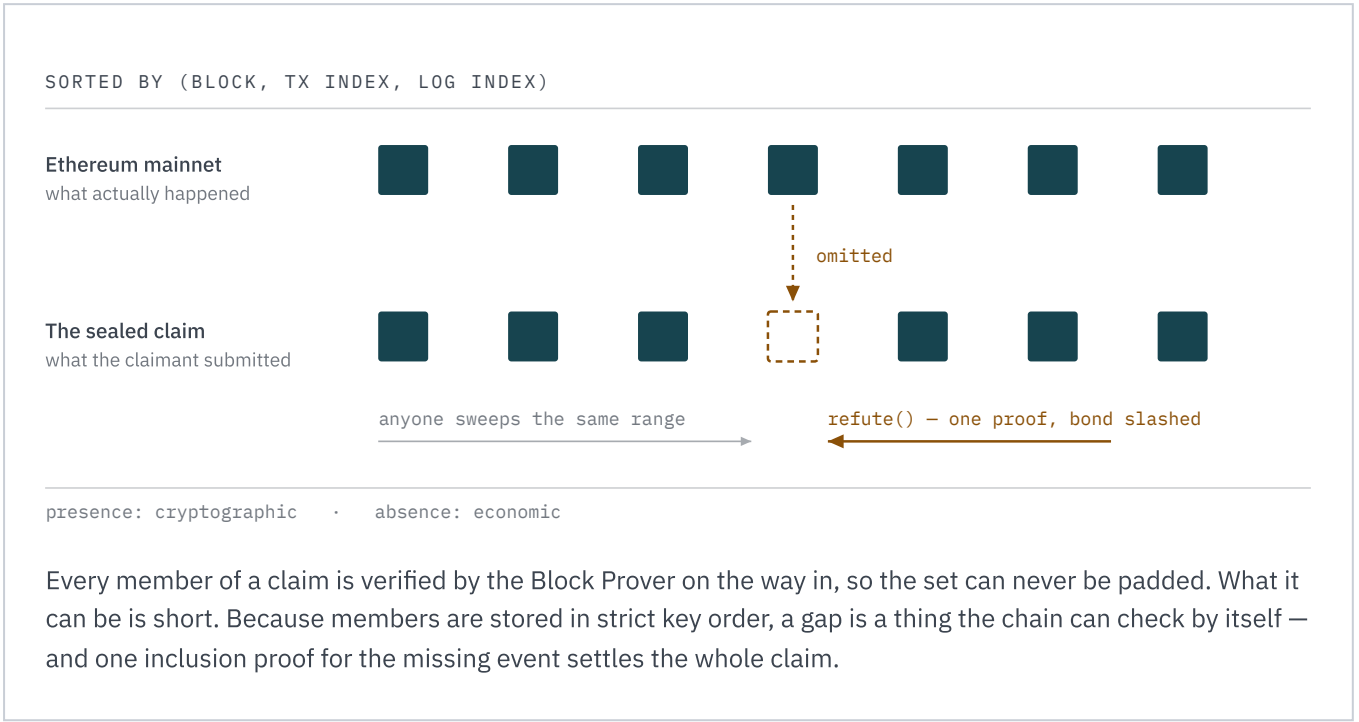
Utuh *Indonesian: whole, intact, nothing missing*

Attestcoin proves a transaction *happened*. It cannot prove a set of them is complete. Utuh closes that gap, and builds undercollateralized credit on top of it.

Put it as a question. Any registry on this protocol can tell you what it holds. Ask one whether a borrower has *ever been liquidated* and it can only answer with what someone chose to submit — and the borrower will not be submitting that.

BUIDL CTC 2026 FALL · CREDITCOIN CC3 TESTNET · ETHEREUM MAINNET · UTUH.VERCEL.APP



Every member of a claim is verified by the Block Prover on the way in, so the set can never be padded. What it can be is short. Because members are stored in strict key order, a gap is a thing the chain can check by itself — and one inclusion proof for the missing event settles the whole claim.

THE GAP

An inclusion proof cannot speak about events that do not exist

A Merkle proof places a transaction in a block. A continuity proof anchors that block to an attestation on Creditcoin. The precompile at `0x0FD2` checks both natively, inside a

single block. None of that notices what was left out — and whoever submits the proofs chooses which proofs to submit.

