voidcash: A Zero-Knowledge Payment Pool

The voidcash team
voidcash.io
November 2026

Abstract. A public ledger lets anyone verify a payment without trusting an intermediary, and it shows everyone who paid whom, how much, and what each wallet holds. We describe a pool on Elysium that keeps the public verification and removes that exposure for HYPE. A deposit becomes a commitment in an append-only Merkle tree. Spending a note takes a zero-knowledge proof of membership and conservation of value, together with a signature from the owner's wallet that is checked inside the proof. The chain verifies the proof and learns neither the sender, the recipient nor the amount. Deposits and withdrawals stay public, so privacy depends on the rate of deposits and on how long funds remain in the pool, and we quantify that dependence. The pool charges a single fee of 1% on withdrawals, half of which buys and burns the VOID token.

1. Introduction

Public blockchains settle payments without a trusted party by publishing every payment [1]. The price of that design is that a wallet address works as an open bank statement. Anyone can read its balance, its counterparties and its full history, and a single payment is enough to tie a person to all of it. Salaries, supplier invoices and treasury movements are visible to competitors and strangers by default, which rules out on-chain payments for most serious businesses.

Zerocash [2] showed that a ledger can verify spending without seeing what is spent, using commitments, nullifiers and zero-knowledge proofs. voidcash applies that construction to native HYPE on Elysium [3], an Arbitrum Orbit chain that settles to HyperEVM. Three decisions shape the system. Notes hold arbitrary amounts, so users are not forced into fixed denominations. Every spend needs a signature from the user's existing wallet, so there is no new seed phrase to protect, and leaked privacy keys can expose a history without moving any funds. Gas is paid from inside the pool through ERC-4337 [4], so a private payment never touches the sender's public address.

This paper specifies the construction as implemented, including its limits. Section 12 measures how much an observer can still infer, and Section 13 lists what a user has to trust.

2. Notes

A note is a claim on HYPE held by the vault. We write H for the Poseidon2 hash [5] over the BN254 scalar field $\mathbb{F}_p$, where $p \approx 2^{253.6}$. A note is a tuple (opk, ρ, r, v) in which opk is the owner's public key, ρ and r are uniformly random field elements, and $v < 2^{120}$ is the amount in wei. It appears on the ledger only as a commitment:

$$stub = H(opk, \rho, r), \qquad cm = H(1, stub, v) \tag{1}$$

Splitting the commitment in two lets the owner fix the secrets while the contract fixes the amount. On a deposit the user submits *stub* and the vault computes *cm* itself from the value received, so nobody can commit to more than they paid. With ρ and r random and H modeled as a random oracle, *cm* reveals nothing about the owner or the amount.

Spending the note at leaf index i publishes its nullifier

$$nf = H(2, nk, cm, i) \tag{2}$$

where nk is the owner's nullifier key. The vault records every nullifier and rejects any it has already seen. Without nk nobody can connect nf to cm, so the chain learns that a note was spent and not which one. Binding the index keeps nullifiers distinct for identical notes in one tree.

3. Keys

Users should not have to protect another secret. A user's privacy keys come from one wallet signature σ_M over a fixed message M [6] that names the domain, the chain and the vault. With HKDF over SHA-256 [7] and labels namespaced under `voidcash/v1/`:

$$\begin{aligned} s &= \text{keccak256}(\sigma_M) \\ sk &= \text{HKDF}(s, \text{“sk”}) \bmod p, \quad nk = H(sk) \\ vsk &= \text{HKDF}(s, \text{“view”}), \qquad vpk = \text{X25519}(vsk, 9) \\ opk &= H(nk, x_{\text{hi}}, x_{\text{lo}}, y_{\text{hi}}, y_{\text{lo}}) \end{aligned} \tag{3}$$

where (x, y) is the wallet's secp256k1 public key split into 128-bit limbs. The same wallet gives the same keys on any device, so there is nothing new to back up. This relies on deterministic signing [8], so the app signs twice and refuses a wallet whose signatures differ. Smart-contract wallets cannot use the system, because the circuit verifies only secp256k1 signatures.

