

Randprotocol: Confidential Arbitrary Computations

A Shielded Ledger and a STARK-Verified Virtual Machine: Reference Implementation and Analysis

Dendi Suhubdy
Bitwyre
dendi@bitwyre.com

Anish Mohammed
Zeroknowledge Consulting Ltd
anishmohammed@proton.me

Implementation Draft 3 — September 2026

Written against the `randprotocol/fullnode` repository at commit `26ab1d4` (branch `main`, 2026-09-16): the shielded pool (phase S1), staking (S2), the bridge as notes and call envelopes (S3), zkVM constraint set 6, the recursion VM (milestones M5.1–M5.4), block-level aggregation (merged, genesis-gated) with the pre-v0.1 security review’s fixes, the RAND rename, and chain 10.

All statements concerning the system refer to that revision; functionality not present in it is identified as such. A fifth reconciliation pass (2026-09-24, `main` at `d050259`, releases v0.5 to v0.5.5, chain 14) is recorded as marked addenda where the revision moved: the four-slot hidden-asset bundle and its transaction binding, the token registry, the hardened bridge, the durable lock, the wallet-local tree, and the leading row of Table 6. A sixth pass (2026-09-27, `main` at `8e64781`, releases v0.5.6 to v0.5.8, chain 15) adds addenda for the call-header pin and tier cap, the frozen Poseidon2 constant table, the aggregate binding, and chain 15. A seventh pass (2026-09-29, `main` at `4ab3575`, releases v0.5.9 to v0.6.2, chain 16) adopts the corrections of the first Lean 4 inventory of the papers: the bundle relation is stated in its four-slot form, the supply identity carries the burned registration fee and the vesting register, the safety theorem names the lock-release rule outside its hypothesis, the zero-knowledge theorem names the table-height and blinding hypotheses, the sponge’s collision resistance is stated for fixed-length messages, and the aggregate’s soundness is stated as a theorem (Section 4.14).

Abstract

Settlement systems in conventional finance are confidential by default: account balances and transfer records are visible to the account holder, the institution and, under legal process, a regulator, and to no one else. Public blockchains invert this property. On Ethereum and Solana the global state and the pending-transaction queue are observable by every participant, which exposes institutional order flow to systematic front-running and discloses the counterparties and amounts of every stablecoin transfer. No production protocol currently combines the confidentiality of a bank ledger with the deterministic settlement, transparent verification and post-quantum security of a public chain.

This paper presents Randprotocol, a layer-one blockchain designed to close that gap, as implemented in its reference full node. The native asset, RAND, exists only as notes in a fully shielded pool: a depth-32 Poseidon2 commitment tree and a nullifier set replace accounts, and every transfer is a fixed four-input four-output *bundle* (two-input two-output until chain 13) whose validity is established by a transparent, hash-based STARK proof of the execution of a single pinned RISC-V guest program. The same proof system supports *confidential arbitrary computation*: RV32IM programs are deployed by content hash and invoked with a proof of execution in place of their inputs, and the chain records eight public output words. Consensus is chained HotStuff with stake-weighted quorums, a three-consecutive-view commit rule and epoch-derived validator sets drawn from a public register; stake is bonded from and withdrawn into the pool. A guardian-attested bridge deposits assets from four external chains as notes. Validator signatures are CRYSTALS-Dilithium2 and note envelopes use ML-KEM-768 with ChaCha20-Poly1305.

We describe the implementation component by component and then develop its formal analysis against the deployed parameters: the bundle and call relations proved by the nine-table zero-knowledge virtual machine, including the taint-accumulator construction that substitutes for an in-circuit assertion; the lookup-bus arguments binding the program digest and the salted input commitment; the FRI soundness of the production profile (80 queries, 20 grinding bits, blowup 8, a degree-2 extension of the Goldilocks field), stated under both the proven proximity-gaps bound and the ethSTARK conjecture; the value-balance and supply-audit identities of the pool; and the safety and liveness of the epoch-aware consensus rules.

The analysis identifies the properties the implementation does not yet provide. Fees, stake weights, deposit amounts at the pool boundary and program identities are public; the current wallet discloses to the queried node the leaf it intends to spend; an 80-query proof occupies approximately 1.3 MB (1.43 MB for the four-slot bundle of chain 14 onward), limiting a 4 MiB block to three shielded transfers (two from chain 14; chain 15 raises the block to 20 MiB); block-level proof aggregation, slashing, EVM and SVM guests and an independent audit are outstanding at this revision (the guests have since been vendored, though EVM calls are inadmissible under the tier-14 call cap, and three founder-led audit rounds have since been closed; no independent audit has been performed); and the GPU prover has not been executed on hardware. The software is deployed on a test network of 26 validators (chain 16 since 2026-09-28, every validator operated by the founding team) and is not suitable for value-bearing use.

Keywords: Confidential computation, shielded pool, note commitments, nullifiers, zero-knowledge virtual machine, STARK, FRI, Plonky3, RISC-V, HotStuff, staking, post-quantum cryptography, Dilithium, ML-KEM, front-running, confidential stablecoin, cross-chain bridge, reference implementation

Contents

1	Introduction	4
1.1	Motivation	4
1.2	System overview	4
1.3	Scope	5
1.4	Contributions	5
2	Background and Related Work	6
2.1	Byzantine consensus	6
2.2	Transparent proof systems	6
2.3	Zero-knowledge virtual machines	6
2.4	Privacy-preserving ledgers	6
2.5	Front-running	7
2.6	Post-quantum standards and bridges	7
3	Architecture	7
3.1	Workspace structure	7
3.2	Roles	8
3.3	Transaction flow	8
3.4	Disclosure summary	9
4	Implementation	9
4.1	Cryptographic primitives and identities	9
4.2	Notes, bundles, transactions and actions	10
4.3	Fee floors, programs and the executor interface	11
4.4	The ledger	12

4.5	Staking and the supply audit	14
4.6	Genesis	15
4.7	Chained HotStuff with epochs	15
4.8	Instruction set, syscalls and emulator	16
4.9	The bundle guest	17
4.10	The sponge and the two digests	18
4.11	Trace tables and buses	19
4.12	Machine configuration, proofs and verification	20
4.13	The chain-side executor	21
4.14	The recursion VM and block-level aggregation	21
4.15	Storage	24
4.16	Networking, mempool, event loop and RPC	25
4.17	The wallet	26
4.18	The bridge codec and the bridge as notes	26
5	Formal Analysis	27
5.1	Preliminaries and notation	27
5.2	The relations proved by the node	28
5.3	Soundness of the deployed parameters	31
5.4	Pool and ledger properties	32
5.5	Consensus properties	33
5.6	Information disclosed by a transaction	35
5.7	Bridge properties	35
6	Evaluation	36
7	Implementation Status and Limitations	37
8	Planned Work	42
8.1	Block-level aggregation and the prover market	42
8.2	Remaining pool work	43
8.3	zkVM milestones	43
9	Conclusion	43

1 Introduction

1.1 Motivation

Three observations motivate the design.

Confidentiality of conventional ledgers. A commercial bank maintains a ledger of balances and a log of transfers, both of which are disclosed only to the account holder, the institution and, under legal process, a regulator. Payroll, supplier payments, treasury operations and trading positions are settled under this regime. The confidentiality is functional rather than incidental: it permits an enterprise to pay suppliers without disclosing its cost structure to competitors, and a fund to accumulate a position without signalling its intent to the market. A settlement system intended to replace such ledgers for institutional use must provide equivalent confidentiality, and must do so by construction rather than by reliance on an operator.

Exposure of order flow on public chains. On Ethereum and Solana both the state and the pending-transaction queue are public. A participant submitting a large swap, a lending-position change or a token purchase announces its intent to every relaying node before inclusion. Daian et al. [13] documented the resulting front-running and sandwiching activity, which has since become a structural feature of both networks. Private relays and encrypted mempools [14, 15] relocate the trust to a relay operator or a threshold committee; they do not alter the fact that a transaction’s content is permanently public once included. For an institution the exposure is therefore not confined to the moment of execution but extends to the permanent disclosure of its positions, sizes and counterparties.

Absence of a confidential stablecoin rail. Dollar-denominated stablecoins are the settlement asset of on-chain finance, and every major instance, USDT and USDC on Ethereum, BNB Smart Chain, Tron and Solana, is a transparent token. Shielded-pool constructions exist for native assets, Zcash [16] being canonical, and both fully homomorphic [22] and zero-knowledge [19, 20] constructions exist for confidential contracts. None currently offers custody and transfer of a dollar stablecoin with hidden balances, counterparties and amounts on a chain with deterministic finality, post-quantum primitives and no trusted setup. The deficiency is one of system integration as much as of cryptography: a consensus layer, a proof system that validators can verify efficiently, a bridge for the external asset and a wallet capable of producing the proofs must operate together.

1.2 System overview

Randprotocol is a layer-one blockchain with two execution models sharing one proof system. The first is a *shielded pool*: value exists only as notes, each recorded on chain as a hiding commitment, and a transfer is a proof that two notes owned by the spender were consumed and two new notes of equal aggregate value, less a public fee, were created. The second is *confidential arbitrary computation*: a program, RISC-V machine code deployed once under its content hash, is invoked with a proof of execution rather than its inputs, and the chain records the eight words the program elected to publish. A validator executes no program and learns no balance; it verifies proofs.

The design commitments fixed in the implementation are the following.

- **A single verifier for all programs, including the transfer.** A general RV32IM machine is arithmetised once as nine interconnected trace tables. The shielded transfer is itself a guest program of that machine, **bundle**, whose digest is pinned at genesis. Deploying a program is a data operation rather than the construction of a new circuit.

- **Transparent, post-quantum proofs.** The proof system is a FRI-based STARK over the Goldilocks field, built on Plonky3 [31], with Poseidon2 as both the in-circuit and the transcript hash and 80 FRI queries. No trusted setup and no elliptic-curve assumption appears in the proof path.
- **Post-quantum signatures and envelopes.** Votes, blocks and validator actions are signed with Dilithium2 (ML-DSA, FIPS 204 [42]); note plaintexts are conveyed in envelopes sealed with ML-KEM-768 (FIPS 203 [41]) and ChaCha20-Poly1305. A transfer carries no signature: the proof constitutes authorisation.
- **Deterministic finality with public stake.** Chained HotStuff [11] commits a block once three certificates in consecutive views extend it. The validator register is the sole table holding amounts in the clear, and each epoch’s validator set is derived from it.
- **Inexpensive verification, prover-borne proving cost.** A validator verifies a bundle in approximately 16 ms with a warm verifier key; the wallet expends approximately 100 s producing it on a laptop.
- **Bridged assets as notes.** A guardian committee attests to lock events on four external chains; the chain itself derives the deposit note from the attested amount.

1.3 Scope

This is an implementation paper. Earlier revisions of the design [1] preceded the code. The design’s Part II — Verifiable Prover Attribution, with its beacon-assigned bonded provers, threshold-encrypted mempool, prover attestations and weighted-lottery leader rule — is not implemented and is now presented there as a research proposal; nothing in this paper depends on it. The present paper is written against the revision named on the title page, which integrates the three shielded-pool phases and the fifth zkVM constraint set. Section 3 describes the architecture; Section 4 describes the implementation component by component, quoting source where the source is the specification; Section 5 develops the formal analysis in terms of the relations, parameters and rules the implementation realises; Section 6 reports measurements; Section 7 enumerates the properties not yet provided; Section 8 describes planned work.

Two conventions govern the exposition. Every numerical figure is a measurement, and its source is identified. Every security property is stated against the implementation, including where the implementation is weaker than the design: the transcript hash is 256-bit Poseidon2 rather than 384-bit; the zero-knowledge property is statistical rather than perfect; the program digest is binding but not hiding; the current wallet discloses the leaf it intends to spend; and the consensus lock is not durable across a restart.

1.4 Contributions

1. A complete description of an operational shielded-pool and confidential-computation full node: a 39,000-line Rust workspace of five crates, its note, bundle, action and block formats, ledger admission rules, staking register and epochs, consensus state machine, storage layout, networking, wallet and bridge (Section 4).
2. A precise statement of the two relations proved by the node’s STARK: the bundle relation with its 47-word digest and taint accumulator, and the call relation with its in-circuit program digest hc and salted input commitment H_{IN} , together with the lookup-bus arguments that bind them (Section 5.2).
3. A soundness and zero-knowledge analysis of the deployed parameters: FRI at 80 queries (approximately 86 proven bits, 260 conjectured, bounded to approximately 100 by the

field term), the 256-bit Poseidon2 transcript (≈ 85 quantum bits), and the statistical hiding of the polynomial commitment (Section 5.3).

4. Determinism, atomicity, value-balance and supply-audit results for the pool, and safety and liveness results for epoch-aware chained HotStuff as implemented (Sections 5.4, 5.5).
5. A characterisation, by transaction kind, of the information disclosed by the chain, and of the throughput consequence of the 80-query proof (Sections 5.6, 6).

2 Background and Related Work

2.1 Byzantine consensus

Lamport, Shostak and Pease established that $n \geq 3f + 1$ replicas are necessary and sufficient to tolerate f Byzantine faults [8]. PBFT [9] made the bound practical with a three-phase, leader-based protocol. Tendermint [10] introduced the lock discipline that permits leader rotation at every block. HotStuff [11] pipelined the phases into a *chained* protocol in which each block carries a quorum certificate (QC) for its parent and a block is committed once three QCs in consecutive views extend it. The node implements chained HotStuff as a pure state machine with epoch-derived validator sets (Section 4.7). Leader election is round robin; the hash-based sortition of the earlier design, adapted from Algorand [12], is not implemented.

2.2 Transparent proof systems

Ben-Sasson et al. introduced STARKs [2]: arguments whose sole cryptographic assumption is a collision-resistant hash, constructed from an algebraic intermediate representation (AIR) of the computation, a Reed–Solomon low-degree extension of the trace and the FRI proximity test. FRI soundness in the list-decoding regime rests on the proximity-gaps theorem [3]; the tighter figures in common use rest on the ethSTARK conjecture, which is identified as such wherever it is invoked below. The Fiat–Shamir transformation is secure in the quantum random-oracle model [4]. Lookup arguments based on logarithmic derivatives (LogUp [28]) permit separate trace tables to exchange facts over a bus; the machine described here is constructed entirely from such buses. The SNARK line, Groth16 [23], PLONK [24], Halo [25], Spartan [26] and HyperPlonk [27], yields smaller proofs at the cost of a trusted setup or elliptic-curve assumptions and is not post-quantum; it is not employed. The node depends on Plonky3 [31] version 0.7.0 for the batch STARK, the Goldilocks field $p = 2^{64} - 2^{32} + 1$, Poseidon2 [32], a hiding FRI polynomial commitment and LogUp.

2.3 Zero-knowledge virtual machines

A zkVM proves the execution of a program for a general instruction set. RISC Zero [33], SP1 [34], OpenVM [36] and Jolt [29] target RISC-V; Cairo/Stwo [30] and Miden [35] target purpose-built instruction sets. All incur a generality overhead relative to bespoke circuits, and all reduce deployment to a data operation. The zkEVM rollups [37, 38] and the Ethereum Foundation’s real-time proving programme [40, 39] have demonstrated proving of general execution at block cadence on commodity GPUs. Randprotocol’s zkVM belongs to this family, with two distinguishing properties: the shielded transfer is a guest program rather than a dedicated circuit, and the STARK is verified natively by every validator without a pairing-based wrapper, so that the verification path remains post-quantum and setup-free.

2.4 Privacy-preserving ledgers

Zcash [16] established the note-commitment and nullifier model adopted here: a balance is a set of unspent notes, each a hiding commitment in an append-only Merkle tree, spent by

publication of a nullifier computable only by the owner. Monero [17] conceals senders by ring signatures. Zexe [18] generalised notes to arbitrary programs; Aztec [19, 20] executes private functions client-side; Miden [35] proves client-side with a general machine; Namada [21] operates a multi-asset shielded pool. Fully homomorphic designs [22] maintain encrypted state on chain at the cost of a decryption committee. Randprotocol’s pool follows the Sapling structure (a fixed four-in four-out shape — two-in two-out until chain 13 — dummy notes, nullifiers derived from a viewing key, sender and receiver envelopes) but is proved with a hash-based STARK and sealed with a lattice KEM, and is therefore post-quantum where Zcash and Monero are not.

2.5 Front-running

Daian et al. [13] quantified front-running on Ethereum and demonstrated its effect on consensus stability. Order-fairness [14] and threshold-encrypted mempools [15] constrain a block producer’s use of a visible transaction. Confidential execution addresses the problem at a lower layer: a shielded transfer discloses neither amount nor counterparty, leaving no signal to exploit. The residual disclosures, the fee and the transaction kind, are analysed in Section 5.6.

2.6 Post-quantum standards and bridges

The node signs with ML-DSA at the Dilithium2 parameter set [42] and seals envelopes with ML-KEM-768 [41]. Guardian bridges follow the Wormhole [43] attestation format; the surveys of Zamyatin et al. [44] and Lee et al. [45] attribute losses in this class to key compromise and verification defects. The bridge employs secp256k1 ECDSA because the source chains verify it natively; it is the sole component that is not post-quantum.

3 Architecture

3.1 Workspace structure

The full node is a Rust workspace (toolchain 1.98.1) of six crates (Table 1). The logic that must be bit-for-bit deterministic across validators, namely the ledger and consensus, is isolated from I/O and from the proof system so that it can be simulated in-process.

