WHITEPAPER · VERSION 1.1 · OCTOBER 2026

ZELRA: privacy infrastructure for onchain finance

Proving that a financial condition holds, without revealing the data behind it.

ZELRA.NETWORK STATUS: PRE-LAUNCH

Abstract

ZELRA is a verification layer for Ethereum. An issuer, such as a bank or a custodian, attests a numeric value about a holder: the collateral it holds for them, a credit score, a balance. The holder can later prove to any onchain application that the value is at least some threshold, that the credential had not expired, and that it belongs to a set the application trusts, without revealing the value, the credential, or who they are. The proof is a Groth16 zero-knowledge proof made in the holder's browser in about half a second and checked by a contract for about 240,000 gas.

Applications consume proofs through one contract, `Zelra`, which enforces that the proof is fresh, that it was made against a current state of the set, and that each identity uses a credential at most once per application and purpose. Two reference applications ship with the layer: a credit pool that lends against collateral it never sees, and an eligibility drop that pays to addresses with no history. This paper describes the construction, states what the chain learns and does not learn, lists the trust assumptions, and names the limits.

---

1. The problem

Public blockchains are transparent by default. Every balance can be read, every transfer leaves a trace, and every relationship between addresses can be mapped. For most of finance that is disqualifying. A fund cannot borrow against a position that its competitors can read. A business cannot settle with suppliers on a ledger that discloses its margins. A person cannot prove they qualify for something without handing over the numbers that qualify them, to the checker and, often, to everyone.

The usual answer has been privacy as a feature of each application: every protocol builds its own mixer, its own encryption scheme or its own committee. Each one adds trust assumptions, audits and attack surface, and every other application on the chain stays transparent. ZELRA takes the other route. Privacy becomes infrastructure: one verification layer that any application can consume the way it consumes any other state, and what it verifies is a *condition*, not data.

> **What ZELRA is not.** It is not a private payment system and not a mixer. Token transfers remain as visible as the chain makes them. ZELRA hides the inputs of a decision, not the decision itself. A lending pool still shows who borrowed and how much; it no longer shows, or holds, the collateral that justified the loan.

2. Design goals

Reveal the proof, not the data

The only thing that leaves the holder's device is a proof that a condition holds. The value, the credential and the identity stay where they are.

One layer, every application

Lending, payments, trading, real-world assets, identity and compliance consume the same proofs through the same contract.

Permissionless issuance, chosen trust

Anyone may open a credential set. Each application names the sets it trusts when it is deployed. Trust is a decision of the consumer, not of the layer.

Unlinkable across applications

The same holder leaves a different, unrelated tag in every application and purpose. No party can join them.

Nothing to custody, no one to call

No admin keys, no upgrade path, no pause switch, no server that holds keys or data. The layer is a pair of contracts and a circuit.

Ordinary wallets, browser-side proving

A holder needs a wallet that signs a message. Keys are derived from that signature; proofs are made in the page.

3. System overview

Four roles take part. An **issuer** knows a value about a holder and is willing to attest it. A **holder** wants to use that value without disclosing it. A **consumer** is an application, usually a contract, that will act on the condition. A **relay** is an optional service that submits a holder's transaction so that the holder's wallet never appears on the chain next to the action.

1 · Holder derives a key and shares a ZELRA ID → 2 · Issuer attests a value and an expiry into a set →

3 · Holder proves the condition in the browser → 4 · Consumer calls `Zelra.consume` or `attest` →

5 · The chain records a tag, a threshold and a signal

Every step keeps the sensitive material on one side of a line. The issuer puts a *hash* of the credential on the chain, and the credential itself travels to the holder encrypted. The holder's secret never leaves the browser. The consumer receives 256 bytes of proof and a handful of public values, none of which is the value, the leaf, or the identity.

---

4. Identity and credentials

4.1 Keys from one signature

A holder's whole identity is derived from one wallet signature. There is no new seed to back up and nothing to register.