Publishing (opk, vpk) in the vault's registry lets others pay the wallet's address. The design separates three powers. The key vsk reads the notes sent to a user. The key nk also reveals when they are spent. Neither can move anything. Moving a note takes a fresh signature from the wallet, checked inside the proof, because opk commits to the wallet's public key. A user tricked into signing M on a malicious site loses their privacy and keeps their funds.

4. Transactions

A transaction spends up to two notes, creates two notes, and can move value out of the pool. Its public statement $\mathbf{x}$ and private witness $\mathbf{w}$ are:

$$\begin{aligned} \mathbf{x} &= (rt, nf_0, nf_1, cm'_0, cm'_1, e, h, d_{\text{hi}}, d_{\text{lo}}) \\ \mathbf{w} &= \big(\{nk_a, \rho_a, r_a, v_a, i_a, \pi_a\}_{a=0,1},\ \{stub'_b, v'_b\}_{b=0,1},\ Q,\ \sigma\big) \end{aligned} \tag{4}$$

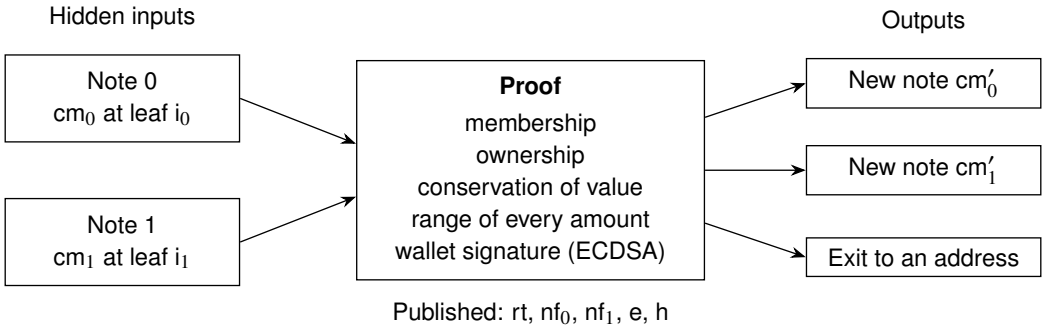
Here rt is a tree root, e is the value leaving the notes, π_a is a Merkle path and Q is the wallet's public key. The value $h = \text{keccak256}(ext) \bmod p$ commits to data the circuit does not inspect, namely the recipient, the permitted caller, the gas fee, the minimum call gas and the two note ciphertexts. The proof shows knowledge of $\mathbf{w}$ such that, for $a, b \in \{0, 1\}$:

$$\begin{aligned} & v_a,\ v'_b,\ e < 2^{120}, \quad i_a < 2^{24} && \textit{range} \\ & cm_a = H\big(1,\ H(H(nk_a, Q), \rho_a, r_a),\ v_a\big) && \textit{ownership} \\ & v_a \neq 0 \implies \text{Root}(cm_a, i_a, \pi_a) = rt && \textit{membership} \\ & nf_a = H(2, nk_a, cm_a, i_a) && \textit{nullifier} \\ & cm'_b = H(1, stub'_b, v'_b) && \textit{outputs} \\ & v_0 + v_1 = v'_0 + v'_1 + e && \textit{conservation} \\ & \text{ECDSA.Verify}(Q, D, \sigma) = 1 && \textit{authorization} \end{aligned}$$

The digest D follows EIP-712 [9] and covers every public input:

$$D = \text{keccak256}\big(\texttt{0x1901} \,\|\, d \,\|\, \text{keccak256}(TH \,\|\, H(rt, nf_0, nf_1, cm'_0, cm'_1, e, h))\big) \tag{5}$$

where TH is the type hash of `Spend(bytes32 transaction)` and $d = d_{\text{hi}} \| d_{\text{lo}}$ is the vault's domain separator, which binds the chain and the contract. The range checks make the conservation equation hold over the integers, because no sum can reach 2^{122} and $2^{122} \ll p$. A note with $v_a = 0$ is a dummy. It skips membership and creates no value, and it still yields a nullifier, so a transaction with one real input looks like a transaction with two. The witness holds a single Q, so both inputs must belong to one wallet.