Crate	Contents
<code>randprotocol-core</code>	Pure logic without I/O or Plonky3: BLAKE3 and Dilithium2 (<code>crypto.rs</code>); notes, bundles, addresses and the commitment tree (<code>notes.rs</code>); transactions, actions, blocks, votes, certificates and validator sets (<code>types/</code>); fee floors (<code>gas.rs</code>); programs and receipts; the executor trait; the ledger and its admission rules with the staking, call-envelope, bridge-notes and aggregation modules (<code>ledger/</code>); genesis; the HotStuff state machine with epochs (<code>consensus/</code>); the bridge’s public state (<code>bridge/</code>).
<code>randprotocol-zkvm</code>	The Rand zkVM, vendored from the upstream research crate at constraint set 6: instruction set, assembler, guests (including <code>bundle</code>), emulator, the Poseidon2 sponge and Keccak/SHA-256 references, the eleven AIR tables — nine mandatory, the hash chips optional per proof (<code>tables/</code>), the EVM and sBPF interpreter guests, the Plonky3 configuration, prover and verifier (<code>machine.rs</code>), the note, viewing-key and call-envelope layers, and the chain-side verifier (<code>executor.rs</code>). The only crate linking Plonky3.
<code>randprotocol-rvm</code>	The recursion VM, vendored from the upstream <code>recursion</code> crate (milestones M5.1–M5.4): the field-native instruction set with the <code>REDUCE</code> and <code>SPONGE</code> precompiles, emulator, the eight-instance batch machine (<code>machine/</code>), the N -generic aggregate-verifier program, and the chain-facing <code>aggregate/verify_aggregate</code> interface. Shares one proof type with <code>randprotocol-zkvm</code> by the vendor script’s rename.
<code>randprotocol-node</code>	RocksDB storage, libp2p networking, the conflict-tracking mempool, block synchronisation, the redacted JSON-RPC server, the aggregate-verification executor, the event loop and the <code>rand-node</code> binary.
<code>randprotocol-client</code>	The <code>rand</code> wallet (key management, scanning, coin selection, proving, envelopes) and the RPC client library. Links the prover but neither storage nor networking.
<code>bridge-codec</code>	<code>no_std</code> , dependency-free wire format for guardian attestations.

Table 1: Crates of the reference full node.

3.2 Roles

A **validator** holds a Dilithium2 key with a row in the public register; it proposes in round-robin order, votes, and verifies every proof in every block. An **observer** runs the same binary without a key in the current epoch’s set. A **wallet** holds a 256-bit spend key from which every other key is derived; it scans the chain, proves bundles locally and signs nothing. A **guardian** is an external signer of bridge attestations. An **aggregator** bonds a registered payout address and proves one recursive STARK over a window of finalised bundles, submitting it for a later sealing block (Section 4.14); the role is permissionless, GPU-class, and never required for consensus, which runs on unaggregated blocks unchanged.

3.3 Transaction flow

Figure 1 traces a shielded transfer from the wallet to a committed block; subsequent sections elaborate each stage.

WALLET (randprotocol-client)	EVERY NODE (randprotocol-node)
-----	-----
1. scan: trial-decrypt every envelope with the viewing key; retain own notes	5. JSON-RPC rand_sendTransaction
2. select <= 2 notes; fetch the head anchor and a Merkle witness per input	v
3. prove the 'bundle' guest (tier 14, ~100 s): sk, notes, paths, amounts in; anchor, nf[2], cm[2], fee, burn, asset, time out, as one 8-word digest	6. Mempool: duplicate hash, nullifier or commitment already pending, pool full, then the full ledger probe (10 steps, the STARK last)
4. seal two envelopes (ML-KEM-768 + ChaCha20-Poly1305), submit Transaction{chain_id, bundle, action}	v
	7. HotStuff: leader proposes, validators re-apply the block (verify again), vote, QC, lock, 3-chain commit
	v
	8. one fsynced RocksDB WriteBatch: block, QC, 2 notes, 2 nullifiers, anchor, proposer's rewards, tree frontier

Figure 1: A shielded transfer from wallet to commit. The prover executes only in the wallet on the spender's notes; the verifier executes only within block application, on every node, on public data.

3.4 Disclosure summary

The following classification is referred to throughout. Public on chain: commitments, nullifiers, anchors, fees, burns, the asset index, the time word, envelope ciphertexts, proofs, program code, call receipts, the validator register with stakes and rewards, and every deposit amount at the pool boundary (faucet mint, genesis allocation, validator withdrawal, bridge deposit). Not disclosed: the sender and recipient of a transaction, the value of any note, the correspondence between a nullifier and a leaf, whether an input of a bundle is a dummy, and the private inputs of a call. Two disclosures are structural: fee floors differ by action, so the fee identifies the transaction kind; and the wallet of this revision requests a Merkle witness by leaf index, which identifies to the queried node the leaf about to be spent (closed in v0.5.4: the wallet keeps its own commitment tree, built from the commitment pages it already scans, and computes witnesses locally).

4 Implementation

This section describes the implementation in dependency order: the pure core, the zkVM, the node that supplies I/O, and the wallet and bridge codec. Where a rule is load-bearing the source is quoted, lightly abridged, and the file is identified.

4.1 Cryptographic primitives and identities

crypto.rs defines the base primitives. A **Hash** is a 32-byte BLAKE3 digest; every hash in the protocol is domain-separated by a fixed prefix, so that transaction identifiers, block hashes and state leaves cannot collide by construction. A **Keypair** is a Dilithium2 key pair derived deterministically from a 32-byte seed (public key 1,312 bytes, signature 2,420 bytes); the seed is zeroised on drop. A validator **Address** is `blake3("rand-address" || pk)` rendered in base58. Transaction roots and state roots use a pairwise BLAKE3 Merkle tree with duplication of an odd final leaf.

A wallet identity is a distinct object, defined in the zkVM crate (**notes.rs**, **viewing.rs**) because each derivation is a Poseidon2 hash recomputed in circuit by the **bundle** guest. From

a single 256-bit SpendKey sk:

$$\begin{aligned} \text{nk} &= H(\text{NK}, \text{sk}), & \text{pk} &= H(\text{PK}, \text{nk}), & \text{nf}(\text{cm}) &= H(\text{NF}, \text{nk}, \text{cm}), \\ \text{ovk} &= H^*(\text{OVK}, \text{nk}), & (\text{dk}, \text{ek}) &= \text{KeyGen}_{\text{ML-KEM}}(H^*(\text{KEMSEED}, \text{nk})), \end{aligned}$$

where $H(d, m)$ denotes the Poseidon2 sponge of Section 4.10 applied to $[d, m]$ and H^* a host-side wide hash. Here nk is the viewing key, which permits observation but not spending; pk is the owner field of a note; nf(cm) is the nullifier of the note with commitment cm; ovk opens envelopes sent by the wallet and dk those sent to it. A *shielded address* is pk (32 bytes) concatenated with the 1,184-byte ML-KEM-768 encapsulation key, encoded in base58 with the prefix **rand1** (approximately 1,668 characters). The one-way derivations are what separate viewing from spending: a holder of nk can compute every address, nullifier and decryption key of the party but cannot satisfy the guest, which reads sk as a private input and derives nk internally. A wallet key never signs a transaction.

4.2 Notes, bundles, transactions and actions

Notes. A note comprises 28 machine words (112 bytes): $\text{note} = (\text{pk}, \text{from}, \text{amount}, \text{asset}, \text{time}, r)$, with pk, from, r of eight words each, a 64-bit amount in units of 10^{-9} RAND, a 32-bit asset index (0 for RAND, otherwise the bridge registry’s dense index) and a 32-bit creation time. Its commitment is $\text{cm} = H(\text{CM}, \text{note})$. The note plaintext is never published; it is conveyed in an **Envelope** {kem_ct, to_receiver, to_sender, body}: a fresh 32-byte per-transaction key encrypts the note under ChaCha20-Poly1305 with cm as associated data, and that key is wrapped once under the ML-KEM-768 shared secret encapsulated to the receiver’s address and once under the sender’s ovk. An envelope occupies 1,348 bytes and is bound to its commitment by the AEAD, so it cannot be transplanted to another transaction and an incorrect key fails authentication rather than yielding plaintext.

Bundles. Each value-moving transaction carries one bundle of fixed shape: two inputs and two outputs at revision 26ab1d4 (Listing 1), four and four since chain 14. *Addendum (chain 14, v0.5)*: the shape is four inputs and four outputs, slots 0–1 in a private asset and slots 2–3 in RAND, with two burn fields (burn_a, burn_r) and a burn_asset word; the two-bundle transaction is gone, so a bridge burn, a bond and a token transfer each carry this one bundle (Section 4.9).

```

1 pub struct Bundle {
2     pub anchor: Word8,                // a recent block-end tree root
3     pub nullifiers: [Word8; 2],       // of the two spent notes (a dummy also publishes one)
4     pub commitments: [Word8; 2],     // of the two created notes (change is always published)
5     pub fee: u64,                    // public, always in RAND, credited to the proposer
6     pub burn: u64,                   // value leaving the pool into the action (0 for a
    transfer)
7     pub asset: u32,                  // the single asset this bundle balances; 0 = RAND
8     pub time: u32,                   // the height targeted by the sender; copied into both
    outputs
9     pub envelopes: [Envelope; 2],    // envelopes[i] sealed against commitments[i]
10    pub proof: Vec<u8>,               // postcard(rand_zkum::Proof) of the 'bundle' guest
11 }

```

Listing 1: notes.rs: the bundle; every field is public chain data.

The fixed shape implies that one input and one output are frequently dummies: a dummy input is a zero-value note owned by the spender, and the change output is published even when zero. A transaction of variable shape would disclose the absence of change.

Transactions and actions. A transaction consists of a chain identifier, an optional bundle and an action (Listing 2).

```

1 pub struct Transaction { pub chain_id: u64, pub bundle: Option<Bundle>, pub action: Action }
2 pub enum Action {
3     None, // tag 0: plain transfer
4     Mint { cm, envelope, amount: u64, minter: PublicKey, signature }, // tag 1: faucet, no
        bundle
5     Deploy { base_pc: u32, words: Vec<u32> }, // tag 2
6     Call { program: ProgramId, proof: Vec<u8>, input_envelope: Option<CallEnvelope> }, // tag 3
7     Bond { validator: Address, amount: u64, registration: Option<Registration> }, // tag 4
        (S2)
8     Unbond { validator, amount, nonce, signature }, // tag 5 (S2), no bundle
9     Withdraw { validator, amount, nonce, time, r: Word8, envelope, signature }, // tag 6
        (S2), no bundle
10    BridgeAttest { attestation: Vec<u8>, recipient: ShieldedAddress, r, time, asset, envelope
        }, // tag 7 (S3)
11    BridgeBurn { asset_bundle: Bundle, asset, amount, relayer_fee, to_chain, to }, // tag 8
        (S3)
12 }

```

Listing 2: `types/transaction.rs`: the transaction and its nine actions.

No signature covers the transaction and no sender field exists. The identifier is the domain-separated BLAKE3 hash of the encoding in which the bundle proof is replaced by its digest, $\text{blake3}(\text{"rand-txid-2"} \parallel \text{bincode}(\text{tx}_{\text{digest}}))$, so that a transaction hashes to the same identifier whether it carries its proof or, once its block is sealed, only the proof’s digest (the pre-v0.1 review’s M1, Section 4.14; a hard fork on every identifier, taken at the chain-9 cut); replay protection is provided by the nullifier set. Exactly three actions are carried without a bundle, because their signer owns no notes from which to pay: a faucet `Mint`, signed by a validator key over the chain identifier, the new commitment, its envelope and the public amount; and the validator-signed staking actions `Unbond` and `Withdraw`, replay-protected by the register’s nonce. Every other action requires a bundle, which pays the action’s fee floor from the spender’s notes. A `Deploy` or `Call` is thus an ordinary shielded transaction whose bundle is a zero-value self-transfer; the chain does not learn whose notes funded it.

Blocks and certificates. A `Vote` is a Dilithium2 signature over the domain-separated BLAKE3 hash of the view and block hash; a `QuorumCertificate` is a view, a hash and a set of votes whose signers’ stake satisfies $3 \sum w > 2W$ in the set of the block’s epoch; a `Block` is a signed header (height, view, parent, proposer, timestamp, transaction root, state root, the parent’s QC) with its transactions.

4.3 Fee floors, programs and the executor interface

`gas.rs` fixes the caps and floors. A program is limited to 4,096 words; a proof to 2 MiB (raised from 1 MiB because the smallest production proof at 80 queries exceeded the former cap); an attestation to 16 KiB; a note envelope to 2,048 bytes and a call envelope to 18,432; a block to 2,000 transactions and 4 MiB. The floor is a function of the action:

$$\text{fee_floor}(a) = \begin{cases} 0 & a \text{ bundle-less (Mint, the register actions, Aggregate)} \\ B & a \in \{\text{None, Bond, BridgeAttest, RegisterAggregator}\} \\ B + 100,000 \cdot |\text{words}| & a = \text{Deploy} \\ B + \text{call_fee}(t) & a = \text{Call at tier } t \\ 2B & a = \text{BridgeBurn (two bundles)} \end{cases} \quad (1)$$

with bundle base $B = 10^6$ units = 0.001 RAND and $\text{call_fee}(t) = 10^6 + 10^5 \lfloor (t - 10)/2 \rfloor$. Any excess over the floor is a tip. On a chain without the aggregation section the entire fee is credited to the proposer’s **rewards** field in the validator register; with it, the proposer keeps exactly B of each bundle’s fee and the excess accrues to an **unsealed_fees** bucket, paid to the aggregate that covers the bundle, or swept to the including proposer at the sealing window’s expiry (Section 4.14). There is no fee burn and no per-instruction metering: no node executes a program, a node’s work per transaction is one STARK verification of near-constant cost, and termination is enforced by the tier rather than by a gas counter.

program.rs defines the content address

$$\text{program_id} = \text{blake3}(\text{"rand-program"} \parallel \text{base_pc} \parallel \text{words})$$

and the on-chain **ProgramRecord**, whose **code_hash** field holds the zkVM’s in-circuit digest hc ; no **deployer** field exists, since a deployment identifies no party. A receipt records the transaction hash, program identifier, tier, the eight outputs, the height and index, and the proof’s H_{IN} ; there is no effect field, because outputs are published data from which no state transition follows.

confidential.rs defines the interface between the ledger and the proof system. In addition to **check_program**, **verify_call** and **warm**, the trait exposes **verify_bundle** against the pinned bundle digest, the bundle-digest recomputation, and the Poseidon2 evaluations required for commitments and tree nodes, so that **randprotocol-core** does not depend on the field-arithmetic crates. **StubExecutor** substitutes a trivial implementation for the consensus simulation; **DisabledExecutor** rejects deployments and calls on a chain configured with **confidential: false**.

4.4 The ledger

ledger/mod.rs implements the state-transition function. The state consists of

- the commitment tree: depth 32, node hash $H(\text{NODE}, \text{left}, \text{right})$, append-only, maintained as a *frontier* ($O(32)$ state and $O(32)$ per append), so that consensus state never materialises the leaf set;
- the set of all commitments ever appended (the frontier cannot decide membership) and the set of all nullifiers ever spent;
- the 256 most recent block-end roots (the *anchors*);
- the validator register $\text{Address} \rightarrow \{\text{public_key}, \text{stake}, \text{pending}, \text{rewards}, \text{payout}, \text{nonce}\}$;
- the aggregator register $\text{Address} \rightarrow \{\text{public_key}, \text{bond}, \text{payout}, \text{nonce}, \text{unbonding}\}$, present only on a chain whose genesis carries the aggregation section (Section 4.14);
- the program map, an optional bridge state, and the supply counters (excluded from the state root);

together with the transient height, timestamp and the deposits created by the block under application.

Admission. **validate_inner** executes the ten steps of the design in order of increasing cost and returns the results of the expensive checks so that application verifies each proof exactly once:

1. size caps on every variable-length field, prior to any parsing;

2. chain identifier;
3. shape and floor: a bundle-less action must carry no bundle and every other action must; the transaction's own bundle is denominated in RAND and burns nothing, except that a Bond's bundle must burn exactly the bonded amount; $\text{fee} \geq \text{fee_floor}(a)$;
4. the anchor is one of the 256 most recent block-end roots;
5. $\text{time} \in [\text{height} - 256, \text{height}]$, for the bundle and for a bundle-less withdrawal's time word;
6. the two nullifiers are distinct and unspent; the two commitments are distinct and absent from the tree;
7. action-specific inexpensive checks: faucet enabled, mint within the 100 RAND cap, minter a validator with a valid signature; program words decode (deploy); program exists and input envelope within its cap (call); the staking rules; the bridge rules, including a burn's second bundle in full;
8. the bundle digest, recomputed by the ledger from the published plaintext with the taint word fixed to zero, equals the proof's eight output words;
9. the bundle proof verifies against the genesis-pinned $\text{hc}_{\text{bundle}}$;
10. for a call, its own proof against the program's `code_hash`, followed by $\text{fee} \geq B + \text{call_fee}(t)$, which is decidable only once a verified proof has disclosed the tier.

```

1 // 8-9. the bundle's digest, then its proof
2 if let Some(b) = bundle { self.check_bundle_proof(b, executor)?; }
3 // 10. the call's own proof, then its tier-dependent fee
4 if let (Some(record), Action::Call { proof, .. }) = (call_record, &tx.action) {
5     let outcome = executor.verify_call(record, proof).map_err(TxError::InvalidProof)?;
6     let min = gas::BUNDLE_BASE + gas::call_fee(outcome.tier);
7     if tx.fee() < min { return Err(TxError::FeeTooLow { min, fee: tx.fee() }); }
8     verified.call = Some(outcome);
9 }

```

Listing 3: `ledger/mod.rs`: the final admission steps, at which the proofs are verified.

Application. `apply_tx` appends the bundle's two commitments, inserts its two nullifiers, credits the fee to the proposer's register row, and then applies the action: a mint appends its note; a deployment inserts a `ProgramRecord`; a call records a receipt; a bond adds the burned amount to the validator's stake; an unbond moves stake into the pending queue; a withdrawal derives a deposit note and appends it; a bridge attestation derives a deposit note in the bridged asset; a bridge burn appends the asset bundle's outputs and an outbound message. The commitments of a withdrawal's and an attestation's deposit are deliberately absent from the wire: the ledger derives them from the published blinding r , the public amount and the recipient it already holds, so that a validator cannot declare one amount and mint a note of another. `apply_block` enforces the size caps as a consensus rule, verifies the proposer signature and transaction root, applies to a scratch copy, records the block-end anchor, and commits only if the resulting root equals the header's.