For credit, the sentence that matters is the one nobody can prove:

> *This borrower has never been liquidated.*

A borrower assembles their own history, submits the flattering half, and every proof checks out. This is not hypothetical. Season 1 of this hackathon drew 76 submissions; more than twenty were on-chain credit scores or reputation lending, and one of those, CreditX, took a prize. Every one of them inherits the hole, the winner included — placing well is not the same as closing it, and an inclusion proof gives none of them a way to.

Creditcoin's own `loan-flow` tutorial shows the shape cleanly, and it is the reference most builders will start from. `_markLoanAsFunded` and `_noteLoanRepayment` each take one proven transaction; `ASCBase.execute` verifies it through `0x0FD2` and records the query so it cannot be replayed. Every present fact is cryptographic, and that half is sound. Nothing asks whether the set is complete — which is correct for a loan already registered on chain, and insufficient the moment the question is about a stranger. And `markLoanAsExpired` is `onlyOwner`: default is declared by somebody trusted, not proven. For a tutorial that is an honest simplification. For underwriting it is the problem moved one layer up.

THE MECHANISM

Two halves, each sound on its own

NOTHING INVENTED

Members are proven on entry

Every event enters through `appendBatch`, which runs the Block Prover on it before it is recorded, checks the receipt status, and takes the transaction index from the proof rather than the caller. An aggregate cannot be inflated.

NOTHING OMITTED

Completeness is bonded

The claimant stakes the assertion that the set is complete. Anyone may break it by proving one in-scope event the set does not contain. Absence is never proven — a claim of absence is refuted by presence, which Attestcoin does prove.

What scales, and what does not

Settling a claim is constant work: the registry never verifies a whole set, so one spanning ten thousand events is broken by a single proof or by none at all. *Building* one is not. `npm run gas` finds every transaction the registry has ever seen from the registry's own logs, reads the receipts and fits a cost model — no explorer involved. Across the four registries deployed by 2026-08-28, when this fit was taken, 139 transactions:

CALL	GAS, MEAN	OF A 75M BLOCK
`open`	252,750	0.33%
`seal`	206,010	0.27%
`appendBatch`, 1 event	552,956	0.73%
`appendBatch`, 10 events	2,662,045	3.54%
`refute`	611,556	0.81%
`finalize`	209,258	0.27%

Member count does not explain those numbers. One append of *three* events cost 541,464 gas; one of *two* cost 878,903. Fitting the 56 appends the published registries had seen by 2026-08-28 against the call's own calldata gas as well as its member count says why, and holds the worst residual to 32% of the mean append:

```
290,899 gas fixed
  1.51 x the call's own calldata gas   (1.00 would be exact)
 81,427 gas per member on top of its bytes
```

The calldata term is the solid one, and the interesting one: **a proven transaction costs about twice its own calldata gas**, because those bytes are not merely paid for at the door — they are copied, RLP-decoded and hashed. Proving one in-scope log inside a fat mainnet transaction means carrying all thirty kilobytes of it, and that is not a choice the claimant has.

The per-member term is *not* well determined, which is worth saying rather than quoting. An earlier fit over 25 appends put it at 20,526 gas — almost exactly a cold `SSTORE`, a satisfying number and the reason to distrust it. Eleven more appends moved it to 61,265, and twenty more to 81,427 while the calldata multiplier fell from 2.3 to 1.5 — the two terms trading against each other, which is what collinear regressors do. A re-run on 2026-09-13 over the mainnet registry's last 85 transactions reads

317,593, 1.40 and 88,066. Members and bytes are correlated here, and separating them needs appends this repo has not made.

So the ceiling is bytes, not events. A ten-thousand-event claim is a thousand batches and about thirty-three full blocks of gas (32.7 in the 08-28 fit, 33.1 on the 09-13 re-run). Challenging it is still one proof and a binary search.

CLAIM LIFECYCLE

01	**open** — scope, block range, bond. The range must already be attested; see below.
02	**appendBatch** — up to ten queries against one shared continuity proof, in strictly ascending key order.
03	**seal** — the set is published and the challenge window starts. An empty set is a legitimate claim, and the most refutable one there is.
04	**refute** — one proof of an omitted in-scope event. Half the bond to the refuter, half burned.
05	**finalize** — the window elapsed unbroken. The bond returns; the recorded stake stays for consumers to weigh.