The vault checks that rt is a known root, that $nf_0 \neq nf_1$, that neither has been spent, and that nullifiers and commitments are canonical field elements below p, so no alias $nf + kp$ can replay a spend. It computes h and d itself, so nobody can change the recipient or the ciphertexts after the user has signed. It then records the nullifiers and appends both commitments.

The circuit is written in Noir [10] and compiles to 83,781 gates, which fit a domain of $n = 2^{17}$. We prove it with UltraHonk in Barretenberg, a Plonk-style [11] system built on sumcheck [12] with KZG polynomial commitments [13]. There is no circuit-specific trusted setup. The system reuses the universal reference string from the Aztec Ignition ceremony, and the app serves those files from its own origin so that no third-party CDN sees who is preparing a transaction. Proofs are generated in the user's browser, and notes and keys never leave the device. A proof is 9,152 bytes. A transaction that verifies one and updates state costs about 3.5 million gas. Verifier challenges are 127-bit values and each sumcheck round polynomial has degree at most 8, so sumcheck contributes at most $8 \log_2 n / 2^{127} < 2^{-119}$ per attempt. An attacker who tries q transcripts multiplies that by q, and it is only one term of the soundness error.

5. The Note Tree

Commitments live in an append-only Merkle tree of depth $D_T = 24$. Empty leaves are 0, and the root of an empty subtree follows $z_0 = 0$, $z_{k+1} = H(z_k, z_k)$. A path from leaf cm at index i, with i_k the k-th bit of i, is checked by:

$$a_0 = cm, \qquad a_{k+1} = \begin{cases} H(a_k, \pi_k) & \text{if } i_k = 0 \\ H(\pi_k, a_k) & \text{if } i_k = 1 \end{cases}, \qquad \text{Root} = a_{24} \tag{6}$$

Every insertion is a pair at an even index. A deposit pairs its note with an empty leaf, which leaves the root exactly as if the note had been inserted alone. Inserting a pair costs $1 + (D_T - 1) = 24$ hashes instead of $2D_T = 48$. A tree holds 2^{24} leaves, or 2^{23} insertions. When one fills, the vault keeps its final root permanently and starts a new tree, so old notes stay spendable. One proof has one root, so both inputs of a transaction must come from the same tree.

Other transactions land while a user is proving, so the vault accepts any of its last 64 roots. A proof stays valid through the next 63 insertions. If it expires, the user proves again against a newer root. A griefer could force that with 64 small deposits, at the cost of their gas.

6. Finding Notes

A note is useless until its owner can find it, so each output carries a ciphertext for the recipient's viewing key. The sender draws an ephemeral X25519 [14] key pair (esk, epk) and a 24-byte nonce η, then computes:

$$\begin{aligned} S &= \mathrm{X25519}(esk, vpk), \qquad K = \mathrm{HKDF}(S, epk, \text{“note-key”}) \\ t &= \mathrm{SHA256}(S \,\|\, \text{“view-tag”})[0] \\ c &= epk \,\|\, t \,\|\, \eta \,\|\, \mathrm{XChaCha20Poly1305}_K(\eta,\ \rho \,\|\, r \,\|\, v \,\|\, memo) \end{aligned} \tag{7}$$

Our client always writes ciphertexts [15] of exactly 286 bytes, including a fixed 117-byte memo. An empty output carries random bytes of the same length. Those differ from a real ciphertext, whose first 32 bytes are a valid curve point, but the client produces them only on exits, which are public anyway. The one-byte tag t lets a wallet discard 255 of every 256 foreign notes before attempting decryption, so scanning N leaves costs N key agreements and about $N/256$ decryptions. A wallet accepts a note only if it can recompute cm from the decrypted values. The memo in a sender's change note records what the spend did, which is how a sender's history and receipts survive on any device. A recipient's note carries an empty memo of the same length.

