noircash

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

Technical whitepaper — Draft v0.2 — September 2026

noircash contributors

Abstract

NOIR 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. noircash is its private side. Shielding moves NOIR into an immutable 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 NOIR's creator fee recipient, turns the creator share of every trade fee into NOIR for the vault: the fee stream 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 network fees in shares from the notes they spend, so no private transaction is signed or paid for by the user's address. Our contracts are immutable and have no administrator. This paper describes the construction, its security and privacy properties, and its limits.

Contents

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.

noircash 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.** NOIR 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 noircash router lands straight in a note.
3. **Privacy is where holding pays.** The creator share of every trade fee is harvested into the vault that backs the notes. Public NOIR 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 NOIR 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 are immutable and have no administrator. 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 NOIR works everywhere but earns nothing; only notes receive the fee stream.
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, no team revenue from trading or 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 NOIR for the notes, a registry of privacy keys, an append-only Merkle tree of note commitments, the set of spent nullifiers, the vault totals, a gas reserve and a price average. It verifies every spend, and it is itself an ERC-4337 [10] account. Two small immutable contracts of ours surround it, next to the Pons launchpad [9] and Uniswap v4 [8] (Fig. 1 and Table 1).

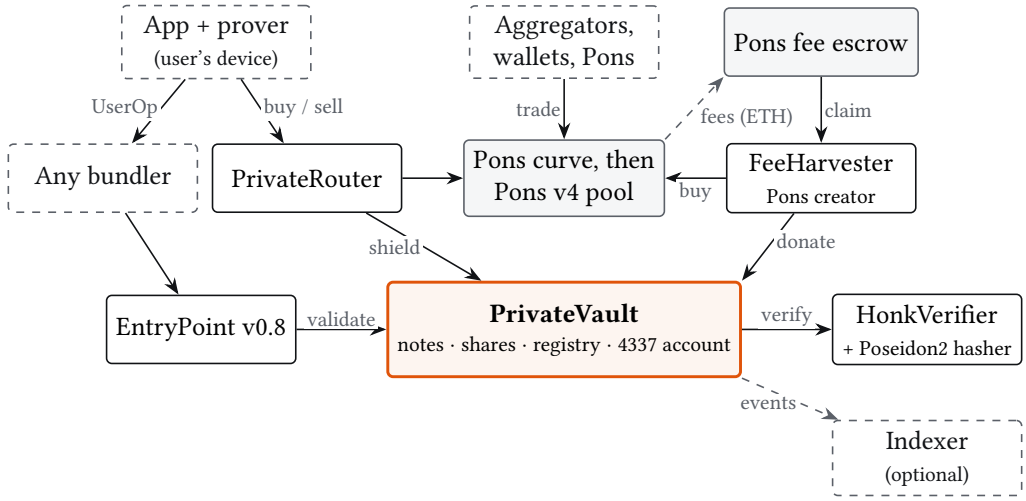

Figure 1: Our contracts (solid), Pons (grey) and off-chain parts anyone can run (dashed).

Component	Role
NOIR	Plain ERC-20 created by the Pons factory: OpenZeppelin `ERC20` and `ERC20Burnable`, no owner, 10^9 supply.
Pons curve and pool	The bonding curve NOIR trades on until graduation, then a Uniswap v4 pool with the Pons hook and locked liquidity. Pons charges its fees on both.
`PrivateVault`	Holds NOIR for the notes. Key registry, note tree and nullifiers, vault totals B and S, gas reserve, price average. Verifies every spend. Acts as the ERC-4337 account for private transactions.
`FeeHarvester`	NOIR's Pons creator fee recipient. Turns the creator fees into gas and into NOIR donated to the vault. Callable by anyone.
`PrivateRouter`	`buy`: buys NOIR and shields it in one transaction. `sell`: spends notes and sells for ETH. Settles ERC-4337 sales. `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 owner, proxy, pause, 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).

4 Cryptographic construction

4.1 Primitives and notation

Let p be the BN254 scalar field modulus,

$$p = \texttt{0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001},$$

and let H denote the Poseidon2 [4] sponge over $\mathbb{F}_p$ as computed by Noir's `Poseidon2::hash`. The same definitions are implemented three times and must agree: in the Noir circuit, in the Solidity library `NoteHash`, and in the client. 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:

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

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. Every note created by `shield` or `transact` carries a 286-byte ciphertext for its owner:

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

with a fresh ephemeral key pair (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 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

noircash 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:

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

where $x_{\mathrm{hi}}, x_{\mathrm{lo}}$ 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:

$$
\mathsf{Spend}(\texttt{bytes32 } transaction), \qquad transaction = \mathsf{H}(root, \mathsf{nf}_0, \mathsf{nf}_1, \mathsf{cm}_0, \mathsf{cm}_1, exit, extDataHash),
$$