THE PART THAT IS EASY TO GET WRONG

A window nobody could have acted inside is not a window

A claim may not open until its *entire* range is already attested on Creditcoin, checked against the ChainInfo precompile at `0x0FD3`.

```
if (!CHAIN_INFO.is_height_attested(scope.chainKey, toBlock))
    revert RangeNotAttested(scope.chainKey, toBlock);
```

Without that gate a claimant could cover a range whose tail is not yet attested, and the window would expire on a claim nobody was *able* to break. Attestation heights only advance, so once `toBlock` is attested the range stays provable for the claim's life.

A second detail, equally load-bearing: the refuter takes half the slashed bond, not all of it. If they took all of it, a claimant caught lying could refute their own claim and walk away whole — making a false claim free to attempt. The burned half is what puts a price on being wrong.

SOURCE CHAIN

Underwritten on Ethereum mainnet, not Sepolia

CC3 Testnet attests Ethereum *mainnet* as chain key 3, from genesis height 0, tracking a few dozen blocks behind the real head — 34 measured on 2026-09-13, about 70 and about 100 on earlier days. Contracts on a free testnet can therefore be underwritten on real Aave positions and real USDC flows, with no capital at risk and nothing simulated.

```
NPM RUN BALANCE

attested   sepolia  chainKey 1  height 11530890
attested   mainnet  chainKey 3  height 25798250
```

Proofs for blocks 210,000 back still resolve in about seven seconds, with only a dozen continuity roots. Most entrants will reach for Sepolia by reflex.

BEFORE THE APPLICATION

What this does not give you

Utuh makes the inputs to a credit decision trustworthy. It does not make the loan safe. Those are different problems and only the first is solved here.

RISK	DETERRED BY	BOUNDED
The clean claim is false — the borrower was liquidated	the bond, refutable by anyone	yes, at `enforceableLoss`
The clean claim is true and the borrower defaults anyway	nothing	no

And the bond is not collateral. A claim must be finalized to be usable, and finalizing refunds the bond — so by construction **no bond is locked while a line is open**. What it bought was confidence in the history, spent before the money moved.

Two things the runs make visible. The cap has bound in every one of them, never the volume term, so the limit came from what someone staked rather than from the history — "underwritten on proven history" describes where the inputs came from, not where the number did. And the twenty percent in the formula has nothing behind it: an invented constant is exactly what twenty Season 1 projects shipped, and this is another one. The only difference is that its inputs cannot be faked.

THE APPLICATION

UtuhCredit — nothing bridges, only proof crosses

History stays on Ethereum. Credit is issued in CTC on Creditcoin. Repayment happens back on Ethereum. Underwriting rests on two claims that are adversarial in **opposite** directions, which is what makes the pair sound.

CLAIM	ASSERTION	A LIE PAYS BY	WHAT STOPS IT
Volume	proven Aave USDC repayments	inflating it	every member verified by `0x0FD2` on append
Clean	complete set of liquidations, normally empty	omitting one	bond, refutable by one liquidation proof

```
limit = min( 20% of proven volume x the lender's stated rate,
             10 x the bond behind the clean claim )
```

The second term is the consumer half of the mechanism. The registry cannot size a bond because it does not know what a claim will be used for; only the party about to lend knows its own exposure. A line never risks more than a liar stood to lose.

Reading a history is not the same as owning it

Underwriting reads a public chain, and nothing about reading it proves the reader holds the key that wrote it. Before a line opens, the borrower binds their Ethereum address to their Creditcoin account with one ordinary transaction:

```
calldata = bytes12("utuh:control") || <creditcoin account>
```

`proveControl` verifies it through the Block Prover and reads the sender out of the decoded transaction. No signature scheme of our own, no relayer to trust. The tag stops ordinary calldata from ever reading as a commitment, and naming the account inside it stops anyone replaying someone else's. Any supported source chain will do — an EOA address derives from its public key and is the same on all of them, so Sepolia gas proves exactly as much as mainnet gas.

Default without proving a negative

A drawn line is settled by the borrower proving repayment landed at the lender's Ethereum address. If no finalized claim arrives before the deadline, the line defaults. The contract never establishes that a payment was missed — the burden sits with the only party who could have discharged it. Silence is the default condition, not an inference.