Wallets read events through an optional indexer that serves the same stream to everyone, so it cannot learn which notes belong to whom. The wallet rebuilds the tree and checks its root against the vault, which catches a changed commitment. It does not catch everything. The vault accepts any of its last 64 roots, and ciphertexts and nullifiers are not in the tree, so a dishonest indexer can hide notes or misreport a balance. It cannot move funds, and a wallet that doubts it can read the chain directly.

7. Private Gas

If a private payment were sent from the user's wallet, the transaction's sender field would name them. So the vault itself acts as an ERC-4337 account. A UserOperation names the vault as its sender and carries the transaction and proof in its signature field, and the private state change happens during validation. A bundler submits the operation, and the EntryPoint reimburses it from a gas fee g that comes out of the notes and is bound to the proof through h. The value leaving the notes splits as $e = g + u$, where u is the amount exiting the pool and is zero for a private send. The vault requires

$$g \geq (V + C + P_V) \cdot m \tag{8}$$

where V, C and P_V are the verification gas limit, the call gas limit and the pre-verification gas, and m is the maximum fee per gas. It deposits g in the same validation, so no operation is ever paid for by anyone else. Unused gas is not refunded. It is swept to the fee collector of Section 10. An operation is about 6 million gas at its limits and uses about 5 million, and Elysium has no parent-chain data fee. At the prices of 3 November 2026 an operation pays for its limits at 0.12 gwei and is charged 0.1006 gwei for what it uses, so g is about 7×10^{-4} HYPE and roughly 30% of it goes to the collector.

The proof leaves gas limits and prices free, so bundlers can estimate and price an operation, and it fixes a minimum call gas in *ext* so that no bundler can starve execution. Each operation's nonce key comes from nf_0, so users never queue behind each other. An invalid proof returns the standard signature-failure code instead of reverting, which lets a wallet estimate gas with a placeholder and prove only once.

Elysium has no public bundler today, so at launch we run the only one, behind a gate that admits only the vault's operations. It can delay a user and it sees the network address behind each operation. It can keep the whole of g and nothing beyond it, and it cannot read notes. Anyone can run another, and any account can call `transact` directly and pay the gas itself, at the cost of making that account visible.

8. Exits

Moving value to a public address is an exit. With $\phi = 1\%$ the vault takes a fee and pays out the rest:

$$f = \lfloor \phi u \rfloor, \qquad u' = u - f \tag{9}$$

The recipient, u', f and g are public. If $u' > \Theta$ the exit is held for 24 hours, during which the vault's guardian can stop it. Θ defaults to 1,000 HYPE and the admin can set it between 1 and 1,000,000 HYPE. The hold gives the guardian time to act on a large exit.

The vault keeps three totals, and its balance always covers them:

$$\text{balance} \geq B_{\text{notes}} + B_{\text{exits}} + F \tag{10}$$

B_{notes} is the sum of unspent notes, B_{exits} is exits decided and not yet paid, and F is fees not yet swept. A deposit of v adds v to B_{notes}. A transaction subtracts e from it, adds u' to B_{exits} and f to F, and sends g to the EntryPoint.

9. Receipts and Links

Privacy should not stop a user from proving a payment to an accountant or a counterparty. The memo in the sender's change note holds the opening of the note they created, and a receipt is the tuple (tx, i, to, ρ, r, v). Anyone can check that transaction tx created commitment cm at index i, read opk_{to} from the registry at that block, and verify:

$$cm = H\big(1, H(opk_{to}, \rho, r), v\big) \tag{11}$$

That shows the transaction created a note of value v for the owner key registered at that address. It says nothing about the sender. It also does not show that the recipient can find the note, because nothing on-chain checks a ciphertext against its commitment. A nullifier needs the recipient's nk, so the verifier cannot tell whether or when the note is later spent.