State root.

$$\text{state_root} = \text{blake3}(\text{"rand-state-2"} \parallel \text{tree_root} \parallel R_{\text{nf}} \parallel R_{\text{val}} \parallel R_{\text{prog}} [\parallel R_{\text{bridge}}]), \quad (2)$$

where `tree_root` is the Poseidon2 root of the commitment tree (the value a wallet anchors a proof to, so that the pool's contents are committed without being readable), R_{nf} is a BLAKE3 Merkle root over the sorted nullifiers (recomputed per block in $O(n)$, adequate below approximately 10^6 entries), R_{val} a root over one leaf per register row including the length-prefixed unbonding queue and the 1,216-byte payout address, R_{prog} a root over program identifiers, and R_{bridge} present only on a bridged chain. On a chain whose genesis carries the aggregation section, the domain is **rand-state-3** and the root gains a component R_{aggr} over the aggregator register (one leaf per entry: address, bond, nonce, unbonding, payout keys); a chain without the section keeps equation (2) byte-for-byte. The domain suffix **-2** marks the break from the account-based ledger. *Addendum (chain 14)*: the domain is **rand-state-4** and the root gains a component over the token registry (name, symbol, decimals, authority, supply and, for a bridged token, its backings with their counters); **rand-state-5**, genesis-gated for the next cut, adds the faucet's per-epoch budget.

4.5 Staking and the supply audit

`ledger/staking.rs` governs the single public table. The constants are `MIN_STAKE = 1,000 RAND`, `MAX_VALIDATORS = 100`, `UNBONDING_EPOCHS = 2`, and a genesis-selected `epoch_blocks` (1,000 on the test network). The set for epoch $e \geq 1$ is a pure function of the register as of the final block of epoch $e - 1$ (Listing 4).

```

1 pub fn derive_set(register: &BTreeMap<Address, ValidatorEntry>) -> ValidatorSet {
2     let mut eligible: Vec<_> = register.iter().filter(|(_, e)| e.stake >= MIN_STAKE).collect();
3     // Descending stake, then ascending address: the cap must not depend on map order.
4     eligible.sort_by(|(a_addr, a), (b_addr, b)| b.stake.cmp(&a.stake).then_with(|a_addr, b_addr| a_addr.cmp(b_addr)));
5     eligible.truncate(MAX_VALIDATORS);
6     ValidatorSet::from_entries(eligible.into_iter().map(|(_, e)| (&e.public_key, e.stake)))
7 }

```

Listing 4: `ledger/staking.rs`: the set-derivation rule.

A **Bond** is a bundle whose burn equals the bonded amount, so that stake leaves the pool rather than becoming a note; a registration (public key, payout address, signature over the chain identifier and payout) accompanies a validator's first bond. An **Unbond**, signed by the validator key with the register's nonce, moves stake into **pending** with release epoch $e + 2$; the validator leaves the set at the next boundary. A **Withdraw** pays released pending amounts and accrued rewards into a note derived by the chain as $H(\text{CM}, \text{payout.pk}, 0, \text{amount} - B, 0, \text{time}, r)$; the bundle base B is paid to the proposer from the withdrawn amount. Rewards are the fees of blocks proposed; there is no inflation and no slashing.

`ledger/supply.rs` maintains the monotonic counters outside the state root: genesis deposits, genesis stakes, faucet mints, withdrawal deposits, fees paid and burns — and, on a chain with the aggregation section, four more: subsidies paid, sealed blocks, aggregator bonds and slashed amounts. The pool cannot sum its own notes, but every crossing of its boundary is public, and the identity

$$\begin{aligned}
 \text{pool_value} &= D_{\text{gen}} + M_{\text{faucet}} + D_{\text{withdraw}} + D_{\text{vest}} - F - U, \\
 \text{register_total} &= \sum_v (\text{stake}_v + \text{pending}_v + \text{rewards}_v), \\
 V_{\text{reg}} &= \sum_e (\text{amount}_e - \text{claimed}_e - \text{revoked}_e - \text{bonded}_e), \quad (3) \\
 \text{issued} &= D_{\text{gen}} + S_{\text{gen}} + V_{\text{gen}} + M_{\text{faucet}} [+ \text{subsidised}], \\
 \text{pool_value} + \text{register_total} + V_{\text{reg}} &= \text{issued} - \text{slashed} - F_{\text{reg}}
 \end{aligned}$$

is an invariant that a node verifies by full replay and reports through the supply RPC method, which exposes the new counters separately. Here U counts every bundle burn — a bond's, which

re-enters as stake, a registration fee's, and a bridge or token burn's — while only two terms destroy issuance and stand on the right: F_{reg} , the native-token registration fees burned under the genesis `burn_registration_fee` switch (v0.5.5, audit v5 TOK-2), and slashed, the aggregator bonds slashed for equivocation, which reads zero on every chain and is retired since v0.6 (Section 4.14). The vesting terms exist since v0.5.11 (genesis *vesting*: timelocked allocations to team, investors and founding partners, a third holder beside the pool and the validator register): V_{gen} is the entries' amounts at genesis, issuance beside S_{gen} ; V_{reg} (the node's `vesting_in_register`) is what the register still holds; D_{vest} is the notes that claims and revocations created, net of their bases; a claim moves value from the vesting register into the pool, a vested bond moves it into a validator's stake and back on unbond, and neither creates or destroys any. The Lean inventory of September 2026 found an earlier statement of this identity given twice, once as issued and once as issued — slashed, and without the registration-fee and vesting terms; the form above is the one the node checks (`docs/supply.md`). Subsidy issuance is schedule-bound rather than proposer-bound: the aggregate's deposit is minted at the schedule's current index, the proposer-kept fee floor counts at inclusion and an expired excess at its sweep, and the bucket itself is not part of `register_total`. The two genesis terms are accepted from the genesis file; the correspondence between a genesis note's sealed value and its declared amount is not proved.

4.6 Genesis

A genesis file specifies the chain identifier, timestamp, one register entry per validator (key, stake of at least `MIN_STAKE`, payout address), the epoch length, an `alloc` list of deposit notes `{cm, envelope, amount}`, the faucet and confidential flags, the FRI profile, the pinned `hc_bundle` and optional bridge and aggregation sections. All fields enter the genesis hash; a node whose bundle guest does not digest to the pinned value refuses to start. The aggregation section carries the bond, the cover cap, the subsidy base and halving interval, the sealing window, and the admitted shapes (each a FRI profile, tier, six declared heights, the guest's `hc` and the aggregate program digest); a zero program digest is refused at validation, so the placeholder a cut script ships cannot reach a fleet (Section 4.14). Genesis notes carry fresh commitment randomness, so the file is generated once and distributed unchanged.

4.7 Chained HotStuff with epochs

`consensus/hotstuff.rs` is a pure state machine that consumes messages and timeouts and emits actions (broadcast, persist, commit, fetch, arm a timer). The rules are those of chained HotStuff, stated here as implemented.

Proposals. `on_proposal` accepts a block only if its view lies within 10^6 of the current view, its height exceeds the committed height, its proposer is the leader of its view in the set of its epoch, its signature verifies, its justify QC certifies its parent and verifies against the set of the parent's epoch, its view exceeds the parent's, its height equals the parent's plus one, and, on a bridged chain, its timestamp neither precedes the parent's nor exceeds it by more than 60 s (`MAX_TIMESTAMP_STEP_MS`, a validity rule that replay applies too). The block is then applied on a copy of the parent's ledger (the point at which every proof is verified), inserted in a bounded tree (512 entries); `high_qc` is raised, the block's view is entered, the lock-and-commit rule is evaluated, and a vote is cast if the block is safe — on a bridged chain, only if its timestamp is also at most 15 s ahead of the replica's own clock (`MAX_CLOCK_DRIFT_MS`, a vote rule only: the block stays in the tree, and replay, which reads no clock, never applies it).

Epochs. $\text{epoch}(h) = \lfloor h / \text{epoch_blocks} \rfloor$. The set for a block at height h with parent p is derived from the register after the final block of epoch $\text{epoch}(h) - 1$ on the branch ending at p , cached per (epoch, start block), and persisted when the epoch's first block commits so that

replay and synchronisation verify historical certificates without historical registers. An empty derivation carries the previous set forward. A validator whose key is absent from the current set observes until admitted, so a validator bonding after genesis joins without restart.

Voting. A validator votes at most once per view and only for a block in its current view whose justify QC is newer than its lock or which extends the locked block. The safety state (view, high QC, locked QC, last voted view) is fsynced before the vote is transmitted.

Certificates. Every validator collects votes and assembles QCs locally; a QC forms when the voters' stake, weighted in the set of the certified block's epoch, exceeds two thirds.

Lock and commit. On each new block b^* , with b'' certified by b^* 's justify, b' by b'' 's and b by b' 's, the lock advances to the QC on b' when newer, and b commits together with its uncommitted ancestors if and only if the three certifying QCs occupy consecutive views and the path reaches the committed head (Listing 5).

```

1 let qc_b = b1_justify.view;           // certifies b
2 let qc_b1 = b2_justify.view;         // certifies b'
3 let qc_b2 = b_star.header.justify.view; // certifies b''
4 if qc_b1 != qc_b.saturating_add(1) || qc_b2 != qc_b1.saturating_add(1) { return; }

```

Listing 5: consensus/hotstuff.rs: the three-consecutive-view commit rule.

Pacemaker and proposal construction. Timeouts double on consecutive expiry up to a cap and reset on progress; a NewView from a validator in a higher view advances a lagging replica. The leader proposes once it holds a QC for the previous view or a NewView quorum, paced to a block interval (1 s on the fleet; the integrated branch's cluster tests use 3 s so that a bundle proof does not outlive its 256-block anchor on a contended machine). Candidates are applied individually on a trial copy and omitted on failure.

Resumption. On restart the replica is reconstructed from the committed head and the persisted epoch sets, with both `high_qc` and the lock set to the head QC. The persisted lock is therefore not restored, whereas the vote-once-per-view state is. Section 5.5 treats this as a hypothesis. *Addendum (v0.5.1–v0.5.5):* the lock is restored ahead of the head QC and never lowered below it (v0.5.1); `extends_locked` withholds the vote and fetches an unknown locked block; the lock is lowered only on signed `NotHeld` attestations (Dilithium2 over `rand-not-held-1`, the genesis hash and the block hash) from strictly more than two thirds of the current set's stake (v0.5.4 introduced the attestation, v0.5.5 raised the threshold from a third to a quorum); every certified block above the head is persisted and re-executed at resume, a persisted high QC whose block is not among them is dropped as a ghost, and a fallback lands on the highest certified block held (v0.5.5). Proposals: one block per (view, leader), a block more than eight views ahead refused before its signature is checked, off-branch eviction when the tree is full (v0.5.4); on a bridged chain the drift bound is 15 s and a block may not exceed its parent's timestamp by more than 60 s (a validity rule, replayed).

4.8 Instruction set, syscalls and emulator

The machine implements a subset of RV32I with the M extension: LUI AUIPC JAL JALR, the six branches, loads and stores of byte, halfword and word width, the twelve immediate and register ALU operations, the eight multiply and divide instructions, and ECALL. FENCE, CSR instructions and EBREAK are absent. Each instruction decodes into 23 small integer fields, which

constitute the message fetched by the CPU table from the program table. Seven syscalls form the entire external interface (Table 2).

Syscall	No.	Semantics
HALT	0	Terminate; the eight output slots are final.
WRITE_OUTPUT	1	Set public output slot a0 to a1 ; at most one write per slot; unwritten slots are zero.
READ_INPUT	2	Return private input word a0 from the committed input table; repeated reads of one index agree; an index $\geq n$ is unsatisfiable.
POSEIDON2	3	Hash the $\mathbf{a1} \leq 4096$ words at word address a0 in place (rate-4 sponge), overwriting the first eight words with the digest.
KECCAK	4	One Keccak-f[1600] permutation of the 50-word state at word address a0 , in place; the keccak table performs the memory traffic (constraint set 5).
SHA256	5	One SHA-256 compression of the 16-word block at word address a0 , in place; the sha256 table performs the memory traffic (M4.4, optional per proof on the keccak table’s exact terms).
READ_PUBLIC	6	Return public input word a0 from the public segment, a second, independently indexed input vector bound to the unsalted $H_{\text{PUB}} = \text{pv}[\text{PUB}_{0..7}]$ (constraint set 6).

Table 2: The syscall interface.

Note commitments, nullifiers and Merkle verification are not syscalls but assembler routines that stage a domain-tagged message in memory and invoke POSEIDON2; the Merkle walk is a counted loop, so program size is independent of tree depth. `emulator.rs` provides the reference semantics and records one `CycleEvent` per CPU row; it executes up to the largest tier’s budget, rejects Poseidon2 and Keccak pointers at or above 2^{30} in agreement with the AIR, and enforces the 65,535-word limit on programs and input vectors imposed by the 16-bit counter of the digest rows.

4.9 The bundle guest

`guests::bundle` is the program against which every shielded transaction is proved: 3,811 hand-assembled words, with digest `hcbundle` pinned at genesis. Its private-input vector comprises 612 words: the spend key; for each of two inputs the note’s `from`, amount, asset, time, r , a 32-level Merkle path and a leaf index; the anchor; for each of two outputs the recipient `pk`, amount and r ; and the bundle’s public fee, burn, asset and time. It establishes, in order: `nk` and `pkself` from `sk`; for each input of non-zero amount, the note’s commitment (computed with `pk = pkself`, so that a note not owned by the spender yields a commitment absent from every tree) and its Merkle path to the anchor; both nullifiers, dummy or otherwise; both output commitments with `from = pkself` and the bundle’s asset and time; every amount below 2^{63} ; the 64-bit balance $a_1 + a_2 = a'_1 + a'_2 + \text{fee} + \text{burn}$ with explicit carries; agreement of each input’s asset with the bundle’s; and $\text{nf}_1 \neq \text{nf}_2$, $\text{cm}_1 \neq \text{cm}_2$.

Addendum (chain 14, v0.5): the hidden-asset guest. Chain 14 pins `guests::bundle_hidden` instead, a four-slot guest whose asset A is a private input: slots 0–1 carry A (RAND or any registered token) and slots 2–3 carry RAND, and the digest names no asset, so a token transfer publishes exactly what a RAND payment does. It establishes, in the same two ways, the relations of the node’s soundness table: structurally, the key derivation, the staging of every note with its slot’s asset and the bundle’s time, every nullifier (dummies included), the dummy skip that carries no value, the mask $\text{burn_asset} = A \cdot [\text{burn}_A \neq 0]$ and the digest

layout; by taint, membership of each non-dummy input under the anchor, each input’s asset against its slot’s, the six nullifier and six commitment pair inequalities, eleven range checks below 2^{63} , every carry, and the two conservation equalities $a_0 + a_1 = a'_0 + a'_1 + \text{burn}_A$ and $a_2 + a_3 = a'_2 + a'_3 + \text{fee} + \text{burn}_R$. The proof’s public input segment is no longer empty: it carries the *transaction binding* $\text{blake3}(\text{"rand-tx-bind-1"} \parallel \text{bincode}(\text{chain_id}, \text{bundle}', \text{action}'))$, the transaction with every bundle-proof byte string blanked, as eight words, so a copied proof cannot ride a changed destination, validator, memo or envelope. The proof is pinned to tier 14 with neither hash chip, and still proves in about 100 s. A twelve-variant weakened-guest harness and a 25,000-witness mutation fuzz against an independent host model back the table.

The instruction set has no assertion primitive, so a violated relation cannot abort execution. Instead each violation is disjoined into a register `BAD`, and the guest’s single output is

$$\text{digest} = H(\text{BUNDLE}, \text{anchor}, \text{nf}_1, \text{nf}_2, \text{cm}_1, \text{cm}_2, \text{fee}, \text{burn}, \text{asset}, \text{time}, \text{bad}),$$

47 words following the domain tag. The ledger recomputes the digest with $\text{bad} = 0$. A dishonest witness therefore yields a digest the ledger cannot reproduce, and the transaction is rejected at admission step 8. The taint word is placed on a field the verifier fixes, never on a sender-supplied field; an earlier draft folded it into `time`, which a sender could have compensated by publishing the corrupted value.

```

1 a.extend(li(T0, domain::BUNDLE as i32)); a.push(sw(BASE, T0, BUF));
2 copy_word8(&mut a, BASE, T0, ANCHOR, BUF + 4);
3 copy_word8(&mut a, BASE, T0, NF1, BUF + 36); copy_word8(&mut a, BASE, T0, NF2, BUF + 68);
4 copy_word8(&mut a, BASE, T0, CM_OUT1, BUF + 100); copy_word8(&mut a, BASE, T0, CM_OUT2, BUF +
  132);
5 /* fee_lo, fee_hi, burn_lo, burn_hi, asset, time at BUF+164 .. BUF+184 */
6 a.push(sw(BASE, BAD, BUF + 188));
7 a.extend(call_poseidon2(ptr_words(BUF), 48)); // domain(1) + msg(47)
8 for i in 0..8 { a.push(lw(T1, BASE, BUF + 4 * i)); a.extend(write_output(i, T1)); }
```

Listing 6: `guests.rs`: the bundle guest’s digest, with the taint as an explicit final word.

A dummy input bypasses its Merkle check, and the bypass cannot be exploited: its condition reads the same amount registers that contribute to the balance sum, so a non-zero amount cannot both contribute value and take the bypass. The guest executes approximately 9,160 cycles including 953 program-digest rows and 154 input-digest rows, which requires tier 14.

4.10 The sponge and the two digests

The permutation is Plonky3’s `Poseidon2Goldilocks<8>`: width 8, S-box x^7 , 4 initial full rounds, 22 partial rounds, 4 terminal full rounds. The sponge has rate 4 and capacity 4 in overwrite mode without padding; a digest is four Goldilocks elements split into eight 32-bit words. Round constants are drawn from a fixed development seed (`PERM_SEED = "RandZK"`); adoption of production constants is a pending, circuit-preserving change that constitutes a hard fork. *Addendum (chain 15)*: since audit ZKV-2 (`f3b8bea`) the 86 constants are committed as a literal table (`poseidon2_constants.rs`) — the same ChaCha12 draw under the same seed, printed once, so that no hash output moved and no future `rand` or `chacha20` upgrade can move one; they remain a seeded draw rather than a published instantiation. The *program digest* `hc` seeds the capacity lanes with $(\text{HC}, \text{base_pc}, \text{len})$ and absorbs four words per permutation; the *input digest* H_{IN} seeds the capacity with $(\text{IN}, n, 0)$, absorbs four fresh salt words, then the inputs. The salt is drawn from operating-system entropy per proof and leaves the prover only within a call envelope the caller elects to publish.