EVIDENCE

Both outcomes, on the live chain

A real address that was genuinely liquidated on Ethereum files a claim saying it never was. A watcher sweeps the same range and takes the bond.

```
NPM RUN CREDIT

216001 blocks of Ethereum mainnet, ~30 days
  6130 USDC Repay events, 124 LiquidationCall events
  borrower and liquidated address are picked out of that window at runtime

volume: 4 proven Aave USDC repayments — aggregate 46871.138101 USDC
clean:  0 liquidations — the claim asserts this set is empty

0x58039c0c... was liquidated 1 time(s) in this window
filing a clean claim anyway — asserting the set is empty
  refuted with one real liquidation proof
  status Refuted   reward 1.0 CTC

volume claim 14: Finalized  aggregate 46871.138101 USDC
clean  claim 15: Finalized  aggregate 0

proven volume 46871.138101 USDC at 15000000000000 wei/unit
20% of that = 140613.414303 CTC
bond cap    = 10.0 CTC   (10x enforceableLoss 1.0, not the 2.0 CTC bond)

openLine reverted: SubjectNotControlled
  subject 0x476551e292547C70bB27307d87Df54bAeb0f644b
  caller  0x50577827700c0cF60240ABd07f041F09C95d4748
```

Two things bind here. The proven volume would justify a line of a hundred and forty thousand CTC; the clean claim it rests on carries a two CTC bond, and only half of that is enforceable — a claimant can front-run their own refutation from a second address and take the refuter's share back, so the burn is what is guaranteed. One CTC of enforceable loss, ten times that, ten CTC. A small bond is not allowed to underwrite a large loan, and it is the deterrent that sizes it, not the deposit.

A bond here stands behind a specific line, not a general reputation. `UtuhCredit.openLine` reaches a claim only through `UtuhRegistry.isUsable(claimId, exposure)`; the finalized claim is spent by the line it opens, `underwrittenThrough` consumes the history range so it cannot be underwritten twice, and the limit is capped at ten times the enforceable loss.

And then the line does not open at all. Every number above was read off a public chain by someone who does not hold the key that produced it, and the contract says so. The refusal is the demonstration.

THE WHOLE LOOP

Drawn on Creditcoin, repaid on Ethereum, settled

The mainnet run ends in a refusal, and that is the honest end of it. To show the other half of the product, the borrower has to be someone who can act — so the same contracts run again against Sepolia, with two parties transacting for themselves.

The source-chain contract there stands in for Aave, not for anything under test. The payments are real transfers, the events are real logs in real blocks, and Creditcoin attests them exactly as it attests Aave's. A scope is a scope; the registry cannot tell the difference and does not need to.

```
NPM RUN FULL

borrower sweeps for their own claim
  publicnode=3  tenderly=3   two independent endpoints agree; the union is what
gets claimed

borrower pays the lender 3 x 0.001 ETH        Sepolia blocks 11568493, 495, 496
adverse event recorded against the defaulter  Sepolia block 11568498

borrower binds their address
  calldata 0x757475683a636f6e74726f6c01a802c650cccef077208a93c1cf43025239003f
  controllerOf(0x01a802C6...) = 0x01a802C6...

claim 1  volume            Finalized   0.003 ETH, 3 members
claim 2  clean (borrower)  Finalized   0 members
claim 3  clean (defaulter) Refuted     one proof, half the bond burned
claim 4  repayment         Finalized   0.000525 ETH, 1 member

volume would justify 12.0 CTC; the clean bond guarantees 1.0 CTC of loss
limit 10.0 CTC                        (10x the enforceable loss, not the bond)
drawing 10.0 CTC obliges 0.000525 ETH back, at the lender's terms
borrower drew 10.0 CTC   (61.9868252125 -> 71.9867101675)
paid 0.000525 ETH back on Sepolia, block 11568588
borrower withdrew 4.0 CTC of returned bonds   (credited, then pulled)

line 1: Settled
settledThrough(borrower) = 11568619 — that payment cannot settle another line
```

Credit issued on Creditcoin against history proven from Sepolia, repaid on Sepolia, settled on Creditcoin. Nothing was wrapped, locked, or bridged — the only thing that crossed in either direction was proof.

THE WATCHER

A bond nobody tries to take is a deposit

Every guarantee here rests on one sentence — anyone may refute a claim by proving one event it left out. For a long time nothing in the repo was watching, which made that a possibility rather than a fact.

