.

12 Robinhood Chain

Robinhood Chain is EVM compatible and permissionless for deployment. Properties that matter here: its sequencer excludes transactions tied to sanctioned addresses and orders transactions by arrival; Pons V2, Uniswap v4 and ERC-4337 EntryPoints are available (Pons only on mainnet, so the protocol is tested on forks of it); and `block.number` returns the L1 block number. The sequencer could censor transactions to our contracts. It cannot forge a proof or alter a transaction. This dependency cannot be removed.

13 Implementation and measurements

The implementation consists of Solidity contracts, a Noir circuit, a web app that proves in the browser, and an optional indexer. The contracts and the client are tested end to end on forks of Robinhood Chain against the real Pons launchpad. The circuit figures are from the current build; the gas and timing figures were measured on an earlier deployment with the same circuit and verifier, and will be measured again on mainnet:

Measure	Value
Circuit	83,781 gates (2^{17}); about 73,000 check the wallet signature
Proof size	9,152 bytes
On-chain verification	about 2.35M gas
Proving time	about 1.3 s in Node; 3–5 s in desktop Chrome
Private sale through handleOps	about 4.76M–5.25M gas
End-to-end sale with a public bundler	9.4 s, one wallet signature
Poseidon2 hash (Yul)	about 20,600 gas per call
Toolchain	nargo 1.0.0-beta.22, bb 5.0.0-nightly.20260522

Table 5: Measurements.

A Stylus implementation of the hasher was measured at about 38,300 gas per call, because each call pays program initialisation and the chain has no Stylus cache manager; the Yul hasher is used. The verifier dominates the cost of a private transaction.

14 Related work

Zcash [1] introduced shielded notes with commitments and nullifiers; ZeroTrace follows its note model and its treatment of faerie gold, but applies it to one ERC-20 on a general-purpose chain. Tornado Cash [2] and Railgun [3] are shielded pools over existing assets, where privacy is a separate destination with no reason to stay; ZeroTrace pays its private holders half of the token’s own creator fees. Confidential token standards based on fully homomorphic encryption, such as ERC-7984 [16], depend on external decryption infrastructure. Draft privacy-native token standards (ERC-8287/ERC-8302) are private from issuance but not ERC-20 compatible, so they do not trade on existing AMMs. To our knowledge, no prior design combines a tradable launchpad token, a private vault for it, a share of the trading fees that accrues only to private holders, and self-paid ERC-4337 gas, with no administrator who can touch a note.

15 Conclusion

ZeroTrace keeps ZERO an ordinary token that trades anywhere, and makes holding it in private the way to earn from its trading. Shielded ZERO becomes encrypted notes; half of the creator share of every trade fee is harvested into the vault that backs them. Every spend is a single on-device proof, authorised by the owner’s wallet, submitted by anyone and paid for from the notes themselves. The practical strength of the privacy grows with the anonymity set, and so with every holder who keeps ZERO in private.

A Deployment

Robinhood Chain, chain ID 4663. Every ZeroTrace contract is deployed through CreateX's CREATE3 with a salt guarded by the deployer's address and the chain ID, so the addresses below were fixed before deployment and nobody else could deploy at them. ZERO's and its curve's addresses were set by Pons when it created the token. All our contracts are verified on Sourcify.

Contract	Address
ZERO (Pons token)	0x316fa3AB9A8FD8d7567a823DedecF28d9FEE2894
Pons curve	0xE276F7E250bC2cD45f1e344d9adc9c9687c2c897
PrivateVault	0xf3935bec6247170B28e1755c85957A72f243D84D
FeeHarvesterV2	0x276B8ba2e38a5e1CB43FF3493e620CFF6cEC4878
FeeHarvester (at launch)	0x8416DA7325cc619A5E560D8Ef6f413e94950CA5
PrivateRouter	0x9EB58318791e61decBCf8A4d169a0FaD8b6fb987
HonkVerifier	0x3DAd6EB076169F409A7901dff7BD26be3E883B09
Poseidon2 hasher	0xb63c2AA1370B32CcfaBaf5d24F854D817b405833
Pons V2 factory	0x7eD598BcEf8bd9Edd8C97A195C6d13f40801EC7e
Pons V2 meme hook	0xE5e702641Ea86F4ae6cC3cDaeD2B886f976Be044
Pons V2 fee escrow	0xd3AFEB2a57f70eF218Aa82451c51B2fb0416Ac9e
Uniswap v4 PoolManager	0x8366a39CC670B4001A1121B8F6A443A643e40951
EntryPoint v0.8	0x4337084D9E255Ff0702461CF8895CE9E3b5Ff108
CreateX	0xba5Ed099633D3B313e4D5F7bdc1305d3c28ba5Ed

B Constants

Constant	Value	Where
TREE_DEPTH	24	tree, circuit, client
ROOT_HISTORY_SIZE	64	tree
SHARES_BITS	120	circuit
VIRTUAL_SHARES	10 ⁶	vault
maxOpCostWei	0.0005 ETH (immutable)	vault
NETWORK_FEE_INTERVAL	1 hour	vault
MAX_PRICE_STEP_BPS	1000 (10%)	vault
EMA weight	1/8	vault
TEAM_SHARE_BPS	5000 (50%)	harvester
MAX_IMPACT_BPS	50 (0.5%) per buy, one buy per L1 block	harvester
MAX_PREMIUM_BPS	1500 (15%) above the average	harvester
CALLER_TIP_BPS	50 (0.5%), a cap on the gas refund	harvester
GAS_OVERHEAD	25,000 gas	harvester
SUCCESSOR_DELAY	24 hours	harvester
targetDeposit	0.1 ETH (immutable)	harvester
Creator tax	0	Pons
Standard fee, Pons share	100 (1%), 30%	Pons
Note ciphertext	286 bytes	client

References

- [1] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox. *Zcash Protocol Specification*. Electric Coin Company.
- [2] A. Pertsev, R. Semenov, R. Storm. *Tornado Cash Privacy Solution*, 2019.
- [3] Railgun. *Privacy System*. <https://docs.railgun.org/wiki/learn/privacy-system>.
- [4] L. Grassi, D. Khovratovich, M. Schofnegger. *Poseidon2: A Faster Version of the Poseidon Hash Function*. AFRICACRYPT 2023.
- [5] Noir. *The Noir Language*. <https://noir-lang.org>.
- [6] Aztec. *Barretenberg and UltraHonk*. <https://github.com/AztecProtocol/aztec-packages>.
- [7] J. Groth. *On the Size of Pairing-Based Non-interactive Arguments*. EUROCRYPT 2016.
- [8] H. Adams et al. *Uniswap v4 Core*. Uniswap Labs, 2024.
- [9] Pons. *Pons V2 launchpad*. <https://docs.ponsfamily.com/v2>.
- [10] V. Buterin et al. *ERC-4337: Account Abstraction Using Alt Mempool*.
- [11] *ERC-7562: Account Abstraction Validation Scope Rules*.
- [12] R. Bloemen, L. Logvinov, J. Evans. *EIP-712: Typed Structured Data Hashing and Signing*.
- [13] W. Chang et al. *EIP-4361: Sign-In with Ethereum*.
- [14] V. Buterin et al. *EIP-7702: Set Code for EOAs*.
- [15] *ERC-4626: Tokenized Vaults*.
- [16] *ERC-7984: Confidential Fungible Token*. OpenZeppelin Confidential Contracts.