A registered address is one way to be paid. A private address is a second identity derived from sk, with its own nk and vsk and the same wallet key. Its public keys travel in a link, and nothing on-chain connects them to the wallet. A payment link is a bearer note. Its key is a fresh secp256k1 private key derived with a second wallet signature, the note is paid to the keys it derives, and whoever holds the link can claim it. The sender can take it back until then, and so can anyone who obtains both of the sender's signatures. Both kinds of link keep their contents in the URL fragment, which browsers do not send to servers.

10. Fees and the Burn

VOID is an ERC-20 token launched on Volta [16], a launchpad on Elysium. It trades on a Uniswap v4 pool against native HYPE from its first block, with an LP fee of 0, a tick spacing of 60 and a single position that runs from a market cap of 40 HYPE to the top of Uniswap's price range. Nobody needs it to use voidcash, and it gives no claim on the vault. It launched with a supply of 10^9, all of it in the pool, with no premine. VOID is burned in two ways. The fee collector sends what it buys to the address `0x...dEaD`, where nobody can spend it. Volta's buyback destroys what it buys, which lowers the total supply. The burned total is therefore $(10^9 - \text{totalSupply}) + \text{balanceOf}(\texttt{0x...dEaD})$. The vault sweeps exit fees and unused gas to a fee collector, which splits whatever arrives in half. One half is owed to the team. The other half buys VOID and burns it, and once split, nobody can take it. An exit of u HYPE therefore allocates $\phi u/2$ to the burn, before the caller's refund and the swap fee.

Every swap on the pool pays Volta's swap fee $\tau = 1\%$, fixed for the life of the pool: in HYPE on a sale, and in VOID on a buy. Volta sells the VOID it takes to later buyers, before their swaps reach the pool, so every fee ends up in HYPE. Volta keeps 30% of each fee, and VOID's launch splits the rest in half between the holder of its creator NFT, which is the team, and a buyback that also burns VOID. A trade of size W

therefore pays 0.3%W to Volta, 0.35%W to the team and 0.35%W to the buyback. The buyback's share waits in Volta's FeeVault until Volta's executor spends it, and only Volta's operators can run the executor. Volta's documentation says it runs once a pool has collected 0.01 HYPE, and at most every 10 minutes.

The collector has to buy without feeding arbitrageurs. Within active liquidity L the pool is a constant-product market with virtual reserves $X = L/\sqrt{P}$ of HYPE and $Y = L\sqrt{P}$ of VOID, where P is VOID per HYPE [17]. Adding Δ HYPE multiplies the marginal price of VOID by:

$$\frac{\text{price}'}{\text{price}} = \left(1 + \frac{\Delta}{X}\right)^2 \tag{12}$$

Volta takes the fee on a buy out of the VOID, so all of Δ reaches the pool, unless Volta fills part of the buy at the spot price from the VOID it holds, which moves the price less. The collector caps each buy at $\Delta_{\max} = X\delta/2$ with $\delta = 0.5\%$, so the collector's own swap moves the price by at most $\delta + \delta^2/4 \approx 0.5006\%$. It buys at most once every 12 seconds and requires at least $\Delta P(1-\tau)(1-\delta/2)$ VOID out, halving Δ when the market cannot fill that. That check uses the spot price of the same moment, so it limits slippage within a buy and does nothing against a trade placed before it. What does not fit waits for the next call, and before anyone has bought, the pool has no active liquidity and the collector does not buy. Anyone can trigger a buy, and the caller's gas refund is capped near 0.5% of what the call spends.