No length binding. Overwrite mode without padding means the sponge does not bind its message length: the all-zero initial state and the overwrite of the rate lanes make a message of

at most four words and its zero-extension within the first block collide, $\text{sponge_hash}(1, 2, 3) = \text{sponge_hash}(1, 2, 3, 0)$ (audit ZKH-3; the Lean development proves both the collision and that the function is not collision-free on all words). Collision resistance is therefore an assumption only on domain-separated messages of *one fixed length per domain*, and every hypothesis of Section 5.2 that invokes it is to be read that way. The node meets the hypothesis by construction: each domain tag of the note layer is used at exactly one message length (28 words under CM, nine under NF, and so on), and hc , H_{IN} and H_{PUB} seed the message length into a capacity lane before absorbing a word, so no two lengths share a starting state. Constraint set 7 adds POSEIDON2_LEN, the same sponge with the length in capacity lane 4, for variable-length data (HCS-4). Any future caller with variable-length input must encode the length itself.

4.11 Trace tables and buses

The machine is a batch of AIRs proved jointly, exchanging facts over LogUp buses whose balance the batch verifier checks. Nine tables are always present — the ninth, **public**, since constraint set 6 — and the hash chips **keccak** and **sha256** are optional per proof on identical, independent terms (a zero declared height removes the instance entirely), so a guest pays for the hash it actually calls (Table 3).

Table	Height	Role
program	2^{h_P} , declared, $h_P \in [4, 22]$	One row per instruction word with an in-circuit decoder; VALID and two multiplicities.
cpu	2^t	One row per cycle, preceded by $\lceil \ell/4 \rceil$ program-digest rows, $1 + \lceil n/4 \rceil$ input-digest rows and the public-segment digest rows. The only table with public values (34 since constraint set 6: the 26 of set 5 and H_{PUB}).
memory	2^{h_M} , declared, $h_M \in [t + 2, 24]$	Registers and RAM in one table sorted by (space, address, timestamp); read-after-write consistency is a transition constraint. Declared because a Keccak row performs 100 accesses.
alu	2^{t+1}	Every arithmetic, logical, shift, comparison, multiplication and division result over four byte limbs.
range, nibble	256 each	Preprocessed byte range, powers of two, 4-bit AND/OR/XOR.
poseidon2	2^{t+2}	One row per round in 32-row blocks (30 round rows, 2 idle).
input	2^{h_I} , declared, $h_I \in [2, 20]$	One row per committed private-input word, on two separate buses.
public	2^{h_U} , declared, mandatory	One row per public-segment word with its index, on the PUBLIC_DIGEST/PUBLIC_READ pair; an empty segment is four rows, height 2 (constraint set 6).
keccak	2^{h_K} , declared, $h_K \in [5, \min(t + 5, 20)]$ or absent	Keccak-f[1600], one row per round in 32-row blocks (24 rounds, 8 idle), 2,612 columns; performs its own memory traffic; every limb is derived from asserted bits, so no range lookups are required.
sha256	2^{h_S} , declared, or absent	SHA-256, one row per round in 64-row blocks, 466 + 10 columns; performs its own memory traffic (M4.4).

Table 3: The trace tables at tier t .

The buses are `PROGRAM` and `PROGRAM_WORD` (program to cpu; the latter with multiplicity constrained equal to `VALID`, which makes the digest a set-equality argument), `MEMORY` (a multiset equality with $ts = 4 \text{ clk} + \text{slot}$), `ALU`, `RANGE8`, `POW2`, `AND4/OR4/XOR4`, `POSEIDON2`, `INPUT_DIGEST` and `INPUT_READ` (separated so that the mandatory digest absorption and the reads cannot exchange budget), `PUBLIC_DIGEST` and `PUBLIC_READ` (the same separation for the unsalted public segment), `KECCAK` and `SHA256` — sixteen in all. Two invariants govern every row kind: each bus message column is constrained on every row kind that emits it, and each bus count is forced to zero wherever its message columns are unconstrained. The 111 tests in `tests/cheating.rs` each apply a specific tampering and assert rejection by the constraint checker, the lookup balance or the verifier.

The audit of 2026-09-12 identified and closed two critical defects in the CPU table’s hash row groups, both now pinned by regression tests: free-standing absorb and write-back rows lacked an entry gate, permitting a prover to splice write-back rows arbitrarily and obtain four unconstrained RAM writes per row at an unbounded pointer; and `HASH_FIN` was not confined to write-back rows. Both attack witnesses verified prior to the fix and are rejected after it. Each table’s maximum constraint degree is pinned by a test below the ceiling of 9 permitted by blowup 8 and the zero-knowledge flag (`cpu` and `alu` attain 8).

4.12 Machine configuration, proofs and verification

`machine.rs` configures Plonky3: Goldilocks with a degree-2 binomial extension as challenge field, a Poseidon2 duplex challenger, a hiding Merkle commitment and a hiding FRI PCS. The production profile was restored to the design parameters in constraint set 5 (Listing 7).

```

1 pub enum FriProfile {
2     /// 16 queries, 4 PoW bits - for 'cargo test'. Not a production target.
3     Test,
4     /// 80 queries, 20 PoW bits, blowup 8 - the whitepaper's parameter table. The M2.2 retune
5     to
6     /// 27 queries hit the ethSTARK *conjectured* 100-bit target but abandoned the *proven*
7     floor:
8     /// ~86 proven bits at q=80, ~42 at q=27. Reverted 2026-09-12 on the zk audit's finding ZM1
9     .
10    Production,
11 }
12
13 let fri = FriParameters { log_blowup: 3, log_final_poly_len: 0, max_log_arity: 3,
14     num_queries: profile.num_queries(), commit_proof_of_work_bits: 0,
15     query_proof_of_work_bits: profile.pow_bits(), mmcs: challenge_mmcs };

```

Listing 7: `machine.rs`: the FRI profiles following the 2026-09-12 audit.

Tiers and declared heights. A tier $t \in \{10, 12, 14, 16, 18, 20\}$ (TIERS) sizes the CPU (2^t), ALU (2^{t+1}) and Poseidon2 (2^{t+2}) tables and bounds the cycle count by $2^t - 1$. The prover selects the smallest tier whose cycle and Poseidon2-permutation budgets both suffice (an audit correction: a cycle-only selection could exceed the permutation capacity). Six table heights are declared in the proof: `program`, `input`, `keccak` and `sha256` (0 denoting absence in each case), `public` (mandatory; an empty segment declares height 2) and `memory`. `check_declared_heights` bounds all six in a fixed order before any is used to size a structure, so that a malformed header costs a small number of comparisons rather than a multi-second key construction.

Proof and verifier key. A `Proof` is $\{t, h_P, h_I, h_K, h_S, h_U, h_M, \text{public_values}[34], \text{batch}\}$ in postcard encoding. The verifier key (the preprocessed commitment with degree bookkeeping) is indexed by $(t, h_P, h_I, h_K, h_S, h_U)$; the memory height is excluded because the memory table has no preprocessed columns. The key is derived from a deterministically seeded configuration

reproducible by any verifier and cached in a 64-entry FIFO. The proving configuration draws fresh entropy, so two proofs of one execution differ.

Verification. `Machine::verify(hc,  $\pi$ )` checks the public-value count, the canonicity of every public value, $HC_{0..7} = hc$, $pv[TIER] = t$, the declared shape, equality of the proof’s degree-bit vector with that implied by the declared shape (which also fixes the instance count at nine, ten or eleven), and then executes `verify_batch` with the cached key. Since constraint set 6 the public values carry $PUB_{0..7} = H_{PUB}$: unlike H_{IN} , the public segment is *unsalted*, so `Machine::verify_public(hc,  $w$ ,  $\pi$ )` recomputes $H_{PUB} = public_digest(w)$ natively from caller-supplied words and compares. The chain publishes no words: `verify_call` and `verify_bundle` invoke it with $w = \emptyset$, so the only admissible H_{PUB} is the empty segment’s digest — plain `verify` would leave H_{PUB} bound only in-circuit, to a segment the chain never saw. Every untrusted integer is bounded before use.

4.13 The chain-side executor

`executor.rs` implements the trait with a `Machine` and the pinned bundle guest. `check_program` requires alignment, non-emptiness, that `base_pc + 4 $\ell$` does not exceed the 32-bit address space (audit finding ZH4: a program that could be paid for but never proved), and that every word decodes; it returns `hc`. `decode_and_check` performs the declared-shape checks and, for a bundle, additionally requires the declared program and input heights to equal the pinned guest’s, so that a proof of a differently shaped guest is rejected before key construction. `verify_bundle` invokes `Machine::verify` against `hc_bundle`; `verify_call` against the record’s `code_hash`; both read the eight outputs and the eight H_{IN} words as 32-bit integers. `warm` constructs keys for tiers 10, 12 and 14 at a deployed program’s height for the two input-height classes in use, with $h_K = 0$; `warm_bundle` constructs the bundle’s key at startup.

4.14 The recursion VM and block-level aggregation

Constraint set 6 coincides with the delivery of the design’s remedy for the 80-query proof size: block-level proof aggregation, implemented and merged (genesis-gated; activation awaits the production measurements, Section 7). The chain-facing piece is the recursion VM (rVM) of the upstream M5 milestones, vendored as `randprotocol-rvm`.

The recursion VM. The rVM is a second, smaller machine whose native word is the Goldilocks element and whose instructions are the verifier’s own operations: 24 field-arithmetic, memory, hashing and control instructions, extended by two measurement-gated precompiles — **REDUCE** (one chip row per batch-opening reduction column) and **SPONGE** (the leaf-sponge absorb loop as a row kind of the Poseidon2 chip) — to 26. Its emulator is the reference semantics; the production artifact is one *N*-generic *aggregate program* per admitted inner shape, which replays the native batch verifier over a witness tape of *N* proofs in a counted loop with a fresh challenger per proof, and publishes a four-element interface digest of `[inner_vk_digest || N || 34 · N]` — the inner verifier-key digest, the count, and every inner proof’s 34 public values — so a single registered digest covers every window size. The machine batch has eight instances (a preprocessed program table, register and RAM memory tables, a public table, a permutation-per-row Poseidon2 chip, a 66-column CPU, the optional reduce chip, and a range provider) over eight buses, tiers $\{8, \dots, 23\}$, and a verifier key (FriProfile, Program, Tier): an in-circuit program digest is impossible at the program’s 443,893 instructions, and the program is a public registered artifact. Three measurement-gated cuts (allocator liveness, **REDUCE**, **SPONGE**) brought the verifier program from 5,682,847 to 1,968,619 CPU rows per inner proof (51,605 permutations), tier 21 with 6.1% headroom; the plan’s **FRIFOLD/EXPBITS** precompiles were rejected with arithmetic (their combined share is $\approx 3\%$). At those heights one aggregate of *N* bundle proofs

is a proof of $N \times 1,968,619$ rows: the machine classes are $N=1 \rightarrow \geq 64$ GB, $N=2 \rightarrow \geq 128$ GB, $N=3 \rightarrow \geq 160$ GB of host memory on CPU, or the chunked CUDA backend on an 80 GB device (Section 7). The exit test of record is the test-profile twin: one rVM proof of the verifier program over one real bundle proof, natively verified (327,321 bytes, 18.3 s dominated by the program key build; the production-profile run and the $N \geq 2$ runs are written and deferred to the big-machine runbook).

The aggregator register. A second, permissionless register mirrors the validator register’s semantics field-for-field: join with `RegisterAggregator` on a bundle burning the genesis bond; `UnbondAggregator` (release at head +256); `WithdrawAggregator` paying bond $-B$ as a chain-derived note; `SlashAggregator` for equivocation only (two signed headers, one nonce, different content, submitted by anyone) — *refused on every aggregating chain since v0.6* (INTERFACE-1): the daemon’s honest retry after a lost race re-claims the same first unsealed rows under a fresh nonce, and the Lean model shows that every slashing rule writable on the headers, including the audits’ proposed restriction to overlapping cover sets within a time window, still slashes that retry; the bond stays refundable and slashed in Equation (3) is identically zero. The registered artifacts live in genesis: the admitted shapes and, per shape, the aggregate program digest; the verifier-key cap is rebuilt by every node at startup (a one-time ≈ 30 – 70 s build at production, cached 64-entry FIFO, after which a warm aggregate verify costs ≈ 1 – 2 s).

The Aggregate action and admission. A bundle-less action

Aggregate (covers, proof, aggregator, nonce, time, r , envelope, signature)

covers a set of finalised, in-window, unsealed bundles of one registered shape (at most `MAX_COVERS` = 3 at the measured machine classes). Admission runs nine steps in cheap-before-expensive order: caps and chain; register row, nonce and signature; time; the cover set; the payout commitment’s newness (mempool-claimed, as is the (aggregator, nonce) pair); the per-bundle declared-shape check against the stored proof headers; the recompute-and-compare — the node recomputes the inner verifier-key digest from the registered shape, the interface list from the covered bundles’ public fields, and the interface digest, and compares all three against the aggregate’s published values (the M5.3 pinned vectors are the byte-for-byte conformance suite for this step); the rVM verification on the admission workers; and payment: a deposit note of $\text{subsidy}(n) + \sum \text{excesses}$ to the payout address, its commitment derived by the ledger as for a withdrawal. One aggregate covers one shape; the chain’s transfer relation is one shape in practice, and the admission check is what makes it a fact. Two of these bounds are block-validity rules in the ledger and not only admission policy, since the pre-v0.1 security review (H1): a leader holding an aggregator key could otherwise re-sign a committed proof under fresh nonces, and every replica, applying each **Aggregate** in a block with a seal-blind covered source, would have minted the subsidy once per copy. The ledger’s `unsealed_fees` bucket therefore records every included bundle, excess or not, keyed by transaction hash with its coverable-until height; the entry leaves at the covering aggregate or at the window’s sweep and never returns, and step four refuses any cover without an entry (`CoverNotCoverable`: never applied, already covered, or past its window), a missing entry at payment being the same refusal rather than a zero share. A block carrying a second **Aggregate** is refused at the second, by index, before its covers are looked up (`SecondAggregate`). The node’s three admission verdicts (unknown, sealed, outside the window) remain its error reporting. Binding the aggregate proof’s public input to the (aggregator, nonce) pair was deliberately left undone at this revision: with the coverable set a block rule a replay has nothing to re-cover, and the binding would change the rVM program’s interface. *Addendum (AGG-2, cc20e91, 2026-09-25)*: the binding is now in place; the interface is $[\text{inner_vk} \parallel N \parallel B \parallel 34 \cdot N]$ with $B = H(\text{“rand-aggregate-bind-1”, chain_id} \parallel \text{aggregator} \parallel \text{nonce})$

as eight words, so an aggregate re-signed by another key or under another nonce is refused, and the aggregate program’s digest changed accordingly.

What an aggregate proves. No earlier revision of any of the papers stated the aggregate’s guarantee as a theorem; the Lean inventory recorded the omission, and the statement is the gate the verification’s phase 2b sets for activation.

Definition 4.1 (Aggregate relation R_{agg}). *Fix an admitted shape Σ (FRI profile, tier, six declared heights, the bundle guest’s hc) with inner verifier key vk_Σ and its digest d_Σ . For N covered bundles with published fields $\text{pub}_1, \dots, \text{pub}_N$, let $\text{pv}_i \in \mathbb{F}_p^{34}$ be the public values the ledger recomputes from pub_i (the bundle’s digest, hc , the transaction binding and the declared heights) and B the binding word of AGG-2. The aggregate’s public output is the interface digest*

$$\text{iface} = H_{\text{rVM}}([d_\Sigma \parallel N \parallel B \parallel \text{pv}_1 \parallel \dots \parallel \text{pv}_N]),$$

and $(\text{iface}, (\pi_1, \dots, \pi_N)) \in R_{\text{agg}}$ if and only if $\text{Verify}(\text{vk}_\Sigma, \text{pv}_i, \pi_i) = 1$ for every i .

Theorem 4.2 (Aggregate soundness, conditional). *Assume (a) the rVM batch STARK at the production profile is knowledge-sound for its eight-instance constraint system with LogUp balance, with error ϵ_{rVM} ; (b) the registered aggregate program P_{agg} , run on a witness tape of N proofs, computes the native verifier $\text{Verify}(\text{vk}_\Sigma, \cdot, \cdot)$ exactly and publishes iface only when every run accepts (program correctness, checked against the M5.3 pinned vectors but not proved); (c) the inner batch STARK is knowledge-sound with error ϵ_{STARK} (Theorem 5.8); and (d) H_{rVM} is collision-resistant on the fixed-length interface list. Then from any prover that causes the ledger to accept an **Aggregate** covering bundles $1, \dots, N$ with probability non-negligibly exceeding $\epsilon_{\text{rVM}} + N\epsilon_{\text{STARK}}$, an extractor obtains, for every i , a witness w_i with $(\text{pub}_i, w_i) \in R_{\text{bundle}}$.*

Proof sketch. Admission recomputes d_Σ , the list and iface from the registered shape and the covered bundles’ own public fields and compares them with the aggregate’s published values, so an accepted aggregate’s interface digest is the honest one; by (d) its preimage is the honest list. By (a) an extractor obtains an execution of P_{agg} — the program is a public registered artifact bound by the rVM verifier key, not digested in-circuit — on a tape $(\pi_1, \dots, \pi_N)$ whose published iface equals that digest, and by (b) that execution accepted every π_i against vk_Σ and pv_i . Each π_i is then an accepting inner proof of the bundle guest on the ledger’s own public values, and (c) with Theorem 5.8 extracts w_i . $\square$

Remark 4.3 (What the rVM verification found). *Hypothesis (a) is where the machine was incomplete at the revision this paper describes, and the zk reviews of 27–28 September 2026 and the Lean development of the rVM’s chips (on the reviews’ patched constraint sets) agree on the list. A **STOREE** bound only the low lane of its stored word (RVM-1); the fix binds the high lane through the register table, so any proof of the machine’s soundness must include that table and not the CPU table alone (the Lean confirms that the CPU table accepts the forged trace and the register table refuses it). The reduce chip’s continuation constraint alone lets a run park on the table’s last row and escape every transition constraint (RVM-2); the required last-row constraint is that the final row be padding, and the fixed chip meets it. The reduce chip’s descriptor pointer and bases could wrap at $p - 1$ (ARITH-4, the audits’ OPCODES-3 and TABLES-2), closed by the range checks of ZKQ-3 and proved closed only under the hypothesis that the limbs are bytes, a hypothesis the Lean shows is necessary; the fixed reduce chip has 27 constraints, of degree up to 5. All of these are in v0.6 (the rVM re-vendored at circuits **fbe29b8** and **642db8d**) and the in-circuit **VERIFIER-1** check on the recursive verifier is in constraint set 7. Hypothesis (b) — that the aggregate program is the verifier — is checked by conformance vectors and by one end-to-end recursive proof, not proved; a verifier-program model is the verification’s phase 2b, and until it exists, and a forged aggregate has been tried against the fixed machine, aggregation remains genesis-gated (Section 7).*

Subsidy, sealing and pruning. The subsidy is $\text{subsidy}(n) = 100 \text{ RAND} \gg \lfloor n/210,000 \rfloor$ over sealed blocks (zero from the 64th halving, an asymptote of $\approx 42 \text{ M RAND}$), minted at the schedule index n of the sealing block; the proving share is each covered bundle’s excess over the fee floor, held in `unsealed_fees` until coverage or expiry (Section 4.3); a bundle at exactly the floor is recorded with excess zero and is coverable all the same, so an aggregate over it applies and pays the subsidy alone. The sweep of expired entries is part of `Ledger::close_block`, the one block-end step that both the proposer and the replica run before the state root (the review’s L1: the proposer had omitted it, so the first expired bucket on an aggregating chain would have made every leader reject its own block). A sealing block writes the per-bundle `sealed_by` marks atomically with the block (`seals` family); after a further 256 blocks a node may rewrite each covered bundle’s record to its *pruned* form — the full transaction plus its 34 public values and 7 shape bytes, $\approx 3.3 \text{ KB}$ against the $\approx 1.3 \text{ MB}$ raw record — while aggregate proofs, cover lists and unsealed bundles are never pruned. The trust model is unchanged: every historical bundle remains covered by a STARK the node verified itself, the aggregate or the original, so no finality-signature trust is introduced; the state root is unaffected, since proof bytes are node-local witness data.