So: plant an incomplete claim, tell no one, and start a watcher that has never heard of it.

```
NPM RUN BAIT  ·  NPM RUN WATCH

claim 9 is sealed and short by one.
nobody has been told.

claim 9: sealed with 3 member(s), bond 2.0 CTC
  range 11553920..11554493 on chain key 1
  window closes at CC3 block 5363733 (now 5363713, 20 to go)
  swept independently: 127.0.0.1:1=err  127.0.0.1:2=err
  no endpoint answered — cannot say anything about this claim, will retry

    ... claim 6 is still Sealed, still queued ...

  swept independently: publicnode=4  drpc=err  1rpc=err
  union: 4 in-scope event(s)
  INCOMPLETE: 1 event(s) missing
  first gap at block 11554739 tx#126 log#0
  refuted with one proof. reward 1.0 CTC
```

The first pass had every endpoint unreachable and said so, leaving the claim queued rather than deciding it was fine on the strength of never having managed to look. The second pass caught it. Two of three endpoints still failed there and it did not matter, because a refutation verifies itself: the union is the right input and the Block Prover decides what is real. Had there been no gap, the same run would have said *inconclusive, 1 answered* rather than clean — the negative case is exactly as strong as the endpoints that looked, and saying so is the only honest option.

Nobody has to run that watcher. The console at utuh.vercel.app/app/ runs the same sweep from a browser — the daemon's own function, bundled into the page — and sends the refutation from the visitor's wallet. A GitHub Actions workflow sweeps both published registries every hour with no key, goes red if a sealed claim is short, and carries the watcher's read position between runs in the Actions cache so a claim sealed between two runs is not skipped. And `npx utuh-mcp` puts the role behind the Model Context Protocol: version 0.4.0 on npm and in the official MCP Registry, and a one-click `utuh-mcp.mcpb` on GitHub release v0.4.0 — five tools, claims as resources, and a confirmation demanded before the one tool that spends. The same server answers remotely at `https://utuh.vercel.app/api/mcp` (Streamable HTTP, no key; `refute_claim` returns an unsigned transaction).

COMPOSITION

A spotless Aave record says nothing about Compound

A clean claim only ever speaks about the contract its scope names. A borrower liquidated on Compound and untouched on Aave is not clean, and a lender that asked about Aave alone would never find out — the claim it read was true.

So the lender lists the adverse-event classes it cares about, and a line requires one finalized, empty claim for each. Exposure is capped at the *weakest* of them, since breaking any single one is enough to have made the underwriting wrong.

```
limit = min( 20% of proven volume x the lender's rate,
             10 x the smallest enforceable loss among the clean
claims )
```

FOUND BY ATTACKING IT

Eighteen holes, closed

Each of these was found after the flow already worked, which is exactly when a design stops being examined. The first one is the one this whole project is about: a check that reads convincingly and binds nothing.

```
/// @param repayRequired ... Set by the lender at draw time;
function draw(uint256 lineId, uint256 amount, uint256
repayRequired, uint64 repayWindow) external {
    if (l.borrower != msg.sender) revert NotBorrower();   // the
borrower calls this
    ...
    l.repayRequired += repayRequired;                      //
with a number they chose
```

The comment says the lender sets the terms. The function is callable only by the borrower, and takes the terms as arguments. Draw the whole limit, pass one wei, and the loan is free. The live run happened to be honest.

`draw` now takes an amount and nothing else. What must come back and by when are computed from lender policy, converting CTC through the same rate that produced the limit, rounding up so no draw is ever small enough to owe nothing.

- **Anyone could borrow against a stranger's record.** `openLine` read a subject's history and never checked who was asking. Closed by the control commitment above.
- **One underwriting could fund unlimited lines.** The bond cap bounded each line while bounding nothing in aggregate. A finalized claim is now spent when it opens a line.