Suppose an attacker buys with a HYPE just before the collector and sells right after. Let $\gamma = 1 + \delta/2$ and $\alpha = a/X$. All of a enters the pool, and the attacker receives the VOID out less the fee. The cap is computed after the front-run, so the collector's buy multiplies the HYPE reserve by at most γ. Solving the constant-product invariant through the three swaps, the attacker's sale returns, per HYPE swapped in, $(1-\tau)^2 R(\alpha)$ with:

$$R(\alpha) = \gamma^2 \frac{1+\alpha}{1+(1-\tau)\gamma\alpha} \tag{13}$$

Since $(1-\tau)\gamma < 1$, this rises strictly from γ^2 as $\alpha \to 0$ to $\gamma/(1-\tau)$ as $\alpha \to \infty$. With the fee paid on both legs, the attacker's profit is bounded by:

$$\Pi(a) = a\big[(1-\tau)^2 R(\alpha) - 1\big] \;\le\; a\big[(1-\tau)\gamma - 1\big] \tag{14}$$

At $\tau = 1\%$ and $\delta = 0.5\%$ the bound is a loss of 0.75% of the attacker's capital, at any size, and of 1.50% for a buy that is small next to X. A single buy breaks even only at $\delta^* = 2\tau/(1-\tau) \approx 2.02\%$, four times the cap. If Volta fills part of the attacker's buy from the VOID it holds, the attacker pays the spot price for that part and pushes the pool less, and the bound still holds. The cap limits the pace of buying and leaves the total open, though. The backlog is public and anyone can trigger each buy, so an attacker who buys first and holds through n buys gets the same return with γ^n in place of γ. That turns positive from $n = 5$, about 48 seconds from the first buy to the fifth, once the backlog exceeds about 1% of X. Volta's buyback is exposed the same way whenever a single buyback spends more than about 1% of X. Its pending amount is public, and its size and timing are up to Volta's operators. What remains against this attacker is the market risk of holding through the window. These results assume third parties cannot change the pool's liquidity. They describe mechanics, and we make no claim about what they do to VOID's price.

11. Security

We assume Poseidon2 is collision-resistant and pseudorandom, UltraHonk is knowledge-sound and zero-knowledge, ECDSA is unforgeable, and at least one participant in the Ignition ceremony was honest. The following arguments are sketches.

No inflation. A proof that verifies has a witness. The range and conservation constraints give equality over the integers. Membership holds for every input with $v_a \neq 0$, and each note has one nullifier, so a spend removes real value exactly once. The vault fixes the value of every deposit itself, so unspent notes never sum to more than B_{notes}.

No double spend. The range check on i_a gives each leaf one index. Two spends of the same leaf with different nullifiers need $nk \neq nk'$ under the same cm, which the ownership constraint turns into a collision in H. After q hash evaluations the probability of a collision is at most $q^2/2p$, which is below 2^{-94} even at $q = 2^{80}$.

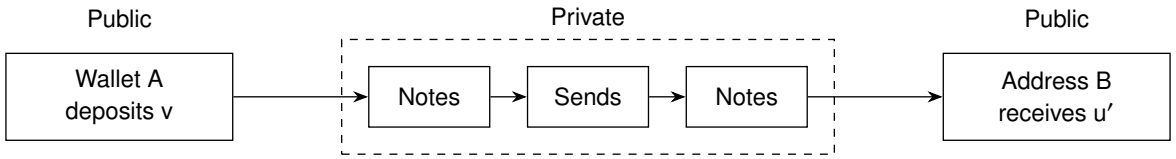
No theft. The authorization constraint needs a signature under the key committed in opk, so a forged spend breaks ECDSA or finds a collision. The signature covers every public input, so it cannot be reused for another recipient, and the domain separator stops replay elsewhere.

Privacy of a send. The chain sees $(rt, nf_0, nf_1, cm'_0, cm'_1, g, c_0, c_1)$, a proof, the gas limits and prices, an exit event of amount zero, and the submitter's address. Zero knowledge hides the witness, the nullifier is a pseudorandom function of nk, and commitments and ciphertexts hide their contents. None of this depends on the recipient or the amount. The sender stays hidden provided they use a shared bundler and default gas settings.