`sig`	An EIP-712 signature of a typed `Identity` message in the domain `ZELRA`, bound to the chain id. The wallet shows it as what it is, including a warning that the signature is the key. Deterministic for ECDSA wallets that follow RFC 6979 (MetaMask, Rabby, hardware wallets).
`rs`	The canonical 64-byte `r ‖ s` of that signature (low s), so every encoding a wallet may return gives the same key
`secret`	`keccak256(rs) mod p`, where p is the BN254 scalar field. The secret never leaves the browser.
`idc`	`Poseidon(secret)`, the identity commitment. The public half of the identity, never written to the chain in the clear.
`encPriv`	`keccak256(rs ‖ "zelra/enc") mod n`, a secp256k1 key to which issuers encrypt credentials

Because the derivation is deterministic, signing again on another device yields the same identity, and forgetting the key on a device loses nothing. Because it is bound to a wallet and a chain, a different wallet or a different chain is a different identity: ZELRA never links two wallets of the same person. A typed message, rather than a plain text one, makes it hard for another site to trick a holder into signing the same thing.

4.2 The ZELRA ID

What a holder hands to an issuer is a ZELRA ID: `zelra1:` followed by the base64url encoding of 69 bytes, the identity commitment (32 bytes), the compressed encryption public key (33 bytes) and a 4-byte checksum, the

first bytes of `keccak256` of the two. It carries enough for the issuer to compute a leaf and to encrypt a memo, and nothing that identifies the holder or reveals any other credential they hold.

4.3 Credentials and memos

A credential is four numbers: the identity commitment, a value, an expiry and a random salt. The issuer computes the leaf `Poseidon(idc, value, expiry, salt)` and inserts it into its set. The salt matters: without it, anyone holding a ZELRA ID could hash candidate values and recognise the holder's leaf on the chain. Next to the leaf the issuer publishes a memo: the value, the expiry, the set, the time of issue and the salt, encrypted to the holder's key with an ephemeral ECDH on secp256k1 and AES-256-GCM over a fixed-length plaintext, so not even the size of a memo hints at the value. The memo is opaque to everyone else. A holder discovers their credentials by reading every memo of a set and keeping the ones that decrypt and whose leaf matches; the node they ask sees a holder reading a whole set, never which leaf is theirs.

The expiry is the credential's only revocation mechanism, by design: the sets are append-only, so nothing is ever deleted, and an issuer who wants to withdraw an attestation simply stops renewing it. For collateral, short validity periods are the sensible choice.

---

5. Credential sets

A set is an append-only Merkle tree of depth 20, hashed with Poseidon, with room for 1,048,576 credentials. Anyone may open one by calling `createSet(name, schema, unit)`; the caller becomes its issuer and alone may insert leaves. Issuing one credential updates the twenty nodes on the leaf's path. Issuing a batch of up to 64 updates the tree level by level, so a batch of k leaves costs about 2k + 20 hashes instead of 20k, and produces one new root.

Each set has a current root. A root that has just been replaced stays valid for one hour, so a proof made a moment before a new issue is not rejected by it. The issuer of a set can be rotated in two steps, an offer and an acceptance, so a mistyped address cannot strand a set.

The layer itself trusts no issuer. A set is exactly as trustworthy as the party that issues into it, and every consumer chooses, at deployment, which set it will accept. Choosing an issuer is like choosing a custodian. The reference deployment goes through a helper contract, `ZelraSetup`, that opens the sets and deploys the consumers bound to them in a single transaction, so the credit pool trusts exactly the set that transaction created and no one can slip a lookalike set in between. Names are not identities: two sets can share a name, which is why consumers bind to a set id and drops to an issuer address.

6. The statement

A proof establishes the following statement, with six public values and everything else private:

> "I know a secret whose commitment, together with a value, an expiry and a salt, forms a leaf of the set whose root is `root`; the value is at least `threshold`; the expiry is later than one hour after `provedAt`; `nullifierHash` is Poseidon of my secret and `scope`; and the proof is bound to `signal`."