Sealed-form sync. A pruned bundle is accepted by a joining node only when its covering aggregate is applied in the same batch (batch-atomic coverage) or already local; anything else is a raw-form fallback to another peer, never a ban. Serving is *coverage-closed*: a `Blocks` response extends past the count asked and the soft byte budget until every served pruned entry’s seal mark resolves inside the batch (capped only by the reader’s wire limit), because a batch cut between a pruned block and its cover is deterministically unservable and produced the sealed-sync stall the capstone test demonstrated. A joining node thus replays one rVM verification per sealed window in place of N bundle verifications. What binds the served bytes is the certified transaction root: since the review’s M1 the transaction identifier is taken over the bundle with its proof replaced by the proof’s digest (Section 4.2), so the pruned marker form — the marker followed by exactly that digest — hashes to the raw identifier, and the root a quorum certificate certifies binds every byte of a sealed block’s transactions but the proof bytes the covering aggregate stands in for. `apply_block_for_sync` verifies the root over the transactions as served in both forms and requires each marker’s side-table entry to attest the hash the root certified; the side table can vouch for a proof but never rename a transaction. Before the fix the root leaf for a marker was the side table’s own attested hash, which nothing recomputed from the served bytes, so a sync peer could substitute a pruned bundle’s envelopes, or a stateless `Call`, and the joiner would persist and re-serve them.

The daemon and gating. `rand-node aggregate` (with `-watch`) is the aggregator’s work loop: poll the unsealed work list, fetch the raw bundles, prove the aggregate, seal the payment at the current schedule index, sign and submit. Everything above hangs off `genesis.aggregation`: a chain without the section behaves byte-for-byte as chains 8, 9 and 10 do — no register, no admission, full fee to the proposer — and activation, on a later cut, awaits only the production measurements that fill `admitted_shapes[0]` (Section 7). The cluster suite’s sealed-sync capstone runs with a 32-block window rather than 8: once the window became a block rule, the view change or two that follow the proof left an 8-block window missing its last admissible block once in two runs.

4.15 Storage

Each data directory holds one RocksDB instance with sixteen column families: `blocks`, `qcs`, `block_index`, `txs` (whose values are a `Raw/Pruned` enum: the full record, or the transaction with its 34 public values and 7 shape bytes once sealed and past the pruning gate), `meta`,

programs, **receipts**, **notes** (leaf index to commitment, envelope and height; dense from zero and the sole copy of the leaf set), **nullifiers** (to spending height), **anchors** (height to root, pruned to the window), **validators**, **epoch_sets**, **bridge_spent**, **bridge_burns**, and **seals** (per-bundle seal marks and per-aggregate payment records, written atomically with the sealing block). **meta** holds the head, genesis hash, chain identifier, safety state, tree frontier, pinned bundle digest, supply counters, bridge metadata, and — on a chain with the section — the aggregation config, the aggregator register and the **unsealed_fees** bucket, all three restored at load. A commit is a single fsynced **WriteBatch**.

verify_chain checks block 0 against the genesis file and the build’s bundle guest against the pinned digest, then replays every block through the executor, including bundle and call proofs, in **quick** mode, additionally verifying every proposer signature and QC in **full** mode. On the first inconsistency it truncates to the last consistent height, rewriting notes, nullifiers, anchors, validators and frontier from the replayed ledger while retaining the safety state, and defers to synchronisation for the remainder. It also recomputes the supply counters and reports any stored snapshot that disagrees with the replay. Because replay re-verifies every historical proof under the running binary’s constraint system, a constraint-set change is a hard fork; six have occurred (chains 4 to 8, and constraint set 6 at the chain-9 cut of 2026-09-16, which also carried the transaction-identifier change of Section 4.2).

4.16 Networking, mempool, event loop and RPC

Networking employs libp2p 0.54 over TCP with Noise and Yamux; gossipsub topics per chain identifier for consensus, transactions and status; Kademlia seeded from a bootstrap list; optional mDNS; a CBOR request-response protocol for block synchronisation in batches of at most 100; ping and redial. A synchronising node re-executes every fetched block, including proofs, before acceptance.

The **mempool** has neither senders nor nonces to order and therefore tracks *conflicts*. It indexes each pending transaction’s nullifiers, commitments (including the deposit commitment a withdrawal or a bridge attestation would create, which is not on the wire), bridge digests, and, on an aggregation chain, the payout commitment of an aggregate and the (aggregator, nonce) pair. Admission splits in two: a cheap **precheck** (duplicate hash, conflict, capacity, the byte-level aggregate preflight) runs synchronously, and the state-dependent half — including every STARK verification — runs on four blocking workers behind a 64-deep queue against a lazily refreshed ledger snapshot, with **insert_verified** re-running the state-dependent checks against the tip the transaction is actually pooled on. A bounded refused-hash cache (8,192 entries, FIFO, permanent verdicts only) answers a repeat refusal for free, and a per-peer token bucket (burst 16, refill 4/s, keyed on the forwarding peer’s propagation source) meters gossiped submissions. The proposer orders candidates by fee with ties broken by hash, so that honest proposers over one pool select the same block; re-checks the inexpensive half of admission against the ledger under construction; and fills to 2,000 transactions or 4 MiB. After each commit, transactions whose note has been spent, whose commitment now exists, or whose anchor or time has left the 256-block window are pruned.

The **event loop** multiplexes network, RPC, consensus timers and the pacer; executes consensus actions; on commit writes the batch, prunes the mempool, publishes status, persists a new epoch’s set and warms verifier keys for newly deployed programs. Admission-time proof verification no longer runs here: gossipsub uses application-level validation, so a transaction is forwarded only once it has verified on the worker pool — deliberately under **ValidationMode::Permissive**, since a Strict/Permissive mix across a fleet drops messages, at the cost that every delivered message must be reported back to gossipsub exactly once (accept, reject or ignore) or this node silently stops forwarding it. Block-*application* verification (**apply_block** inside **on_proposal**) remains on the loop: moving it changes when a vote is emitted, a consensus decision not taken lightly.

The **RPC** is redacted by construction; no balance query exists, a balance being a property of a spend key. The shielded surface comprises `rand_getTreeInfo`, `rand_getCommitments` (a page of at most 1,000 leaves with envelopes), `rand_getNullifiers`, `rand_getAnchor`, `rand_getWitness` (a 32-level path for one leaf), `rand_getCompactBlocks` (up to 128 blocks per call: per block its height, hash and timestamp, and per transaction its note commitments with envelopes and its spent nullifiers — the one-round-trip note stream a light wallet syncs with), `rand_sendTransaction`, the test-network `rand_mint`, and a `rand_getTransaction` that renders a transfer as two nullifiers, two commitments, a fee, a burn, an asset, a time and two lengths. Public state is served in full: `rand_getValidators`, `rand_getEpoch`, `rand_getSupply`, program and receipt queries, `rand_getCallEnvelope`, and the bridge’s registry and burn log. Requests may be batched (at most 20 per POST; notifications are refused `-32600`), a WebSocket `newHeads` subscription shares the port (bounded: 64 connections, 8 subscriptions each, 64 KiB frames, 256 heads of backlog), and the request body limit is derived from the largest admissible transaction (≈ 8.86 MB). A node may hold *viewing keys* — never spend keys — for explorer-side scanning: `rand_importViewingKey` registers one in memory (at most 64, cleared at restart) and `rand_getViewingNotes` lazily scans at most 10,000 leaves per call, returning received notes with their nullifier and spent state and ovk-opened sent notes; `rand_checkTransaction(hash, key)` is the stateless one-call payment proof, disclosing whatever the given per-transaction key opens, with a wrong key indistinguishable from an empty disclosure. The aggregation surface (Section 4.14) adds `rand_getAggregate`, `rand_getAggregators`, `rand_getUnsealed`, per-block `sealed/sealed_by` fields, and an aggregation section of `rand_status`.

4.17 The wallet

The wallet stores the spend key alone (key file version 2) together with a cache of opened notes, recoverable by rescanning from leaf 0. `rand sync` retrieves every leaf and envelope and trial-decrypts each with the viewing key (as receiver) and the outgoing viewing key (as sender); the node learns only that scanning is occurring. `rand send` selects at most two notes (a wallet holding value across three or more notes must first consolidate), retrieves the head anchor and one witness per real input, proves the bundle locally in approximately 100 s at tier 14, seals two envelopes, submits, and awaits the commit; with `-no-wait` the inputs are marked pending and released when the nullifier appears or the 256-block window elapses. `rand bond` proves a bundle whose burn is the stake. `rand call` proves the program and then a fee bundle, and by default seals a *call envelope*: the salt and input vector under a fresh per-call key wrapped to the caller’s ovk and optionally to an auditor’s address, with H_{IN} as associated data. `rand open-call` decrypts it, recomputes H_{IN} , re-executes the program in the emulator, compares the outputs with the receipt, and exits with a non-zero status on any discrepancy. `-no-envelope` publishes nothing, irreversibly. `-cuda` selects the GPU backend and fails rather than falling back.

4.18 The bridge codec and the bridge as notes

The wire format and guardian model are unchanged from the account-based revision and are summarised: an attestation comprises a version byte, a guardian-set index, n 66-byte secp256k1 signatures (index, r , s , v) with strictly increasing indices and low- s normalisation, a 51-byte body header and a payload; the digest is $\mu = \text{keccak256}(\text{keccak256}(\text{body}))$; the quorum is $\lfloor 2n/3 \rfloor + 1$; a rotation must originate from the governance emitter and be signed by the *current* set, with a one-day grace period applying to transfers only; an inbound transfer must originate from the registered emitter of its own emitter-chain field; consumed digests constitute the replay set.

Phase S3 relocated the value. `BridgeState` holds no balances. The registry maps an asset identifier to `{chain, token, index}` with dense indices assigned at first observation (0 denotes

RAND), and `next_index` is consensus state. A `BridgeAttest` deposits one note

$$\{\text{pk} = \text{recipient.pk}, \text{from} = 0, \text{amount}, \text{asset} = \text{index}, \text{time}, r\}$$

whose commitment the chain derives from the attested amount; the payload’s 32-byte recipient field carries the domain-separated BLAKE3 hash of the recipient’s shielded address ($\text{pk} \parallel \text{ek}$), and the action carries the full address, which the ledger hashes and compares. The depositor must predict the note’s `time` and `asset` because the envelope is sealed before the transaction exists; both are subject to the ledger’s windows, and a lost first-observation race costs a fee bundle and a re-proof rather than an unopenable note. The payload’s relayer fee is recorded but not paid, so the deposit is gross and the pool total matches the amount locked on the source chain. A `BridgeBurn` is the only two-bundle transaction: the transaction’s RAND bundle pays $2B$, and the action’s asset bundle ($\text{asset} = \text{index}$, fee 0) burns exactly the amount, of which the relayer fee is a portion paid on the destination chain. Both bundles undergo identical admission, with nullifier and commitment uniqueness enforced across the pair. The bridge root, re-pinned in S3, commits the registry with indices, `next_index`, the spent set and the sequence, and enters the state root only on a bridged chain. Thirty-nine externally generated vectors are pinned at the signature level (23 exact); the account-era ledger-level vector pass has not been reconstructed for the pool.

Addendum (chain 14, v0.5): one zUSD, seven backings, the hardened bridge. The asset registry moved out of `BridgeState` into the ledger-level token registry; a bridged token has `MintAuthority::Bridge` with a list of backings, each a (chain, token, decimals) triple with its own `locked` counter, and `total_supply = \sum \text{locked}` holds because lock and release are the only writers of either. Chain 14’s one bridged token, zUSD (index 1, eight decimals), is backed by USDT and USDC on Ethereum, BNB Smart Chain and Solana and by USDT on Tron; a burn naming a backing that is unlisted, short of the amount or not releasable is refused before its digest or proof are touched. A burn is one hidden-asset bundle, not two. Hardening: B1, a per-backing daily mint cap (genesis `tokens.mint_cap_per_day`, $100,000 \times 10^8$ on chain 14) and a Dilithium2 pause key that can only pause (`PauseMints`; `UnpauseMints` needs the post-quantum guardian quorum); B2, `MAX_TIMESTAMP_STEP_MS = 60,000` as a replayed validity rule and `MAX_CLOCK_DRIFT_MS = 15,000` as a vote rule; B3, a second quorum of Dilithium2 co-signatures by the same guardians over “rand-bridge-pq-cosign-1” $\parallel \text{chain_id} \parallel \mu$ on every attestation, inside the transaction binding; B4, `RegisterBridgedToken` and `ListBacking` under that quorum after genesis; F1, the deposit blinding derived as $\text{blake3}(\text{“rand-deposit-r-1”} \parallel \mu)$ rather than chosen by the submitter. A genesis-gated *rules v2* (v0.5.4, absent from chain 14) adds `RotatePqGuardians` and `RotatePauseKey` under a rotation nonce and replaces the calendar-day cap with a rolling window plus a registry-wide cap. On mainnet, 2026-09-20: 1 USDT locked, minted, transferred, burned and released on each of the four chains, then 9 USDT more minted on each and left locked — 36×10^8 units of zUSD against 36 USDT in custody, audited equal. The bridge runs on guardian set 0; the post-quantum keys at genesis are those of set 1 and co-sign every mint.

5 Formal Analysis

Earlier revisions of the design [1] analysed an idealised system. This section analyses the implemented system at its deployed parameters. Where the implementation is weaker than the design, the weaker bound is stated.

5.1 Preliminaries and notation

Let λ denote the security parameter. $\mathbb{F}_p$ is the Goldilocks field, $p = 2^{64} - 2^{32} + 1$, and $\mathbb{F}_{p^2}$ its degree-2 extension. A machine word is an integer in $[0, 2^{32})$ embedded in $\mathbb{F}_p$; a Word8 is a

tuple of eight machine words, the encoding of four Goldilocks elements and the width of every key, commitment, nullifier and digest. $H(d, m)$ denotes the Poseidon2 sponge applied to $[d, m]$ with domain tag d . Classical bounds are distinguished from Grover bounds ($2^{n/2}$ for preimage search [6]) and BHT bounds ($2^{n/3}$ for collision search [5]). Signatures are ML-DSA at the Dilithium2 parameter set [42]; the KEM is ML-KEM-768 [41]. An argument of knowledge is complete, knowledge-sound and zero-knowledge in the standard sense; the STARK considered here is statistically zero-knowledge, as stated by Plonky3's hiding FRI implementation.

5.2 The relations proved by the node

Both relations are instances of a single machine relation.

Definition 5.1 (Machine relation R_{vm}). *The statement is $x = (\text{pc_entry}, t, \text{out}_{0..7}, \text{hc}_{0..7}, \text{in}_{0..7}, \text{pub}_{0..7}) \in \mathbb{F}_p^{34}$ together with declared heights $(h_P, h_I, h_K, h_S, h_U, h_M)$. The witness comprises a program $P = (\text{base_pc}, w_0, \dots, w_{\ell-1})$, an input vector $v \in [0, 2^{32})^n$, a salt $s \in [0, 2^{32})^4$, a public segment u and an execution. $(x, w) \in R_{\text{vm}}$ if and only if: every $x_i < p$ and every $\text{out}_i, \text{in}_i < 2^{32}$; $\text{pc_entry} = \text{base_pc} \equiv 0 \pmod{4}$, $\text{base_pc} + 4\ell \leq 2^{32}$, and every w_j decodes; $\text{hc} = \text{hc}(P)$ and $\text{in} = H_{\text{IN}}(s, v)$ as defined in Section 4.10, and $\text{pub} = \text{public_digest}(u)$, unsalted, so the verifier recomputes it natively; $t \in \{10, 12, 14, 16, 18, 20\}$ and the emulator, started at base_pc with zero registers and memory and answering $\text{READ_INPUT}(i)$ with v_i and $\text{READ_PUBLIC}(i)$ with u_i , halts after c rows with $c + \lceil \ell/4 \rceil + 1 + \lceil n/4 \rceil \leq 2^t - 1$ and at most 2^{t-3} Poseidon2 permutations; the output slots at halt equal out , unwritten slots being 0 and no slot written twice; and the declared heights satisfy $2^{h_P} \geq \ell + 1$, $2^{h_I} \geq n + 1$, $h_K = 0$ if and only if no KECCAK was executed and otherwise $2^{h_K} \geq 32 \cdot \#\text{keccak}$ with $h_K \leq \min(t + 5, 20)$, h_S on identical terms for SHA256, $2^{h_U} \geq |u| + 4$ with $h_U = 2$ for an empty segment, and $t + 2 \leq h_M \leq 24$ with $2^{h_M} > \#\text{accesses}$.*

Definition 5.2 (Call relation). R_{call} is R_{vm} with hc equal to the `code_hash` of a deployed program. The verifier compares hc but does not compare H_{IN} against any reference value; H_{IN} is therefore a guest-visible commitment that may subsequently be opened, not a verifier-side identity check.