BN254 is the weakest assumption. Current estimates put it near 100 bits [18], below the roughly 127 bits of collision resistance in Poseidon2 and the 128 bits of secp256k1 and X25519. A break there would let an attacker forge proofs. The verifier is part of the vault's implementation, so an upgrade can replace it. We pin Barretenberg 5.2.1, which has the small-subgroup IPA soundness fix, and our build and deploy script both fail on a verifier without it.

12. What an Observer Can Still Infer

The pool hides what happens inside it, and the edges stay public. A deposit shows who deposited and how much. An exit shows who received and how much. The natural attack is to match them.



Consider an exit. Suppose deposits arrive as a Poisson process of rate λ, and each could have funded it with independent probability β. If a user withdraws exactly what they deposited, only deposits of nearly that size qualify and β is tiny. If they withdraw part of a larger balance, every larger deposit qualifies. An observer who knows only that the funds stayed at most T, and treats every plausible deposit in that window as equally likely, sees Z decoys, where Z is Poisson with mean $\mu = \lambda\beta T$. No decoy appears with probability $e^{-\mu}$, and a correct pick has probability:

$$P_{\text{link}} = \sum_{k=0}^{\infty} \frac{\mu^k e^{-\mu}}{k!} \cdot \frac{1}{k+1} = \frac{1 - e^{-\mu}}{\mu} \tag{15}$$

So P_{link} falls as $1/\mu$. Reaching 5% takes $\mu = 20$, and reaching 1% takes $\mu = 100$. In C:

```
double LinkProbability(double lambda, double beta, double t)
{
    double mu = lambda * beta * t;
    if (mu == 0.0)
        return 1.0;
    return (1.0 - exp(-mu)) / mu;
}
```

Results for an illustrative λ of 10 deposits per hour:

Window T	*Same amount, $\beta = 0.01$*	*Varied amount, $\beta = 0.30$*
10 minutes	P = 0.9917	P = 0.7869
1 hour	P = 0.9516	P = 0.3167
6 hours	P = 0.7520	P = 0.0556
1 day	P = 0.3789	P = 0.0139
1 week	P = 0.0595	P = 0.0020

Time and amount do different work. A one-day window at the same amount still leaves the observer a 38% chance, and at a varied amount it leaves 1.4%. A real observer may also have timing habits, repeated counterparties or network metadata, and within this model any such information raises P_{link}, so read the table as a floor. The key registry is itself a public list of addresses that use the pool. A new pool has a small λ, which means its earliest users get the least cover and should wait the longest. Value paid inside the pool never reaches an edge at all, and it gives this attack nothing to match.

13. What You Have to Trust

The vault sits behind an upgradeable ERC-1967 proxy with three roles. The upgrader can replace the implementation, including the verifier. That is how we would respond to a flaw in the proof system, and it also means the upgrader could change the rules, so users trust it with the funds in the pool. The guardian can pause every spend and exit payout with no expiry, stop new deposits, and stop or resume an exit not yet paid, and it cannot move HYPE. The admin grants roles, including to itself, and sets Θ and the fee collector, which decides where future fees go. All of these actions are public on-chain transactions and take effect at once. At deployment a single key holds all three roles, and the same key launches VOID and holds its creator NFT, receives the team's half of the collector's HYPE and owns the collector.

Elysium adds its own assumptions. Its documentation describes a single sequencer at launch, permissioned validation, and an AnyTrust data committee that falls back to posting full data on HyperEVM. A chain owner can change system parameters, settlement on HyperEVM trails by hours, and the correctness of the chain's state rests on its validators. A sequencer can delay or reorder transactions. Force inclusion through the parent chain is the way around one that is down. Against selective censorship of a busy pool it may fall short, because a proof's root can expire during the delay. A sequencer cannot forge a proof or read a note.