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; $extDataHash = \mathsf{keccak256}(\mathrm{abi.encode}(ext)) \bmod p$; 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 $(\mathsf{nk}_j, \rho_j, r_j, s_j, i_j, \pi_j)$ and the shared wallet key (x, y):

$$
\begin{aligned}
&s_j < 2^{120}, \quad i_j < 2^{24},\\
&\mathsf{cm}_j = \mathsf{H}\big(1, \mathsf{H}(\mathsf{H}(\mathsf{nk}_j, x_{\mathrm{hi}}, x_{\mathrm{lo}}, y_{\mathrm{hi}}, y_{\mathrm{lo}}), \rho_j, r_j), s_j\big),\\
&s_j \neq 0 \Rightarrow \mathsf{MerkleRoot}(\mathsf{cm}_j, i_j, \pi_j) = root,\\
&\mathsf{nf}_j = \mathsf{H}(2, \mathsf{nk}_j, \mathsf{cm}_j, i_j).
\end{aligned}
$$

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

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

The range checks prevent the sum from wrapping the field. Finally the circuit recomputes the EIP-712 digest $\mathsf{keccak256}(\texttt{0x1901} \,\|\, \mathit{domain} \,\|\, \mathsf{keccak256}(\mathsf{SPEND\_TYPEHASH} \,\|\, \mathit{transaction}))$ 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 gas fee in shares, caller-specific parameters (sale terms and a minimum call gas limit) and the two output ciphertexts. Because its hash is a public input covered by the wallet's signature, no relayer or bundler can redirect an exit, 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 gas 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 with pinned hashes. 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 NOIR 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:

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

Shielding a tokens adds a to B and $\mathsf{toShares}(a)$ to S. Exiting e shares removes $\mathsf{toAssets}(e)$ from B and e from S.

Donations. `donate(amount)` adds NOIR to B and leaves S unchanged, so the value of every share rises. The harvester is the protocol's donor: it donates the NOIR it buys with the Pons creator fees, and the gas reserve once the gas deposit is full (Sections 7 and 8). Because donations raise the share price uniformly, every private balance grows in proportion to its shares without any transaction and without revealing amounts. Public NOIR holds tokens, not shares, and receives nothing.

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

$$\mathsf{noir.balanceOf}(\mathsf{vault}) = B + \mathsf{gasReserve} + \sum \mathsf{pendingSales},$$

except for NOIR 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 NOIR and shields it into a note whose stub is $\mathsf{keccak256}(\texttt{"noircash/locked-seed"}) \bmod p$. 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 NOIR and shielding

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

Public NOIR 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 NOIR. 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`, NOIR 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/NOIR 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; NOIR 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:

Share of each trade	With the defaults
Paid by the trader	1.0%
Kept by Pons	0.3%
To the harvester, for private holders	0.7%

`FeeHarvester.harvest`, callable by anyone for 0.5% of the ETH it handles, sweeps the fees Pons holds for NOIR, claims them from the escrow, keeps the vault's gas deposit at its target (Section 8), and spends the rest on NOIR, which it donates to the vault. The buy must land within 3% of the vault's price average after fees; when the market cannot take the whole amount within that bound, the harvester halves it, trying up to eight sizes, and the rest waits for the next harvest.

7.3 Price average

The vault keeps an exponential moving average of the price in NOIR per ETH, used to value gas fees (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, $lastSpot$, $lastBlock$

if block.number $\neq$ $lastBlock$ **then**
  $c \leftarrow \mathrm{clamp}(lastSpot,\ 0.9\,ema,\ 1.1\,ema)$; ▷ `previous update`
  $ema \leftarrow (7\,ema + c)/8$;
$lastSpot \leftarrow x$; $lastBlock \leftarrow$ 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. Moving the average by the 20% gas margin (Section 8) then takes about 15 consecutive L1 blocks, each paying the Pons fees on both legs of a push large enough to move the price by 10%, while the most it can win is that margin on the ERC-4337 operations of the window. We accept this bound.

7.4 Private router

`PrivateRouter` holds no funds between transactions. `buy` buys NOIR 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. `sell` spends notes with a proof and sells the exit for ETH. `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. Minimum outputs default 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 fee in shares out of the notes being spent.

Validation performs the transition. `validateUserOp` decodes the transaction and proof, checks the fixed operation fields, enforces a minimum call gas limit bound in the proof, checks the fee, and then applies the entire private state transition: root, nullifiers, proof, insertion of both outputs, fee to the gas reserve, and the exit credited or held in escrow. Execution only settles sales. 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.