PUBLIC, IN ORDER	MEANING
`root`	The set's root the proof was made against
`threshold`	The bar the value clears; chosen by the consumer
`provedAt`	The chain time the proof speaks for; the credential outlives it by more than an hour
`scope`	The consumer's scope, derived from the chain, its address and a context
`nullifierHash`	The holder's one-time tag for that scope
`signal`	A value the proof is bound to: the caller, or a recipient

PRIVATE	MEANING
`secret`	The holder's secret
`value`, `expiry`, `salt`	The credential's contents
`pathElements[20]`, `pathIndices[20]`	The Merkle path from the leaf to the root, which also hides which leaf it is

The value and the threshold are range-checked to 64 bits before the comparison, and so are the expiry and `provedAt`; the expiry is compared with `provedAt + 3600` on 65 bits, so no arithmetic wraps around the field. Requiring the credential to outlive the whole hour in which a contract will accept the proof means an accepted proof never speaks for a credential that has already expired. The signal enters the circuit as a public input that is squared and discarded; that is enough to bind it, because changing a public input invalidates the proof. The circuit has 12,468 constraints. In a current browser a proof takes about half a second to make; it is 256 bytes long and costs the verifier about 240,000 gas. The proving system is Groth16 over BN254, produced with circom and snarkjs.

7. Verification

The contract `Zelra` is the only entry point consumers need. It offers three verbs.

VERB	WHAT IT DOES
`verify`	A view. Checks a proof against a set and returns true or false. Records nothing.
`consume`	Verifies in the caller's scope and spends the nullifier. One use per identity per consumer and context: drops, votes, allow-lists.
`attest`	Verifies in the caller's scope and records the event without spending the nullifier. For consumers with their own policy, such as a credit line that may be renewed by the same identity.

The scope is `keccak256(chainId, consumer, context) mod p`, computed by the contract from `msg.sender`. A proof made for one consumer therefore verifies only when that consumer submits it, and the nullifier it carries exists only in that consumer's space. The same holder presenting the same credential to two applications leaves two tags that no one can relate; presenting it twice to the same application and context leaves the same tag, which is how "once per holder" is enforced without knowing who the holder is.

Every accepted proof must be fresh: `provedAt` may not lie in the future and may not be more than one hour old. The circuit guarantees the credential is alive for the whole of that hour, so no proof can be accepted for a credential that has expired, and an old proof cannot be replayed later. The root must be the set's current root or one replaced less than an hour ago. On success the contract emits `ProofVerified(setId, consumer, context, nullifierHash, threshold, signal)`, which is all the chain will ever know about the event.

The layer has no owner. There is no function that pauses it, upgrades it, lists or delists issuers, or touches a set it did not create. A bug would be fixed by deploying again at a new address and pointing applications there.

8. Applications

8.1 Private credit

`ZelraCredit` is an ERC-4626 pool. Lenders deposit a dollar token and receive shares; borrowers pay 8% a year nominal through a global index that compounds whenever anyone interacts with the pool (at most about 8.33% effective), plus an origination fee of 0.1% on every draw, added to the debt and paid to the pool, so that borrowing and repaying in the same block is not free liquidity. To open a line of credit of some limit, a borrower proves that an unexpired credential in the pool's trusted set attests collateral of at least 1.5 times the limit. The contract computes that threshold itself and binds the proof to `msg.sender`, so a copied proof is useless to anyone else.

The amount of collateral, the credential and the holder's identity stay private; the chain sees a line, its limit and its debt.

The pool uses `attest` and keeps its own policy: one credential identity backs one account forever. The same account may prove again to raise its limit or to renew after the 30-day term, each time with a proof newer than the last, but a credential can never back a second account and an account can never switch identity. Lenders may withdraw whatever is not lent out. Fourteen days after a line's term ends with at least one dollar unpaid, anyone may mark it defaulted and is paid a bounty of 0.1% of the write-off, at most $100, from the pool's cash, so defaults are marked in the first block they can be. The debt is written off for good, the loss shows in the share price at once, and the account is barred. A residue under one cent is forgiven when a line is closed, so no one is barred over rounding. Collecting the debt is the issuer's business, against the collateral it attested; the pool proves eligibility, it does not seize anything. Shares carry a six-decimal virtual offset, which makes the first-depositor inflation attack uneconomic.

8.2 Eligibility drops

`ZelraDrop` lets a sponsor fund an allocation for holders whose credential in a set clears a threshold, for example a credit score of 700 or more. A holder claims a fixed amount to any address, typically a fresh one with no history. The proof is bound to the recipient, so anyone may submit it and no one can redirect it, and `consume` makes each identity claim at most once per drop. The sponsor learns how many claimed, never who or with what score, and takes back whatever was not claimed after the deadline, to its own address or to one it names in advance if its own can no longer receive the token. When funding a drop the sponsor names the issuer it trusts, and the contract refuses a set issued by anyone else, so a lookalike set with the same name cannot be funded by mistake. The contract also verifies that it really received the full funding, so a token that charges a fee on transfer cannot leave the last claims unpaid.