VOID's market depends on Volta, which is built on Flaunch's contracts and does not publish its Solidity source. What we say about it comes from the ABI its site ships, its documentation, its bytecode and our fork tests, and the launch refuses to run if the tested code or the way its contracts point at each other has changed. Its contracts are not proxies, and all of them but its internal swap pool have the same owner, a single externally owned account. That owner can name a keeper of the FeeVault, and a keeper can withdraw VOID's pending buyback share to any address, so the share meant for burning is in Volta's hands until its executor spends it. The owner also names the operators who run the buyback and can repoint the contracts that new launches use. It cannot reach the team's share in Volta's fee escrow or anything held by voidcash's contracts. Two public functions of Volta's PositionManager could not be identified. The pool lives on a PoolManager whose code is Uniswap v4's, and its owner can name a protocol fee controller, which could add up to 0.1% to each swap. The vault does not depend on Volta at all. If Volta stopped working, deposits, sends and exits would continue, and the HYPE meant for burning would wait in the collector.

Users also trust their wallet and the page they sign on. A compromised wallet loses both funds and privacy. A wallet shows a spend signature as an opaque hash, so a user cannot see what it authorizes, and a malicious page can obtain one that moves their funds. A malicious page that obtains only σ_M can read a user's history. The wallet's RPC provider sees the address, the recipients it looks up and the blocks it asks about, which can link a wallet to its private transactions, so a careful user should run their own node or choose an RPC they trust. If the website disappears, the keys still come from the wallet's signature, the notes can be rebuilt from public events, and any client that reproduces the derivation in Section 3, including the exact message M, can spend them.

14. Conclusion

We have specified a payment pool for HYPE in which the ledger verifies every transaction and learns only what it must. Commitments and nullifiers prevent double spending without revealing which note was spent. A single proof checks conservation of value and the owner's wallet signature, so a leak of privacy keys costs privacy and leaves funds in place. Gas paid from inside the pool keeps the user's address off private transactions. What remains visible is the edge of the pool, and we have given users a way to reason about it. The more people use the pool, and the longer they stay, the better it protects all of them.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, M. Virza, "Zerocash: Decentralized Anonymous Payments from Bitcoin," In IEEE Symposium on Security and Privacy, 2014.

[3] Kinetiq, "Elysium documentation," `https://elysium.kinetiq.xyz/docs`, 2026.

[4] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso, T. Hess, "ERC-4337: Account Abstraction Using Alt Mempool," 2021.

[5] L. Grassi, D. Khovratovich, M. Schofnegger, "Poseidon2: A Faster Version of the Poseidon Hash Function," In AFRICACRYPT, 2023.

[6] W. Chang, G. Rocco, B. Millegan, N. Johnson, O. Terbu, "EIP-4361: Sign-In with Ethereum," 2021.

[7] H. Krawczyk, P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, 2010.

[8] T. Pornin, "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," RFC 6979, 2013.

[9] R. Bloemen, L. Logvinov, J. Evans, "EIP-712: Typed Structured Data Hashing and Signing," 2017.

[10] Aztec Labs, "Noir" and "Barretenberg," `https://noir-lang.org`, `https://github.com/AztecProtocol/aztec-packages`.

[11] A. Gabizon, Z. Williamson, O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge," IACR ePrint 2019/953, 2019.

[12] B. Chen, B. Bünz, D. Boneh, Z. Zhang, "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates," In EUROCRYPT, 2023.

[13] A. Kate, G. Zaverucha, I. Goldberg, "Constant-Size Commitments to Polynomials and Their Applications," In ASIACRYPT, 2010.

[14] A. Langley, M. Hamburg, S. Turner, "Elliptic Curves for Security," RFC 7748, 2016.

[15] S. Arciszewski, "XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305," IRTF CFRG Internet-Draft, 2020.

[16] Volta, "Docs," `https://volta.family/docs`, 2026.

[17] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, D. Robinson, "Uniswap v3 Core," 2021.

[18] R. Barbulescu, S. Duquesne, "Updating Key Size Estimations for Pairings," Journal of Cryptology, vol. 32, pages 1298–1336, 2019.