- **A borrower could buy an unlimited extension for one wei.** Every draw reset the deadline. It is now fixed by the first draw and never moves.
- **A bond said nothing about how long it was exposed.** A lender now states the shortest challenge window it will accept, because a stake is only a deterrent for as long as somebody could still take it.
- **"Never liquidated" was checked by summing, not counting.** A clean claim's aggregate is zero when the set is empty — and also when it holds a liquidation whose amount happens to be zero. It now counts members, which is true whatever metric the scope carries.
- **A refutation could name an event the claimant was unable to include.** Appending runs the scope's metric and rejects a log whose data is too short for it; refuting did not. An honest claimant could have been slashed for an omission they had no way to avoid. Both paths now run the same check.
- **The bond overstated the deterrent by half.** A claimant knows what they omitted from the moment they seal, so they can front-run an incoming refutation from a second address and take the refuter's share back. An earlier note here said commit-reveal would close that; it would not — the claimant holds the private knowledge and simply commits first. What survives is the burn. The registry now reports `enforceableLoss`, and a line sized against the whole bond carried twice the exposure the deterrent covered.
- **The watcher forgot claims it had failed to check.** It marked each one seen *before* inspecting it, so an endpoint outage during the minute a claim sealed made that claim invisible for good — and an unguarded await meant a single lost race killed the process. It now retires a claim only on a verdict that cannot change, retries everything else, survives its own failures, and works soonest-deadline-first.
- **The endpoint list could only grow.** Widening a trust set has to come with narrowing it: an operator who knows a bundled endpoint is rate-limited — or is the claimant's — needs to drop it, and an append-only setting cannot.
- **A claimant who could not receive ether could brick their own claim.** `finalize` is permissionless and pays the claimant, so a contract that rejects the refund would have left its claim stuck in *Sealed* forever — and with it anything waiting on it. Refunds are credited and pulled now.
- **Honest claimants were betting the bond on one RPC.** Sweeping with a single endpoint means a node's omission produces an incomplete claim, which is indistinguishable from a lie and punished the same. Claim building now unions every endpoint, the same way the watcher does.
- **One payment could settle two lines.** Marking a claim spent stops the claim being reused; it does not stop the payment inside it being reused. Two lines, two claims over overlapping ranges, one transfer in both. Each settlement now consumes the range it rests on, per subject.
- **Refutation depended on one hosted service.** Every proof came from Gluwa's hosted Proof Builder, so an outage there meant no claim could be built and, worse, no claim could be *refuted* — the entire enforcement mechanism behind a single endpoint, which is the assumption the registry exists to reject. Proofs are now built locally as well, from the source-chain RPCs and the ChainInfo precompile, and both paths produce the same continuity roots: `npm run provers` proves one transaction each way and compares them. Then the real test — a planted incomplete claim refuted by a watcher run with `PROVER_URL`

pointed at a dead port. A real bond, taken on chain, with no hosted proof service reachable at all. The independent proof path is twenty to thirty times slower than the hosted one — 20 s on Sepolia, 30 s on mainnet against 0.9 s, measured 2026-09-13 — and produces byte-identical proofs; size the challenge window for it, which is the reason not to underwrite against the twenty-block floor.

- **A refutation depended on a node's willingness to do arithmetic.** `pallet-evm` does not always propagate revert reasons in estimation mode, so `eth_estimateGas` can fail on a precompile call that would have succeeded — Gluwa's own SDK ships a workaround, which is how this is known rather than guessed at. Every registry write went out with no fallback, so a node declining to estimate meant a liar kept the bond. Writes now run `eth_call` first, which is free and settles whether the call is good, then estimate, then a limit from the measured cost model. `FORCE_MODELLED_GAS=1` makes every call take the fallback, so it is exercised on real transactions rather than trusted because the arithmetic looks right.
- **A control binding could be replayed, so it was not a binding.** `proveControl` set `controllerOf[subject]` and kept no record of which commitments it had already applied. The proof stays valid forever and anyone may submit it, so the binding was really *whichever proof was replayed most recently*. A subject can move their binding by sending a second commitment — which is how anyone rotates away from a Creditcoin account they no longer control — and an attacker holding the old account could simply put it back, at will, including in front of the subject's own `openLine`. Creditcoin's own `USCBase` records processed queries for exactly this reason; this did not. Each commitment applies once now, keyed by chain and by the encoded transaction, which carries its own signature.
- **A chain key was treated as a global constant.** `CHAIN_KEY` says Sepolia is 1 and Ethereum mainnet is 3, which is true of CC3 Testnet and is not a property of Creditcoin — gluwa's own `networks.json` has key 3 meaning *Sepolia* on `usc-devnet`. Pointed at a different Creditcoin network, this would have underwritten one chain while reporting another, and nothing would have caught it: every proof would still verify, being a perfectly valid proof about the chain it came from. `get_supported_chains` is read now, and the key, the EVM chain id and the transaction encoding are all checked against what the network attests.
- **Endpoint disagreement was resolved by iteration order.** The union sweep wrote each endpoint's answer over the last, so two endpoints describing the same event position differently meant the one that answered last won, silently. The union is an argument about presence, and it does not extend to contents. Conflicts are reported now.

The first of those is why `npm run credit` ends in a revert rather than a drawdown. Pointed at a stranger's history it stops at `SubjectNotControlled`, and the refusal is the demonstration.