Definition 5.3 (Bundle relation R_{bundle}). *Let*

$$\text{pub} = (\text{anchor}, \text{nf}_{0..3}, \text{cm}_{0..3}, \text{fee}, \text{burn}_A, \text{burn}_R, \text{burn_asset}, \text{time})$$

denote a bundle's published fields: slots 0, 1 carry a private asset A and slots 2, 3 carry RAND (asset 0); write $\text{asset}(k)$ for the slot's asset. $(\text{pub}, w) \in R_{\text{bundle}}$ if and only if there exists a witness

$$w = (\text{sk}, A, \{(\text{from}_k, a_k, \text{asset}_k, \text{time}_k, r_k, \text{path}_k, \text{idx}_k)\}_{k < 4}, \{(\text{pk}'_k, a'_k, r'_k)\}_{k < 4})$$

such that, with $\text{nk} = H(\text{NK}, \text{sk})$ and $\text{pk} = H(\text{PK}, \text{nk})$:

1. *for each k with $a_k > 0$: $\text{cm}_k^{\text{in}} = H(\text{CM}, \text{pk}, \text{from}_k, a_k, \text{asset}_k, \text{time}_k, r_k)$, $\text{MerkleVerify}_{32}(\text{cm}_k^{\text{in}}, \text{path}_k, \text{idx}_k) = \text{anchor}$, and $\text{asset}_k = \text{asset}(k)$;*
2. *$\text{nf}_k = H(\text{NF}, \text{nk}, \text{cm}_k^{\text{in}})$ for every k , irrespective of whether the input is a dummy;*
3. *$\text{cm}_k = H(\text{CM}, \text{pk}'_k, \text{pk}, a'_k, \text{asset}(k), \text{time}, r'_k)$ for every k ;*
4. *$a_0 + a_1 = a'_0 + a'_1 + \text{burn}_A$ and $a_2 + a_3 = a'_2 + a'_3 + \text{fee} + \text{burn}_R$ over the integers, with every amount, the fee and both burns below 2^{63} ;*
5. *$\text{burn_asset} = A$ if $\text{burn}_A \neq 0$ and 0 otherwise;*

6. the four nf_k are pairwise distinct and the four cm_k are pairwise distinct.

The guest realises this relation as R_{vm} with $P = \text{bundle_hidden}$ ($v1$ on chains 14 and 15; the branch-free $v2$ on chain 16), $n = 1,204$, and single output $\text{out} = H(\text{BUNDLE}, \text{pub}, \text{bad})$, where $\text{bad} = 0$ if and only if clauses 1, 4 and 6 hold (clauses 2, 3 and 5 hold structurally, the guest computing them from its own derived pk , the slot's asset and the bundle's time). The ledger accepts if and only if $\text{out} = H(\text{BUNDLE}, \text{pub}, 0)$ and the proof's public segment carries the transaction binding (Section 4.9). Until chain 13 the relation had two slots of one public asset, $\text{pub} = (\text{anchor}, \text{nf}_{1,2}, \text{cm}_{1,2}, \text{fee}, \text{burn}, \text{asset}, \text{time})$, a single balance $a_1 + a_2 = a'_1 + a'_2 + \text{fee} + \text{burn}$ and $n = 612$; the first Lean inventory of the papers (September 2026) recorded that this paper still stated that relation while chain 15 ran the four-slot guest, and the definition now states what runs.

Theorem 5.4 (Completeness). *For every $(x, w) \in R_{\text{vm}}$ within the caps, `Machine::prove` yields a proof accepted by `Machine::verify`; for every $(\text{pub}, w) \in R_{\text{bundle}}$, the wallet's bundle proof passes all ten admission steps.*

Evidence. The trace builders are total on valid executions; the tier selected by `Tier::for_workload` satisfies both budgets by construction; the public values are computed by the host functions the AIRs constrain. The property is exercised by the end-to-end tests on every built-in guest, by the wallet-flow test (six bundle proofs and a call proof against a single-node chain) and by the 16 cluster tests. $\square$

The following lemmas concern AIR constraints and bus balances and are conditional on the soundness of the batch STARK and of LogUp.

Lemma 5.5 (Program binding). *Any accepted proof with digest hc corresponds to a program-table witness whose valid rows are exactly $\{(\text{base_pc} + 4j, w_j) : j < \ell\}$ for the unique $(\text{base_pc}, \ell, w)$ absorbed by the digest rows, and every fetched instruction belongs to that set.*

Proof sketch. The digest rows consume exactly ℓ messages on `PROGRAM_WORD`, with base_pc and ℓ seeded into the sponge capacity. The program table provides one message per row with `MULT_WORD = VALID`, and its PC column is an arithmetic progression of step 4, so no two rows share an address. LogUp balance forces equality of the provided and consumed multisets, hence of the sets. Fetches on `PROGRAM` carry zero multiplicity on invalid rows, so only digested rows execute. The one-sided constraint that formerly admitted an undigested reachable suffix was closed in constraint set 3 and is pinned by a cheating test. $\square$

Lemma 5.6 (Input binding). *In any accepted proof, the pairs returned by `READ_INPUT` are drawn from the vector absorbed into H_{IN} ; repeated reads of one index agree, and a read of an index $\geq n$ is unsatisfiable.*

Proof sketch. The input table's real rows carry $(\text{idx}, \text{word})$ with the index pinned to the row position. On `INPUT_DIGEST` they provide with count `IS_REAL`, and the CPU's real input-digest rows consume exactly $\{(i, v_i)\}$, a set equality as in Lemma 5.5. On `INPUT_READ` they provide with count `IS_REAL · MULT_READ` and each read consumes one pair. The buses are separate so that the mandatory digest copy and the reads cannot exchange budget; a shared bus permitted under-absorption of an index while a read of it succeeded. $\square$

Lemma 5.7 (Output, memory and arithmetic consistency). *In any accepted proof, each out_i is the word of the unique write to slot i , or 0; every load returns the most recently stored value at its address, or 0; every store's merged bytes originate from a register value; every ALU result satisfies $c = \text{op}(a, b)$ as defined by the emulator; every Poseidon2 and Keccak row group is entered only from its own `ecall` row and exited only after its final row; and every memory address and hash pointer is bounded below 2^{30} .*

Proof sketch. Boolean WRITTEN_i accumulators pin the outputs. The memory table is sorted by (space, address, timestamp) with range-checked differences and read-after-write transition constraints, and the sort key is computed additively, so that an honest execution with high addresses is provable. Multiplication is the exact identity $AB = T_0 + 2^{16}T_1 + 2^{32}T_2$ over 16-bit halves, valid in $\mathbb{F}_p$ since $(2^{32} - 1)^2 < p$, with a 24-bit-bounded carry; division is $|a| = q|b| + r$, $r < |b|$, with sign correction. The hash row-group entry gates and the confinement of HASH_FIN to write-back rows are the two critical corrections of the 2026-09-12 audit; their attack witnesses verified before the correction and are rejected after it. The Keccak chip derives every limb from asserted bits, so its memory traffic is 32-bit by construction. $\square$

Theorem 5.8 (Knowledge soundness, conditional). *Assume that the Plonky3 batch STARK with the production profile is knowledge-sound for AIR satisfiability with LogUp balance, with error ϵ_{STARK} , and that Poseidon2 is collision-resistant as used in hc , H_{IN} and the note layer — that is, on domain-separated messages of one fixed length per domain, the only form in which the padding-free sponge is collision-resistant (Section 4.10). Then from any prover that causes $\text{Machine}::\text{verify}(\text{hc}, \cdot)$ to accept with probability non-negligibly exceeding ϵ_{STARK} , an extractor obtains (P, v, s) and an execution with $(x, w) \in R_{\text{vm}}$; and from any prover that causes the ledger to accept a bundle with published fields pub , an extractor obtains w with $(\text{pub}, w) \in R_{\text{bundle}}$.*

Proof sketch. The first assertion follows from Lemmas 5.5–5.7 and the CPU transition constraints (program-counter chaining, branch and jump targets, halting), collision resistance excluding a second preimage of hc or H_{IN} . For the second, the extracted execution is by the first assertion an honest execution of the pinned bundle guest on some 1,204-word v ; the ledger’s equality $\text{out} = H(\text{BUNDLE}, \text{pub}, 0)$ and collision resistance force the guest’s output preimage to equal $(\text{pub}, 0)$, so that $\text{bad} = 0$ and the guest’s anchor, nullifiers, commitments, fee, burns, burned asset and time coincide with the ledger’s; $\text{bad} = 0$ is precisely the conjunction of clauses 1, 4 and 6 of Definition 5.3, and clauses 2, 3 and 5 hold because the guest computes them from its derived pk , the slot’s asset and the published fields. The dummy bypass cannot be exploited because its condition reads the registers that feed the balance sum. $\square$

Remark 5.9 (Assumptions). *The soundness of Plonky3’s batch STARK and LogUp gadget is assumed, not established here. The machine’s constraint system has been revised repeatedly: five defects were corrected in constraint set 2, one in set 3, one in set 4, and two critical defects together with several completeness corrections in set 5, each identified by review after deployment. The evidence for the lemmas consists of the constraint system and its 83 cheating tests. Both reviews were internal; no independent audit has been performed.*

Proposition 5.10 (Disclosure by tier and declared heights). *An accepted proof discloses the tier t , whether KECCAK was invoked ($h_K = 0$ or otherwise), the program’s size class h_P , the input vector’s size class h_I and the memory-access size class h_M , and no further information about the cycle count or the trace. For the bundle guest all five are constants; a bundle proof therefore discloses nothing about its execution.*

Theorem 5.11 (Statistical zero knowledge). *Under the hiding property of HidingFriPcs and $\text{MerkleTreeHidingMmcs}$ with fresh entropy per proof, a simulator given x and the declared heights produces a transcript statistically close to an honest proof. The prover draws the H_{IN} salt from operating-system entropy on every proof, so that H_{IN} is a hiding commitment to v . Two qualifications apply: the hiding is statistical rather than perfect; and hc is unsalted, so a verifier able to enumerate candidate programs can confirm which was deployed. The latter does not disclose the execution, and the program is public on chain in any case. Two further hypotheses, implicit in “the hiding property” and stated here because the deployed system did not always meet them: every committed table has more rows than the verifier opens (80 queries plus two out-of-domain points, so at least 2^7), and every LogUp running total is blinded.*

Remark 5.12 (For which chains the theorem holds). *The Lean inventory of September 2026 marked the zero-knowledge statements false for chain 15, and correctly so. A table under 2^7 rows is determined by its openings, so a call proved by a wallet older than v0.6 with 31 or fewer private words carried its inputs in its input table, a one-permutation keccak call its hashed secret, and every program under 64 words its per-instruction fetch counts (the 2026-09-27 zkVM review’s INT-1, COV-2, INT-6, HCS-3); and the unblinded LogUp totals of the branching bundle guest disclosed which input slots were real and the popcount of each spent leaf index (INT-2, GV-1). Bundle proofs cleared the height floor; the total leak was theirs alone. v0.6 (2026-09-28) floors the input, keccak and sha256 tables at 2^7 rows in the prover and refuses to pool a call proof below it, and ships the branch-free guest v2 (`hc_bundle 651043e2...`) for a genesis to select; constraint set 7, on which chain 16 was cut the same day, blinds every LogUp terminal with an $\mathbb{F}_{p^2}$ blind, floors every declared table at 2^7 , range-checks the input and public words and pins guest v2. The theorem holds on chain 16 and later; on chains 14 and 15 it holds for the published fields and the envelopes and fails for the proof bytes in exactly the two respects above.*

5.3 Soundness of the deployed parameters

The production profile is rate $\rho = 1/8$, folding arity 8, $q = 80$ queries, $g = 20$ grinding bits, with challenges in $\mathbb{F}_{p^2}$. Plonky3’s conjectured-soundness function is pinned by a unit test at ≥ 100 bits for this profile.

Theorem 5.13 (FRI soundness as instantiated). *Let N be the largest extended table height in a proof, $N \leq 2^{25}$ (the memory table at its declared ceiling of 2^{24} , plus one bit of zero-knowledge randomisation). Then*

$$\epsilon_{\text{FRI}} \leq \underbrace{\frac{CN}{|\mathbb{F}_{p^2}|}}_{\text{field/DEEP}} + \underbrace{\left(\frac{1+\rho}{2}\right)^q}_{\text{queries, proven}} + 2^{-g}, \quad (4)$$

which at $C = 8$ yields a field term of approximately 2^{-100} and a query-plus-grinding term of

$$q \log_2 \frac{2}{1+\rho} + g = 80 \times 0.830 + 20 \approx 86.4 \text{ bits (proven, proximity gaps)}, \quad (5)$$

as against, under the ethSTARK conjecture,

$$q \log_2 \frac{1}{\rho} + g = 80 \times 3 + 20 = 260 \text{ bits (conjectured)}. \quad (6)$$

The composite bound is therefore approximately 100 bits conjectured, limited by the field term, and approximately 86 bits proven, limited by the queries. A provable 100 bits at this rate requires $q \approx 97$ or $g = 34$; 128 bits requires a degree-3 extension.

Proof sketch. As in the earlier revision: the proximity-gaps theorem [3] bounds by $(1 + \rho)/2$ the probability that a function far from the code passes a single query; DEEP and folding challenges from $\mathbb{F}_{p^2}$ contribute a Schwartz–Zippel term; grinding contributes g bits. Folding of arity 8 alters constants but not the form of the bound. Typical proofs are far below the ceiling (a bundle’s memory table has 2^{16} rows), which strengthens the field term. $\square$

Remark 5.14 (The choice of 80 queries). *Constraint sets 3 and 4 employed 27 queries, satisfying the conjectured 100-bit target ($3 \cdot 27 + 20 = 101$) at approximately 42 proven bits. The audit of 2026-09-12 (finding ZM1) determined that a value-bearing pool should rest on the proven bound and restored 80 queries. The cost is measured: a tier-10 proof grew from 435,529 to approximately 1,202,416 bytes with prove and first-verify times unchanged within noise (Section 6). A bundle is estimated at approximately 1.3 MB, so that a 4 MiB block accommodates three shielded transfers. The selected remedy is block-level aggregation rather than a reduced query count (Section 8).*

Theorem 5.15 (Per-primitive security of the deployed system). *Modelling Poseidon2 as a random permutation, BLAKE3 as a random oracle, Module-LWE/SIS at the Dilithium2 and ML-KEM-768 parameters as hard, and ChaCha20-Poly1305 as a secure AEAD, the security of the deployed system decomposes as in Table 4.*

Role	Primitive	Classical bits	Quantum regime	Q-bits
STARK transcript, Merkle, Fiat–Shamir	Poseidon2, 4 lanes ($\approx$ 256 bits)	128	BHT $2^{n/3}$	$\approx$ 85
Note commitments, nullifiers, tree nodes, hc, H_{IN} (binding)	Poseidon2, 256 bits	128	BHT $2^{n/3}$	$\approx$ 85
Note commitment hiding (r , 256 bits)	Poseidon2 sponge	128	Grover $2^{n/2}$	128
H_{IN} hiding (salt, 128 bits)	Poseidon2 sponge	128	Grover $2^{n/2}$	64
Nullifier unlinkability (nk, 256 bits)	Poseidon2 as PRF	128	Grover	128
Envelopes	ML-KEM-768 + ChaCha20-Poly1305	128	Module-LWE	128
Transaction identifiers, state and block hashes	BLAKE3-256	128	BHT	$\approx$ 85
Signatures (votes, blocks, mints, staking)	Dilithium2	128	Module-LWE/SIS	128
FRI soundness	$\mathbb{F}_{p^2}$, $q = 80$, $g = 20$	100 conj., 86 proven	combinatorial	same
Bridge attestations	secp256k1 ECDSA	128	Shor	0

Table 4: Security of the deployed primitives. The 384-bit transcript hash of the design is not implemented.

Corollary 5.16. *Excluding the bridge, the post-quantum security of the deployed system is*

$$\min\{85, 64, 128, 100\} = 64 \text{ bits},$$

determined by the width of the H_{IN} salt, or 85 bits if the hiding of call inputs is excluded; classically it is 100 bits conjectured and 86 bits proven, determined by FRI. If the FRI term is itself charged the square-root loss of Fiat–Shamir in the quantum random-oracle model (the internal analysis of 2026-09-26, not independently checked), the quantum soundness of the proof system is about 43 bits proven and 50 conjectured, and that becomes the minimum. Including the bridge, it is 0 bits against a quantum adversary.

5.4 Pool and ledger properties

Let Σ denote the set of ledger states and $\delta : \Sigma \times \text{Block} \rightarrow \Sigma$ the partial function computed by `apply_block` with the chain’s executor.

Theorem 5.17 (Determinism and atomicity). *$\delta(\sigma, B)$ is either undefined or a unique state computed identically by any two nodes running the same binary from the same genesis; and either every transaction of B applies in order and the root of Equation (2) equals the header’s, or the state is unchanged.*

Proof. Every input to δ resides in the block or the state; the executor’s verdict is a deterministic function of the proof bytes, the digests and the genesis-fixed FRI profile; the ledger employs ordered sets and a frontier whose root is a function of the appended sequence; deposit commitments are derived from published fields. `apply_block` applies to a scratch copy and commits only on success. The wallet’s entropy (salt, r , envelope keys) reaches the chain only through bytes consumed by the verifier. $\square$

Theorem 5.18 (Value balance of the pool). *Let $V(\sigma)$ denote the sum of the amounts of all unspent notes in σ , a quantity computable by no party. Under the soundness of Theorem 5.8, for every block B with $\delta(\sigma, B) = \sigma'$,*

$$V(\sigma') = V(\sigma) + \sum_{\text{deposits} \in B} \text{amount} - \sum_{\text{bundles} \in B} (\text{fee} + \text{burn}_A + \text{burn}_R),$$

where deposits comprise faucet mints, withdrawals and vesting claims (net of the base) and bridge deposits, and the sum is taken per asset.