8.3 The relay

A claim to a fresh address is only private if the transaction that delivers it does not come from the holder's own wallet. The relay is a small service that accepts a proof over HTTPS, simulates it, submits it to `ZelraDrop.claim` and pays the gas. It cannot redirect a claim, because the recipient is inside the proof; it never receives the holder's wallet or secret; the worst it can do is refuse. It refuses a nullifier it has just submitted and limits how fast any client can send. It does see the network address a request comes from, next to the recipient; a relay should keep no access logs, and holders who care should reach it through a VPN or Tor. Without a relay the holder's wallet submits the claim and the link between the two addresses is visible on the chain, and the application says so.

9. What the chain learns

VISIBLE ON THE CHAIN	NEVER ON THE CHAIN
That a set exists, who issues it, how many leaves it has, its roots	Which holder a leaf belongs to
Each leaf, as a Poseidon hash, and an encrypted memo	The value, the expiry, the identity commitment
Each accepted proof: set, consumer, context, nullifier, threshold, signal, time	The holder's secret, the leaf used, the actual value
In the credit pool: the borrower's wallet, limit and debt	The collateral, the credential, who holds it
In a drop: the recipient address and the amount	The claimant's score, identity, or wallet (with a relay)

Some things remain inferable and should be understood before relying on the system.

- **The anonymity set is the credential set.** A proof shows that *some* leaf of the set clears the threshold. A set with three leaves hides little; a set with thousands hides a lot. Issuers and consumers should prefer large sets and coarse thresholds.
- **Thresholds are public.** Proving "collateral at least 150,000" reveals exactly that. Consumers should ask for the bar they need and no more.
- **Timing.** An issue followed by a proof against the new root a minute later is a weak correlation; the one-hour root grace exists so proofs need not be made in a hurry.
- **The issuer knows everything about its own credentials.** ZELRA hides data from the chain and from consumers, not from the party that attested it.
- **The signal can be an identity.** In the credit pool the signal is the borrower's wallet, on purpose: the pool must know whom it lends to. What stays private is what backs the loan.
- **The node you read from** sees which sets a browser scans. It does not learn which leaf is the holder's, because every memo of the set is read. Like the relay, it sees the browser's network address; a holder's own RPC endpoint avoids that.
- **Across chains** the same wallet has a different identity and leaves different nullifiers, because both the identity signature and the scope include the chain id.

10. Security and trust

Soundness. Nobody can prove a false statement, under the standard assumptions of Groth16 on BN254 and provided the trusted setup was honest. Phase one is the Perpetual Powers of Tau ceremony with 80 contributions. Phase two is specific to this circuit and, at the time of writing, has a single contributor: the development setup. A multi-party phase-two ceremony is part of the launch sequence, and the layer should not hold real value before it has run, because a party that kept the setup's randomness could forge proofs. The ceremony tooling ships with the project; one honest participant is enough.

Issuer honesty. An issuer that attests a false value lets its holder prove a false condition. Nothing in the layer detects this; it is the consumer's choice of set that bounds the risk. A credit pool deployed against a bank's set trusts that bank as it would trust a custodian.

Replay and theft. A proof is bound to a signal, to a scope, to a root and to a moment. Submitting someone else's proof fails because the scope does not match the caller; submitting a proof with a different recipient or caller fails because the signal is a public input; submitting it later fails after one hour or after the nullifier is spent; submitting it against another set fails because the root is not that set's. Front-running a transaction that carries a proof gains nothing, because the proof only benefits the signal it names.

Contracts. The contracts use custom errors, the checks-effects-interactions order, SafeERC20 and re-entrancy guards on every function that moves tokens; leaves are checked to be non-zero field elements and thresholds to fit in 64 bits; every loop is bounded. There are no admin keys and no upgradeability, which also means no emergency switch: in an incident lenders withdraw, sponsors close their drops after the deadline, and the application is pointed at new contracts. The test suite, 78 Foundry tests including invariant suites that drive random sequences of calls, plus tests against forked mainnet state, runs real Groth16 proofs through every rule: stale and future-dated proofs, outdated roots, reused nullifiers, altered signals and thresholds, tampered proofs, the wrong consumer or context, the default cycle and its bounty, the inflation attack and fee-on-transfer tokens. Eight false statements, among them a value below the threshold, a wrong salt and a credential that expires inside the proof's window, are checked to be unprovable.