FOUND BY BUILDING

Two things about Attestcoin that are not in the docs

Both surfaced by exercising the precompile through `eth_call` on the live testnet before spending anything — the Block Prover exposes a `view` twin of `verifyAndEmit`, so an empty wallet is enough to test the entire proving path.

- **There is an undocumented batch overload.** The Proof Builder returns one continuity proof spanning a batch's block range, and it is the array form of `verifyAndEmit` that the proof is shaped for. Feeding the same shared proof to the single-query form works only while every query sits in the same block; one block later it reverts with `Merkle root mismatch`.
- **The cap of ten counts queries, not transactions.** A transaction carrying three in-scope logs spends three slots. Batching by transaction count earns `heights: Value is too large for length`.
- **The hash ChainInfo returns is the attestation's digest, not the block's header.** At Ethereum height 25,925,380 the precompile answers `0x8786ef14…`, the attestation indexer's `digest` field for that height answers the same, and Ethereum's own header hash for that block is `0xd5ea7299…`. Read as a header hash and compared against the source chain, it produces a mismatch about a perfectly healthy oracle.
- **`find_highest_attested_before` is exclusive of its target.** Height 25,925,400 is itself an attestation point, and asking for the highest attestation before it returns 25,925,390 — one full interval back. That is exactly what a claimant wants, since the frontier's own attestation is the one still settling, and exactly wrong for anyone using it to ask whether a height is attested.
- **The two Creditcoin networks sign the same Ethereum block into the same digest.** CC3 Testnet attests Ethereum mainnet under chain key 3 and Creditcoin Mainnet under key 1, from attestor sets with no BLS public key in common — five registered against seven — and their attestation digests for the same height are byte-identical. `get_attestation_height_for_digest` makes that checkable in two `eth_call`s from a browser: hand one network's digest to the other's index and it answers with the height. The two do not run in lockstep, so the comparison has to be taken at the lower of the two frontiers; asked at either network's own edge, the other has not indexed it yet and a lag reads as a disagreement.

STATED PLAINLY

What this does not do

- **Completeness here is economic, not cryptographic.** A bond makes lying expensive. It does not make it impossible — and what it costs is the burn, not the bond.
- **A watcher's silence is only as good as its sources.** Deciding a claim is complete means trusting some node to have mentioned every log — this protocol's own problem, one layer down. Voting across endpoints would not fix it; they can be wrong together. What saves it is that a refutation verifies itself: if *any* endpoint surfaces an omitted event, the Block Prover settles whether it is real, and a fabricated one just fails to prove. So the watcher sweeps every endpoint and takes the union, never a vote. The negative case stays soft and is reported that way — "no gap across 3 endpoints", or "inconclusive, 1 answered".
- **The mechanism punishes scale, though less than it did.** One omission voids the whole claim and there is no amend path, by design. Most of that risk was self-inflicted: claimants used to sweep with one RPC, so a node's omission cost them the bond and looked exactly like lying. They now take the union of every endpoint, and a single-source sweep warns before it seals. What remains is inherent — settling cheaply and building fragilely are one property from two sides — and still argues for shorter claims than the guarantee deserves.