Proof. By Theorem 5.8, the inputs of every accepted bundle are notes in the tree owned by the spender satisfying $a_0 + a_1 = a'_0 + a'_1 + \text{burn}_A$ in the private asset and $a_2 + a_3 = a'_2 + a'_3 + \text{fee} + \text{burn}_R$ in RAND; the nullifier set prevents double counting of an input; the four outputs are appended. Deposits append one note of public amount. No other action modifies the tree. $\square$

Corollary 5.19 (Supply audit). *With pool_value as in Equation (3), $V(\sigma) = \text{pool_value}(\sigma)$ whenever the sealed values of the genesis notes equal their declared amounts, and the invariant $\text{pool_value} + \text{register_total} + V_{\text{reg}} = \text{issued} - \text{slashed} - F_{\text{reg}}$ holds in every reachable state — on every chain through 16 with $\text{slashed} = F_{\text{reg}} = 0$. The chain verifies it by replay; the genesis terms are accepted from the file.*

Proposition 5.20 (Replay and double spending). *A note is spent at most once: its nullifier $H(\text{NF}, \text{nk}, \text{cm})$ is a deterministic function of the note and the owner’s key; admission step 6 rejects a spent nullifier; and two bundles in one block or one mempool spending the same note are rejected by the ledger and refused by the conflict index respectively. A bundle-less validator action is replay-protected by the register’s nonce, a faucet mint by its commitment, and an attestation by its digest.*

Proposition 5.21 (Bounded cost of rejection). *A transaction rejected at admission steps 1 to 7 costs a node $O(1)$ work plus, for a staking or bridge action, one signature verification or one guardian-quorum recovery. A transaction reaching step 9 costs one **Machine::verify** against the bundle key (approximately 16 ms warm); a call reaching step 10 costs a second verification against a cached key, or, at most once per (t, h_P, h_I, h_K) , a freshly constructed one.*

5.5 Consensus properties

Let V_e denote the validator set of epoch e with stakes w_e and total W_e , and let $\text{quorum}_e(S) \iff 3 \sum_{v \in S} w_e(v) > 2W_e$. Assume that within every epoch e the Byzantine members of V_e hold less than $W_e/3$.

Lemma 5.22 (Quorum intersection; uniqueness of certificates). *Two quorums of one epoch share an honest validator. Two valid QCs of the same view certify the same block.*

Proof. $w_e(S_1 \cap S_2) > W_e/3 > w_e(B_e)$. An honest validator votes once per view, its last voted view being fsynched before the vote is transmitted; a QC is verified against the set of the certified block’s epoch, and a block’s epoch is a function of its height, so two QCs of one view are counted in one set. $\square$

Theorem 5.23 (Safety). *If Byzantine stake is below one third in every epoch, every honest replica runs without restarting, and an honest replica’s lock is lowered only by locking on a newer certificate, no two honest replicas commit conflicting blocks at the same height.*

Proof sketch, following Yin et al. [11]. Suppose b commits through a three-chain $b \leftarrow b' \leftarrow b''$ with QCs in views $v, v+1, v+2$, and that a conflicting $\tilde{b}$ is certified at the least view $\tilde{v} > v$ at which any conflicting block is certified. Every honest voter for b'' processed b'' , whose justify

certifies b' , and thus set its lock to at least the QC on b' at view $v + 1$. Some honest h among them belongs to $\tilde{b}$'s quorum. For h to vote for $\tilde{b}$, the safety rule requires that $\tilde{b}$'s justify view exceed $v + 1$, i.e. be at least $v + 2$, or that $\tilde{b}$ extend b' ; the latter is excluded, and the former designates a block conflicting with b certified at a view in $[v + 2, \tilde{v})$, contradicting minimality unless it is b'' , in which case $\tilde{b}$ extends b . The consecutive-view requirement is what guarantees that every honest voter of b'' is locked at $\geq v + 1$. Epoch boundaries do not affect the argument: a block's set is a function of the parent branch derived identically by every replica from the same register snapshot (Theorem 5.17), and a QC is counted in the set of its block's epoch on every replica. $\square$

Remark 5.24 (Deviations affecting the theorem). *(i) The lock is not durable: `resume` resets it to the head QC and `extends_locked` returns true for an unknown locked block, so that a validator restarting between locking on b' and the commit of b may vote for a conflicting block; the vote-once-per-view state is durable, so the uniqueness lemma survives restarts. (Superseded since v0.5.1: the lock is restored on restart and on sync, an unknown locked block withholds the vote, and only a signed not-held quorum lowers it; the residual is that persistence is on vote, so a lock never voted under before a crash is lost — bounded by quorum intersection.) (ii) A validator that unbonds leaves the set at the next boundary while its stake remains bonded for two further epochs; in the absence of slashing or jailing, an equivocation is refused and logged while the validator keeps its stake and its seat, and the assumption of less than one third Byzantine stake concerns operators rather than incentives. (iii) An epoch whose register admits no validator carries the previous set forward. (iv) Addendum (v0.5.5–v0.6.2; the Lean verification of September 2026). The third hypothesis is the one the running rule does not meet. Since v0.5.4 a lock on a block the replica cannot obtain is lowered on signed not-held attestations (audit v4, CON-4) — since v0.5.5 from more than two thirds of the current set's stake, since v0.5.9 each word naming the signer's view and counting only when signed after the lock formed (CN-3). Sync commits only the three-chain prefix (CON-1a) and the lock survives restart and sync (CON-1b), both fixed in v0.5.1 and verified so by the audit v3 re-review. The release path itself is outside the theorem, and the Lean model of the rule as it runs exhibits an execution in which the tolerated Byzantine validators (eight of twenty-six) sign not-held words for a block they hold, three honest validators that were locked on it and lost it have their lock released, and two honest replicas commit conflicting blocks; one crash-restart lowers the honest participation needed to two, because the safety state is fsynced only when a vote is cast (`Action::PersistSafety`) while a not-held word is signed at the in-memory view (`HotStuff::not_held`), so a validator can attest, restart at a lower view, and vote for the block. The verification's remedy — persist the view before signing a not-held word, then either land a third safety rule refusing a vote across a released lock or remove the release path, and prove the resulting rule — is open at v0.6.2. No such execution has been observed on any chain.*

Theorem 5.25 (Liveness after GST). *After the global stabilisation time, with honest stake exceeding two thirds in every epoch and bounded message delay, the chain commits a new block within a bounded number of views.*

Proof sketch. Round-robin leadership within the epoch's set schedules an honest leader within $|V_e|$ views; doubling timeouts and `NavigationView` synchronisation bring honest replicas into a common view; every honest replica assembles the QC locally from gossiped votes, so that an unreachable next leader cannot withhold it; three consecutive honest views produce a commit. The fault tests on the fleet, and the cluster tests that register a fifth validator into the next epoch and unbond one out of it, exercise the epoch transitions. $\square$

5.6 Information disclosed by a transaction

Theorem 5.26 (Simulatability of a shielded transfer). *Under Theorem 5.11, the hiding of $H(\text{CM}, \cdot)$ in r , the pseudorandomness of $H(\text{NF}, \text{nk}, \cdot)$ in nk , and the IND-CCA security of ML-KEM-768 with ChaCha20-Poly1305, the public transcript of a committed transfer,*

$$(\text{anchor}, \text{nf}_{1,2}, \text{cm}_{1,2}, \text{fee}, 0, 0, \text{time}, \text{envelopes}, \pi),$$

is simulatable from $(\text{anchor}, \text{fee}, \text{time})$ and the lengths alone. In particular it discloses nothing concerning the sender, the recipient, the amounts, the change, the leaves spent, or whether an input is a dummy.

Proof sketch. Commitments are hiding in the 256-bit r ; nullifiers are pseudorandom in nk and, being functions of the spent commitment rather than of a public field, are unlinkable to any leaf; envelopes are ciphertexts under fresh keys; π is simulatable and, for the bundle guest, of constant shape. The public fee and time are excluded from the claim. $\square$

Corollary 5.27 (Disclosure by action). *A transfer discloses only its fee and time. A deployment additionally publishes the program. A call additionally publishes the program identifier, the tier, the eight outputs and H_{IN} , while concealing its inputs, trace and caller. A bond publishes the validator and the amount; an unbond and a withdrawal publish all of their fields; a mint, a genesis note, a withdrawal and a bridge deposit publish the amount at the boundary, after which the note is indistinguishable from any other. A bridge burn publishes the amount, asset, destination and relay fee.*

Five disclosures are structural. **Witness requests:** `rand_getWitness(index)` identifies to the queried node the leaf the wallet intends to spend; this is the principal disclosure of the wallet at this revision; the wallet-local tree that removes it shipped in v0.5.4, and a send no longer calls `rand_getWitness`. **Scanning** reveals to a node that scanning is occurring, not the ownership of any leaf. **The fee** identifies the transaction kind, since floors differ by action, and submission timing associates a bundle with the RPC clients connected at that moment. **Deposits** publish their amount once. **The program** is public and `hc` is unsalted.

5.7 Bridge properties

The quorum-unforgeability, emitter-binding, replay and rotation results of the account-based revision carry over unchanged, since `check_attest` is unchanged; each is restated in one sentence. An attestation accepted with fewer than $\lfloor 2n/3 \rfloor + 1$ signatures under honest keys constitutes an ECDSA forgery (strictly increasing indices, low- s normalisation, one recovery per signature). Attestation does not imply authorisation: without emitter binding any source-chain caller could mint; with binding indexed by the attestation’s own chain field, the attack fails. A digest enters the spent set before its effects apply, so an attestation is consumed at most once regardless of relay, and the mempool’s digest index excludes competing relayers from a single block. A rotation must be signed by the current set and advance the index by exactly one.

Proposition 5.28 (Chain-derived deposits). *Under collision resistance of $H(\text{CM}, \cdot)$, an accepted attestation for amount a and recipient hash ρ appends exactly one note of amount a and asset index $\text{idx}(\text{asset})$ owned by the address hashing to ρ , irrespective of the submitter’s published envelope.*

Proof. The ledger computes $\text{cm} = H(\text{CM}, \text{recipient.pk}, 0, a, \text{idx}, \text{time}, r)$ from the attested amount, the registry’s index and the action’s recipient (checked against ρ), blinding and time; the submitter’s envelope is not consulted. A hostile relay may seal an unusable envelope, but every field of the note is public in that transaction, so the recipient’s wallet reconstructs it from the block without decryption. $\square$

Theorem 5.29 (The bridge is not post-quantum). *Guardian public keys are recoverable from any accepted attestation and Shor’s algorithm [7] yields the corresponding private keys; a quantum adversary can therefore mint without bound. The custody counters that would bound the loss reside in the source-chain contracts, outside this code base.*

6 Evaluation

All figures are taken from the repository’s documentation, which identifies the machine and constraint set for each. No figure is a projection except where marked as an estimate.

Guest, tier	Proof size	Prove	First verify	Constraint set, machine
fib, tier 10	435 529 B	5.96 s	213.2 ms	set 4, 27 queries, laptop
fib, tier 10	1 202 416 / 1 195 120 B	6.05 / 5.81 s	232.7 ms	set 5, 80 queries, same machine
fib, tier 10	1 298 729 B	—	—	set 6, 80 queries (+8.0% over set 5)
fib, tier 12	460 441 B	22.91 s	809.2 ms	set 4, 27 queries
fib, tier 12	1 252 338 / 1 263 921 B	22.78 / 22.74 s	837.5 ms	set 5, 80 queries
fib, tier 12	1 359 978 B	—	—	set 6, 80 queries (+8.6%)
fib, tier 10, keccak table present	3 106 757 B	—	—	set 5; rejected by the 2 MiB cap
fib, tier 10, keccak table present	3 198 430 B	—	—	set 6; rejected by the 2 MiB cap
private_payment, tier 10	≈ 435 KB	7.4 s	—	set 4, 27 queries
bundle, tier 14	302 857 B	98.3 s	—	set 4, 27 queries, laptop
bundle, tier 14	≈ 1.3 MB (estimate)	≈ 100 s	—	set 5/6, not re-measured
rVM twin over one bundle proof, tier 19	327 321 B	1 707.7 s	18.31 s	M5.2, 16 queries, contended
rVM aggregate, N=1, tier 19	328 121 B	1 578 s	—	M5.3, 16 queries

Table 5: Measured proving and verification costs. Two sizes are reported where the source records two proofs, the hiding PCS altering the encoding by approximately one percent per run. A warm verification takes approximately 16 ms at either query count; the first verification is dominated by verifier-key construction. The rVM rows are the recursion VM’s test-profile exits (Section 4.14); the twin’s first verify is dominated by the program key build, and both proves ran on a machine executing other builds.

The query count affects size rather than computation: proof size increased by a factor of approximately 2.75 between 27 and 80 queries, while proving and first-verification times were unchanged within measurement noise. A shielded transfer therefore occupies approximately 1.3 MB, of which 99.7% is proof, and a 4 MiB block accommodates three transfers, compared with approximately nine at 27 queries. For reference, a Zcash Sapling transfer occupies approximately 2.7 KB and a Monero RingCT transfer approximately 2.2 KB: elliptic-curve proofs are two to three orders of magnitude smaller than a hash-based STARK over a machine trace with tables of 2,600 columns, and transparency and post-quantum security are purchased in bytes.

Test suite. At the chain-8 genesis build the workspace suite executed 466 tests across 27 binaries in 22 minutes. The constraint-set-6 re-vendor measured **678 passed, 0 failed, 6 ignored** in 43j minutes from a cold target (wallet flow 9 min 37 s; cluster 22 min 44 s over 18 tests; zkVM end-to-end 7 min 25 s including a tier-16 EVM-call proof). The block-aggregation merge adds its conformance suite (the pinned interface vectors byte-for-byte), the register/admission/sealing/sync tests and the sealed-resync capstone: the cluster suite is now 20 tests in

52 minutes, and the full workspace suite passes on the merged tree in approximately 1 hour 45 on a machine executing other builds. At the `v0.1` tag the release suite measured **947 passed, 0 failed, 14 ignored** across 58 binaries on a 48 GB machine (cluster 56 minutes, the capstone last and thrashing at 64 GB resident); on the security-fix branch every binary is green (core 198, node 174 unit, rVM and zkVM 519 passed and 13 ignored, the cluster suite 20 of 20 with the capstone alone at 2,113 s: register, prove, seal at height 46, prune, sealed resync), each fix behind a regression test that failed on `v0.1` first. The core crate contains a deterministic multi-replica HotStuff simulation with partitions, restarts and epoch transitions; the zkVM suite contains 111 cheating tests and the rVM suite 26.

Test networks. Chains 1 to 5 were account-based; chain 4 (constraint set 2) reached 31,952 blocks and chain 5 adopted constraint set 4. Chain 6 was the first shielded genesis (phase S1). Chain 7 (build `01dc23d`, S1 with S3) operated 18 validators at 100,000 RAND each across two laptops and sixteen DigitalOcean regions. Chain 8, generated on 2026-09-13 from build `03c9fb9`, is the first genesis incorporating S2 and constraint set 5: genesis hash `8c742fc9...`, 18 validators each at exactly the 1,000 RAND minimum (quorum 13 of 18), five genesis notes of 1,000 RAND, 23,000 RAND issued, epochs of 1,000 blocks, faucet enabled, production FRI profile, no bridge section. The fleet reached quorum on 2026-09-13 at 07:10:55Z and committed its first block 3.4 s later; as of 2026-09-14 the chain stood at height $\approx 46,800$ with all 18 validators active, epoch 46, an average block interval of 1.43 s, 331 transactions and 76 deployed programs. Fault tests recorded on earlier fleets: one validator of four halted (the chain continued); two halted (the chain halted and resumed without a fork); a RocksDB instance deliberately corrupted (detected, truncated and resynchronised to a byte-identical state); a dropped link (redialled within approximately 30 s). Chain 8 was retired on 2026-09-16 by two cuts in succession, both with the aggregation section left out (the cut script defaults to it) so that the section is enabled only once the production measurements of Section 4.14 are taken on the ≥ 64 GB batch machine. Chain 9 (genesis `dbb7498b...`, cut 13:29Z from build `5f8c6f9`: constraint set 6, the security review’s fixes, libp2p 0.57) reached quorum during the last batch of its rollout and ran for about an hour. Chain 10 (genesis `4d757f11...`, cut 14:04Z from build `a00c88c`) is the RAND chain: the same code under the renamed crates, binaries, RPC prefix, address prefix and hash domains, so a fresh chain by construction, with the same 18 validators at exactly the 1,000 RAND minimum (quorum 13 of 18), five genesis notes of 1,000 RAND, 23,000 RAND issued, epochs of 1,000 blocks, faucet enabled, production FRI profile, no bridge section. Because every peer identity changed with the rename, the two bootstrap nodes were cut over first, then the explorer host, the sixth droplet and the twelve regional validators, all sixteen in about twelve minutes, after which the fleet ran at about one block per second with sixteen peers each; the block explorer was redeployed against the `rand_*` methods in the same minute, and the retired chains’ binaries and data directories were removed from every droplet. One of the two NAT hosts remained unreachable at the time of writing and had not joined chain 10. *Addendum:* chain 10 was followed by chains 11 and 12 (2026-09-17, a short shielded address tried and reverted within the day), chain 13 (2026-09-19, the v0.4 call limits) and chain 14 (genesis `1cff3b7d...`, chain id 14, cut 2026-09-20 from build `b3c594c`): the v0.5 chain, with the bridge and token sections enabled, no aggregation section, and eighteen validators on keys generated off the repository for the cut. It reached height 248,953 by 2026-09-24, when the twelve 48 GB validators filled their disks (thirteen gigabytes of write-ahead log pinned by quiet column families beside a certificate of eighteen Dilithium2 signatures stored twice per block) and quorum was lost for thirteen hours; the recovery, the 256 MB log cap, the once-stored certificate and three resume rules are v0.5.5, rolled to all eighteen at once.