Fee check. With $\mathsf{maxCost} = (verificationGas + callGas + preVerificationGas) \cdot maxFeePerGas$ and $feeWei = \mathsf{toAssets}(gasFee) \cdot 10^{18}/ema$, validation requires

$$feeWei \cdot 10^4 \;\geq\; \mathsf{maxCost} \cdot (10^4 + 2000),$$

a 20% margin over the maximum declared cost, priced at the moving average. ERC-7562 [11] forbids reading the market during validation, but the vault's own storage is allowed.

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.

Sales. A swap cannot run in validation. The sale amount is held in escrow, keyed by the first nullifier; execution approves it to the router, which pulls and sells it, sending ETH to the recipient bound in the proof. If the sale fails, the NOIR goes to the recipient instead, and if execution runs out of gas anyone may release them the same way (`claimSale`).

Gas deposit. Gas fees accumulate in NOIR in the gas reserve. The harvester keeps the EntryPoint deposit at a fixed target (0.1 ETH by default) with the ETH it harvests first, since the Pons creator fees are ETH; only if that is not enough does it sell the gas reserve, within 3% of the average after fees. Once the deposit is full it donates the reserve to the vault. Since fees cover the maximum declared gas, the surplus returns to private holders. Nobody can withdraw the deposit or the EntryPoint stake.

Guaranteed exits. If no bundler serves the vault or the deposit is empty, 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. If the router is unusable, notes can be unshielded and the NOIR sold on any venue.

9 NOIR on Pons

Parameter	Value
Token	Plain ERC-20 created by Pons V2, no owner
Creator fee recipient	`FeeHarvester`
Total supply	10^9 NOIR (18 decimals), fixed, minted into the Pons curve
Team allocation, treasury, reserve	None
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: NOIR's terms on Pons.

Every noircash 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. Pons' record of the token, `getLaunchedToken(noir)`, names the harvester as creator fee recipient, with ETH as the pair, no creator tax and buybacks off; anyone can read it. The harvester takes NOIR's address in its constructor and reverts unless that record exists, names the harvester and pairs the token with ETH, so it cannot be deployed for a token whose fees it does not receive. The wallet that made the launch has no special rights over NOIR: only Pons' owner can redirect the creator fees, after a 3-day timelock. The deployer buys the locked seed once the snipe tax is over. The team can hold NOIR only by buying it.

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.

Party	Can	Cannot
Deployer	Nothing after deployment	Change fees, move notes, pause, upgrade
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 NOIR
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; valid operations pay at least 120% of their maximum gas cost.

Price manipulation. The average is clamped and weighted and moves at most 1.25% per L1 block. Moving it by the 20% gas margin takes about 15 consecutive manipulated blocks, each paying Pons' fees on both legs of the push. The harvester's trades must land within 3% of the average.

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. The fee stream depends on Pons: its owner can redirect it after a timelock, and Pons keeps 30% of the standard fee. The gas fee covers the maximum declared gas, so users pay more than the gas used (the surplus goes to holders). 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 there protection comes from the number of users.

Information	Public
Total supply, B, S, gas reserve, price average	yes
Trades on the curve and the pool: amounts, price, time	yes
Public NOIR 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; 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 fees accrue only to notes, the design aims to make the private set the natural place to hold NOIR, 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 contacts no 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; noircash 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; noircash pays its private holders the token's own trading 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 trading-fee stream that accrues only to private holders, and self-paid ERC-4337 gas without any administrator.

15 Conclusion

noircash keeps NOIR an ordinary token that trades anywhere, and makes holding it in private the way to earn from its trading. Shielded NOIR becomes encrypted notes; 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 NOIR in private.

A Deployment

Robinhood Chain, chain ID 4663. Every noircash contract is deployed through CreateX's CREATE3 with a salt guarded by the deployer's address and the chain ID, so the addresses below are fixed before deployment and nobody else can deploy at them. NOIR's and its curve's addresses are set by Pons when it creates the token.

Contract	Address
NOIR (Pons token)	set at launch
Pons curve	set at launch
PrivateVault	`0x0a55A88524002dd8F053Ffc644015EDc9653963a`
FeeHarvester	`0x76d4979272019475b7eD690d5943E14296bDcDd6`
PrivateRouter	`0x00fe9533Df78aF8e59cF8BC0EcDDe551140920D5`
HonkVerifier	`0xeA05c34fF6a60EBF6E73544E449B0DE7095C7941`
Poseidon2 hasher	`0x33E32aa162d637F149924ab025DAcA55E1571842`
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
`GAS_MARGIN_BPS`	2000 (20%)	vault
`MAX_PRICE_STEP_BPS`	1000 (10%)	vault
EMA weight	1/8	vault
`MAX_SLIPPAGE_BPS`	300 (3%)	harvester
`CALLER_TIP_BPS`	50 (0.5%)	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.