Internal audit. The contracts, the circuit and the application were reviewed by several reviewers working in separate contexts, with static analysis, invariant testing and fork tests. They found 4 High and 8 Medium issues and no soundness break in the circuit, the verifier, the trees or the nullifier logic. Every issue that can be fixed in code has been fixed, including the salted leaf, the typed identity signature, the origination fee, the default bounty and the single-transaction setup. The open High is the ceremony above. An internal review is not an independent one; a third-party audit remains part of the launch sequence.

Operational dependencies. The dollar token of the reference pool is USDC, whose issuer can freeze addresses. The relay is a single service; if it is down, claims can still be submitted from a wallet at the cost of the link. Browsers keep the derived key in local storage; a compromised browser compromises the identity for that wallet, and the holder's remedy is a new wallet.

11. Limitations and future work

- **One statement shape.** The circuit proves "value at least threshold" on a single credential. Equalities, ranges with an upper bound, sums across credentials, and statements about transaction history each need a circuit of their own. The verification layer is designed to host more circuits.
- **Revocation only by expiry.** Sets are append-only. An issuer cannot delete a credential before it expires; it can only choose short validity periods and decline to renew.
- **Trusted setup.** Groth16 needs a per-circuit ceremony. Every new circuit, and every change to this one, needs one.
- **A window before default.** During the 14-day grace after a line's term, a lender who withdraws escapes a loss that is already likely. The bounty makes the default visible the first moment it can be, but does not close that window.
- **A plain interest model.** One fixed rate, no utilisation curve.
- **Ordinary wallets only.** The identity needs a plain ECDSA signature; smart-contract wallets are not supported yet.
- **Proving on the client.** The proving key is 5.6 MB and the witness is computed in the page. This is fine on current laptops and phones and would need work for very constrained devices.
- **Single chain.** The reference deployment targets Ethereum. The contracts are chain-agnostic and the deployment tooling knows Base and the test networks, but a credential set lives on one chain.
- **No independent audit yet.** The contracts and the circuit have been audited internally and tested, not audited by an independent party. An independent audit precedes any deployment that holds value.

12. Status and roadmap

At the time of writing the circuit, the contracts, the browser prover, the issuer and holder tooling, the relay and the deployment scripts exist, are tested and have been audited internally. Nothing is deployed on a public network yet; the application shows every screen as read-only until it is. ZELRA does not go straight to mainnet. The sequence is fixed and ordered:

1. **Public test phase.** Deploy on a test network with test dollars and a running relay, open the application to the public, and let issuers, holders, lenders and sponsors walk every flow with real wallets. Fix what they find.
2. **Ceremony.** Run the multi-party phase-two ceremony and publish its transcript and the hash of the proving key.
3. **Independent audit.** Freeze the code and have the contracts and the circuit audited by a third party.
4. **Mainnet.** Deploy to Ethereum through the single-transaction setup, verify the sources, hand the two reference sets to their issuer, publish the addresses, and run the relay as a service.

5. **Small first.** Smoke-test with small amounts, then open.

---

13. References

1. J. Groth. *On the Size of Pairing-based Non-interactive Arguments.* EUROCRYPT 2016.
2. L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems.* USENIX Security 2021.
3. iden3. *circom* and *snarkjs*: circuit compiler, Groth16 prover and verifier. github.com/iden3.
4. Privacy and Scaling Explorations. *Perpetual Powers of Tau.* Phase-one ceremony for BN254.
5. Semaphore. *Signaling anonymously with zero-knowledge proofs.* The identity commitment and nullifier pattern that ZELRA's scopes extend.
6. ERC-4626, *Tokenized Vault Standard*; EIP-6963, *Multi Injected Provider Discovery*; RFC 6979, *Deterministic Usage of DSA and ECDSA*.

© ZELRA · PRIVACY INFRASTRUCTURE FOR ONCHAIN FINANCE