- **Honest claims pay watchers nothing, and this layer cannot fix it.** A refutation earns only when someone lied; if the deterrent works, almost nobody does. Paying watchers from the burn pool was the obvious patch and it fails: any bonus worth having is also recoverable by a claimant refuting their own claim from a second address, which is the front-running that made `enforceableLoss` necessary in the first place. A token mechanism pretending otherwise would be worse than the honest gap.
- **Watching pays less than the reward suggests.** A refutation only earns when the claimant fails to defend against it. Pricing the guarantee at the burn keeps lenders honest about their exposure; it does not repair the watcher's side of the trade.
- **Large claims are expensive to build.** Not because of storage, but because every proven transaction is carried, decoded and hashed at roughly twice its calldata gas. A ten-thousand-event claim is about thirty-three full blocks of gas. Keys live in a storage array so refutation stays a binary search the chain runs itself, with no witness a claimant could withhold. The replacement — an incremental Merkle root per claim, the refuter supplying an adjacency proof of the two members bracketing the gap — is built and tested on branch `merkle-claims`, 165 tests run in CI on every push to master, and it is not deployed: a redeploy renumbers every claim this document cites.
- **One finalized claim can back a line at every lender at once.** The registry's `isUsable` is stateless and `claimSpent` belongs to one `UtuhCredit` deployment, so the same pair opens a full line at each lender that accepts it. Each lender's cap holds for its own line; nothing bounds the sum, while a liar loses the burn once. A registry-level reservation closes it, and is an ABI change.
- **A lender can deploy terms nobody can meet.** The constructor does not check the repayment window against the registry's challenge floor; set below it, every draw defaults. The published policy is 5,760 blocks against a floor of 25.
- **An overdue line can still be drawn.** `draw` checks the limit and the slot, not the deadline, so an unmarked overdue line can be drawn up to its limit — exposure stays bounded by that limit. Each of these three is pinned by a test so nobody rediscovers it.
- **Read-side only.** Writability is still in third-party audit and not on testnet, so a default is recorded on Creditcoin but consequences cannot yet be enforced back on Ethereum.
- **Binding an address costs the borrower one source-chain transaction.** That is a real onboarding step, and there is no way around it that does not reopen the hole it closes. It is also why the mainnet run cannot finish: no one can bind an address they do not hold.
- **Nobody outside this project has used it yet.** Every transaction into the four deployed contracts, 374 on 2026-09-14, was sent from one of four keys this project holds, so every claim and every refutation on both registries is ours. The mechanism is exercised end to end on the live chain, but no outside watcher, claimant or lender has used it.
- **There is no oracle, and none is faked.** Crossing from USDC units to CTC wei is a price; the lender fixes the rate at deployment where anyone can see and judge it.

READ IT BACK

Nothing here needs to be taken on trust

The landing page at utuh.vercel.app reads both registries from Creditcoin as it draws, and the console at utuh.vercel.app/app/ opens any claim by address: /app/?claim=5 is the claim sealed one event short and broken from a browser, and /app/?deployment=mainnet&claim=20 the false clean claim over 216,000 mainnet blocks, each opening with its verdict.

Measured on 2026-09-14: 212 events proven into 85 claims on the two published registries, 34 of those claims refuted by a proof of an omitted event, and 34 CTC of bond burned. The attestation indexer's own table of every `TransactionVerified` event the Block Prover has emitted holds 143,709 rows for CC3 Testnet, and 248 of them are this project's — 212 members, 34 refutations, 2 control bindings. Blockscout counts 376 transactions and 156.6M gas into the three listed contracts. `npm run judge` re-measures every one of these numbers against the chain, the explorer, npm and the MCP Registry with no key, and exits non-zero on any that no longer holds; 25 of 25 held on 2026-09-14, after the production deploy.

The contracts' own tests: `forge test` passes 211 tests, and `forge coverage` over `src/` reaches 100% of branches (104/104) and lines (432/432). Mutation testing (gambit 0.2.1, 738 mutants) killed 99.1%, 730 of 737; the 7 survivors are equivalent mutants, each explained in `test/MUTATION.md`, and the score was 93.9% before the 38 missing assertions it found got tests. A 30-minute medusa campaign passed all 22 properties and assertion tests over 639,295 calls, and caught both bugs planted in scratch copies of `src/`, shrunk to 7 and 16 calls.

DEPLOYED

CONTRACT	CC3 TESTNET — CHAIN ID 102031, ALL VERIFIED ON BLOCKSCOUT
UtuhRegistry *(mainnet-sourced)*	0x8FA0BD5301D998Be873E31453E53d114929a5Fac
UtuhCredit *(mainnet-sourced)*	0x89FB81b1e453b7Bd18ac1A6AF03C84A40Ce10C57
EvmV1Decoder *(mainnet-sourced)*	0x5cab00c032D7d4436f312Dd51ef59Dc5b860df3F
UtuhRegistry *(Sepolia-sourced)*	0x26880c8980Cd54827543bD34c6c613253c69347b
UtuhCredit *(Sepolia-sourced)*	0x0177aDb82152c8673a85271F7F06336B820324b6
EvmV1Decoder *(Sepolia-sourced)*	0x084c45552A6c45C7269F4a7041E757ABf4Bcc008
SettlementLedger *(on Sepolia)*	0xC8C9053C4E2c0590df684c12e5f2610EFeC9575B

---

211 forge tests, 10 fuzz, 16 invariants, 100% of branches and lines

99.1% of mutants killed 34 refuted, 34 CTC burned 121 guards asserted live

Slither 0, halmos 5 proofs Source chains: Ethereum mainnet and Sepolia

Registry 13.6 KB runtime github.com/PugarHuda/utuh MIT