7 Implementation Status and Limitations

Area	Status at revision 26ab1d4 (addenda to d050259, 8e64781 and 4ab3575 marked)
Chain 16, v0.5.9–v0.6.2 (addendum)	The chain in operation (genesis 20925ae6..., live 2026-09-28 16:39 UTC on build 2c75e08, v0.6.1): 26 validators at 1,000 RAND, quorum 18; constraint set 7 — every LogUp terminal blinded in $\mathbb{F}_{p^2}$, every declared table floored at 2^7 rows, 32-bit range checks on input and public words and salt lanes, verifier keys salted (key_derivation_v2), POSEIDON2_LEN, the rVM’s in-circuit VERIFIER-1 — a hard fork whose build refuses chains 14 and 15; the branch-free bundle guest v2 (hc_bundle 651043e2...; bundle proofs 1,498,821 / 1,497,156 bytes at tier 14, BUNDLE_PUBLIC_LOG_HEIGHT 7) under hardening_v6 (pc window, canonical proof shapes, commit-phase PoW words, the call binding, uncallable deploys); the bridge at guardian set 1, burn sequence 7, 10 zUSD carried; no aggregation section. Before it: v0.5.9 (the 2026-09-27 rescan: fresh not-held words, gossip prechecks, sync budgets, a per-source-chain bridge replay floor), v0.5.10 (address fingerprints, randpay: links, the encrypted memo, a 1,860-byte envelope format for a later genesis), v0.5.11 (genesis vesting: a timelocked register with claim, revoke, bond and unbond, rand-state-6), v0.6 (the zkVM, rVM and aggregation fixes of the 27–28 September zk reviews; the 2^7 private-table floor as pool policy; aggregator slashing retired; rolled to all 26 hosts all-stop/all-start on chain 15) and v0.6.2 (opt-in delegated proving, Phase 1: a rand-prover the wallet’s owner runs, ML-KEM-768-sealed jobs, the spend key handed over — “delegating a proof is handing over custody”; not rolled). The formal-verification inventory of the papers (Lean 4, September 2026) confirmed RVM-1, RVM-2, ARITH-4 and INTERFACE-1 and found the not-held lock release outside the safety theorem (Remark 5.24(iv)).

Area	Status at revision 26ab1d4 (addenda to d050259, 8e64781 and 4ab3575 marked)
Chain 15, v0.5.6–v0.5.8 (addendum)	The chain in operation (genesis cc30e085... , live 2026-09-26 on build dd2ccbe): the same eighteen validator keys at 1,000 RAND each, quorum 13, joined on 2026-09-27 by eight more bonded under version-2 registrations and active from epoch 43 (26 validators, quorum 18), a consensus domain binding every signature to the genesis, the full gated staking section (faucet allowlist of 16 wallets, 10,000 RAND per-epoch faucet and entry budgets, a 3,333 bps weight cap, a two-epoch bond delay, version-2 registrations), the bridge at guardian set 1 (eight keys, quorum six) and burn sequence 7 under rules v2 (global cap 4,000 zUSD per 24 h), 10 zUSD carried from chain 14 as one genesis note, 8 MiB proofs and 20 MiB blocks, no aggregation section. v0.5.6 (internal deep scan): call headers pinned before key construction with calls capped at tier 14, keccak height $\leq 2^{12}$ and sha256 height $\leq 2^{13}$ (a validity rule, which refuses the tier-16 EVM calls of earlier chains); note values bounded below 2^{63} to match the guest's range check; inbound connection caps; an equivocation record that outlives tree eviction. v0.5.7: validator history pruning beyond 24 h. v0.5.8 (pre-release scan): orphans checked before they are kept, consensus gossip and sync requests metered per peer, votes beyond the eight-view window ignored and counted once per (view, voter), status read only from its author, blocking RPC tasks bounded. Aggregate binding (AGG-2), guest provenance (CS6-1) and the Poseidon2 constant table (ZKV-2) landed before the cut.
Chain 14, v0.5–v0.5.5 (addendum)	The chain in operation until 2026-09-26 (cut 2026-09-20): one four-slot hidden-asset bundle for every asset with the transaction binding in the public segment; the RPL token registry under rand-state-4 ; one zUSD over seven backings on the hardened bridge (B1–B4, F1), exercised by a mainnet round trip; three founder-led audit rounds (v3 2026-09-19, v4 2026-09-21, v5 2026-09-24) closed by v0.5.1 (durable lock, three-chain sync commits, fatal conflicting finality), v0.5.4 (one block per (view, leader), signed not-held release, disk guard, wallet-local tree) and v0.5.5 (not-held quorum, durable pending blocks, QC stored once, WAL cap, the stall's recovery rules); the timeout-certificate pacemaker, the aggregate-proof binding and the aggregation shape re-measurement remain open.
Wallet-local commitment tree	Not implemented at this revision. The wallet requests a Merkle witness from a node by leaf index, disclosing to that node the leaf about to be spent. <i>Addendum</i> : implemented in v0.5.4; the wallet builds its tree from the commitment pages it scans and a send never calls rand_getWitness .

Area	Status at revision 26ab1d4 (addenda to d050259, 8e64781 and 4ab3575 marked)
Block-level aggregation	Implemented and merged (2026-09-16), genesis-gated: the aggregator register, the Aggregate action, the nine-step admission with the byte-for-byte conformance suite, subsidy and fee split, sealing and pruning, sealed-form sync with coverage-closed serving, the RPC surface and the aggregate -watch daemon (Section 4.14); reviewed pre-v0.1 and fixed the same day (coverability and one-per-block as block rules, the proof-digest transaction identifier, the proposer's sweep). Chains 9 and 10 were cut without the section; activation on a later cut awaits the production measurements of admitted_shapes[0] on the ≥ 64 GB batch, and until then a block accommodates three shielded transfers. Binding the aggregate proof to (aggregator, nonce) is deferred to the next recursion milestone as defence in depth.
The recursion VM (M5)	Implemented (milestones M5.1–M5.4, merged upstream and vendored as randprotocol-rvm): the 26-instruction field-native ISA, emulator and DSL; the eight-instance batch machine with three measurement-gated row cuts to 1,968,619 rows per inner proof (from 5,682,847); the N -generic aggregate program; the proving-backend split (zero new kernels); the self-verifier, written with its measured requirement (est. tier 22, ≥ 128 GB). The production exit, the $N \geq 2$ aggregates and the self-proof are deferred to the big-machine runbook; the PTX build and production N re-measurement await a provisioned GPU node (80 GB device, ≥ 160 GB host).
Block rewards, issuance	Implemented with aggregation ($\text{subsidy}(n) = 100 \text{ RAND} \gg \lfloor n/210,000 \rfloor$ per sealed block, ≈ 42 M asymptote, payable to the covering aggregate). On every chain through 14, rewards remain proposer fees only.
Fee market	None. Floors with first-come, fee-then-hash ordering; demand above 1.5 transfers per second is rationed by arrival order.
Slashing and jailing	Aggregator equivocation only (two signed headers, one nonce). Validators incur only exclusion; the validator register has no jail flag and no evidence transactions exist. <i>Addendum:</i> aggregator slashing is refused since v0.6 (INTERFACE-1: an honest retry is indistinguishable from equivocation on the headers), so no slashing of any kind exists.
Nullifier accumulator	None. The nullifier root is recomputed per block in $O(n)$, adequate below approximately 10^6 nullifiers.
Envelope validity	Not proved in circuit. A sender may publish an envelope that does not decrypt to the committed note; the recipient's wallet does not treat it as received and the sender has expended value, as in Zcash. Deposit notes require no envelope, all their fields being public.
Bridge relayers	Unremunerated: the inbound relayer fee is recorded but not paid. The account-era ledger-level vector pass has not been reconstructed. A burn requires two proofs (<i>addendum:</i> one since chain 14).
Leader election	Round robin within the epoch's set; no beacon.

Area	Status at revision 26ab1d4 (addenda to d050259, 8e64781 and 4ab3575 marked)
Persistent program state, cross-program calls	Not implemented; a program reads private inputs and publishes eight words.
EVM and sBPF guests	Implemented upstream and vendored (M4.3/M4.4): the EVM interpreter guest (a tier-16 call proves in-suite; <i>addendum</i> : since v0.5.6 calls are capped at tier 14, so no EVM call is admissible on chain 15) and the sBPF interpreter guest reading its ELF from the public segment (1,753,945 $\rightarrow$ 694,498 cycles at constraint set 6, tier 20). The ERC-20 and SPL exit proofs await the ≥ 64 GB batch; chain-side EVM caller authorisation is open (design).
Public input segment	Constraint set 6 : the ninth mandatory table <code>public</code> , unsalted H_{PUB} in <code>pv[PUB_{0..7}]</code> , <code>READ_PUBLIC</code> , native <code>verify_public</code> ; the chain admits only the empty segment. <code>pv</code> count 26 $\rightarrow$ 34, <code>cpu</code> columns 224 $\rightarrow$ 275, verifier key a six-tuple.
Compiler toolchain	A flat-binary loader exists and compiled <code>fib</code> , <code>keccak256</code> , <code>evm</code> and <code>sbpf</code> guests prove; no build pipeline for user programs exists beyond the assembler and raw word files.
GPU proving	The rVM's proving-backend split exists (<code>Backend::Reference, Cuda</code>), mock-driven equivalence, zero new kernels; an 80 GB card runs every tier through 23 unchunked). No PTX has ever been built; hardware bring-up awaits the GPU node.
Zero knowledge	Statistical rather than perfect. <i>Addendum</i> : on chains 14 and 15 the proof bytes disclosed small committed tables and, for bundles, the real-slot pattern and leaf-index popcounts (INT-1/COV-2/INT-6, INT-2/GV-1); closed by v0.6 policy and by constraint set 7 on chain 16 (Remark 5.12).
Transcript hash	256-bit Poseidon2 rather than the 384-bit hash specified by the design; quantum collision resistance approximately 85 bits.
H_{IN} salt	128 bits, giving 64-bit quantum hiding.
hc hiding	None: a deployed program is identifiable by enumeration and two deployments of one program are linkable.
Poseidon2 round constants	Derived from a development seed; adoption of production constants is a pending change altering every digest and constituting a hard fork.
Lock durability	The HotStuff lock is discarded on restart (Remark 5.24). <i>Addendum</i> : durable since v0.5.1, released only on a signed not-held quorum since v0.5.5, fresh words only since v0.5.9; the release path is outside the safety theorem's hypothesis and the Lean model exhibits a conflicting commit through it (Remark 5.24(iv)); open.
Proof verification placement	Fixed (2026-09-14): admission-time verification runs on a four-worker blocking pool behind a 64-deep queue, with gossipsub application-level validation (report exactly once), an 8,192-entry refused-verdict cache and per-peer token buckets (burst 16, refill 4/s). Block-application verification remains on the consensus loop by decision (it changes when a vote is emitted).
Proving concurrency	Capped by a file-lock proving slot across test binaries and sessions; the cluster suite runs 3 s blocks so a contended proof cannot outlive its 256-block anchor.

Area	Status at revision 26ab1d4 (addenda to d050259, 8e64781 and 4ab3575 marked)
Constraint-set upgrades	Each constitutes a hard fork; six have occurred (chains 4 to 8, and constraint set 6 at the chain-9 cut). The transaction-identifier change of the security review (M1) was a further hard fork taken at the same cut; the RAND rename moved every hash domain and so made chain 10 a fresh chain as well.
Test keys	Test-network validator seeds are committed deliberately and must be rotated before any value-bearing deployment. <i>Addendum:</i> chain 14 runs on eighteen keys generated outside the repository, and the chain-13 key files are untracked (audit v3, OPS-1).
Audit	Three internal reviews: 2026-09-10 (node), 2026-09-12 (zkVM, identifying two critical soundness defects) and the pre-v0.1 review of 2026-09-16 over the 90 commits since the chain-8 genesis (four reviewers, one per crate group, each candidate re-traced by an adversarial verifier): one high (H1), one medium (M1) and one liveness finding (L1), all in the genesis-gated aggregation feature and unreachable on any deployed chain, fixed the same day; the zkVM and rVM crates had no findings; ten dependency advisories cleared by moving libp2p from 0.54 to 0.57. No independent audit at this revision. <i>Addendum:</i> three further founder-led rounds since (no independent audit at this revision either) — v3 (2026-09-19; POOL-1, the mint’s commitment unrelated to its amount, fixed as a hard fork at chain 14; CON-1a/1b and SYNC-1; RPC-1 witness-build metering; OPS-1 key rotation), v4 (2026-09-21; CON-3, CON-4, OPS-3, PRIV-1 and genesis-gated STAKE-2, the consensus domain and bridge rules v2) and v5 (2026-09-24; OPS-4, the not-held quorum, durable pending blocks, TOK-2) — each closed by a same-chain release.

Table 6: Discrepancies between the design and the implementation.

8 Planned Work

8.1 Block-level aggregation and the prover market

The remedy is implemented (Section 4.14); what remains is its activation and its hardware path. Activation is one measurement session on a ≥ 64 GB machine (the deferred runbook’s rows 1–6: the ERC-20 and SPL exit proofs, the rVM’s tier-21 production exit, the $N=2$ and $N=3$ test-profile aggregate twins, and the production $N=1$ aggregate), whose numbers fill the activating genesis’s `admitted_shapes[0]`; the zero-digest placeholder a cut ships is refused at genesis validation, so this is a hard gate by construction, and every cut through chain 14 was accordingly made with the section left out; since chain 14 the node also refuses a genesis carrying the section until the four-slot hidden-asset guest’s shape is re-measured, the admitted shape having been measured on the two-slot guest. The GPU path is the rVM’s CUDA backend: the split exists and the device model says an 80 GB card runs every tier through 23 unchunked, but no PTX has ever been built — the first build, hardware bring-up and the production N re-measurement all await a provisioned fleet GPU node (Linux, R580+ driver, CUDA 13, LLVM 21, `sm_80+`, ≥ 160 GB host). The step that raises throughput from three transfers per block to hundreds — bundles travelling from sender to aggregator without individual inclusion —

remains design-level, as does the self-verifier’s end-to-end proof (written; est. tier 22, ≥ 128 GB).

8.2 Remaining pool work

In the order listed by the repository: the wallet-local commitment tree, which removes the witness disclosure (shipped in v0.5.4); a fee-ordered mempool above the floor; a nullifier accumulator; in-circuit envelope validity; validator slashing and jailing with evidence transactions and a jail flag; production Poseidon2 constants; a salted hc should program identity require concealment; remuneration of bridge relayers; and reconstruction of the ledger-level bridge vectors for the pool.

8.3 zkVM milestones

Milestones 4.3 (the EVM interpreter guest) and 4.4 (SHA-256 and the sBPF interpreter guest) are delivered upstream and vendored, as is constraint set 6 (the public input segment). Milestone 5 (the recursion VM) is delivered through M5.4: the ISA, emulator, DSL and verifier program; the machine with its cheating suite; the N -generic aggregate program and chain-facing API; and the backend split with the self-verifier written. The remaining zkVM work is hardware-gated (the runbook above) plus chain-side EVM caller authorisation and a deeper storage index before an EVM contract holds real value.

9 Conclusion

The case for confidential computation is the case for a ledger with the confidentiality of a bank’s and the settlement properties of a public chain: balances that are not broadcast, trades that are not front-run, and a stablecoin transferable between two enterprises without public disclosure. Randprotocol’s reference full node realises such a ledger: a post-quantum-signed, BFT-finalised layer one on which value exists only as note commitments, a transfer is a proof that two concealed notes became two others, a program is deployed once and invoked with a proof in place of its inputs, stake is bonded from and withdrawn into the pool, and a bridge deposits external assets as notes derived by the chain itself. This paper has described that node component by component and analysed it as implemented: the bundle relation with its taint word and the call relation with its two in-circuit digests; a FRI profile of 80 queries that is 100-bit sound under the standard conjecture and 86-bit sound under the proven bound; a 256-bit transcript hash with 85-bit quantum collision resistance; a pool whose value balance is auditable without opening a note; and an epoch-aware chained HotStuff whose safety holds under a three-consecutive-view rule and a non-durable lock.

Transparency and post-quantum security are purchased in bytes: a transfer occupies approximately 1.3 MB and a block accommodates three. The remedy is now built: a recursion VM whose aggregate program verifies a window of bundle proofs in one recursive STARK, and the chain-side machinery around it — a bonded aggregator register, a subsidy paid in new issuance for coverage, and a pruning rule that keeps every historical bundle covered by a proof each node verified itself while shrinking a sealed transfer’s record to 3.3 KB. What remains is operational rather than conceptual: one measurement session on a ≥ 64 GB machine to enable aggregation on the next cut, one GPU node to open the production prover market, the wallet-local tree that would prevent a node from learning which leaf its client intends to spend, validator slashing, and an independent audit. Until these are complete, the system is a research test network and should be treated as such.

References

- [1] D. Suhubdy and A. Mohammed, “Randprotocol: Confidential Arbitrary Computations,” Draft 3, August 2026. `randprotocol.tex` in the same repository.
- [2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity,” ePrint 2018/046.
- [3] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, “Proximity Gaps for Reed–Solomon Codes,” in *FOCS*, 2020.
- [4] J. Don, S. Fehr, C. Majenz, and C. Schaffner, “Security of the Fiat–Shamir transformation in the QROM,” in *CRYPTO*, 2019.
- [5] G. Brassard, P. Høyer, and A. Tapp, “Quantum cryptanalysis of hash and claw-free functions,” in *LATIN*, 1998.
- [6] L. Grover, “A fast quantum mechanical algorithm for database search,” in *STOC*, 1996.
- [7] P. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *FOCS*, 1994.
- [8] L. Lamport, R. Shostak, and M. Pease, “The Byzantine generals problem,” *ACM TOPLAS*, 1982.
- [9] M. Castro and B. Liskov, “Practical Byzantine fault tolerance,” in *OSDI*, 1999.
- [10] E. Buchman, J. Kwon, and Z. Milosevic, “The latest gossip on BFT consensus,” arXiv:1807.04938.
- [11] M. Yin, D. Malkhi, M. Reiter, G. Gueta, and I. Abraham, “HotStuff: BFT consensus with linearity and responsiveness,” in *PODC*, 2019.
- [12] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling Byzantine agreements for cryptocurrencies,” in *SOSP*, 2017.
- [13] P. Daian et al., “Flash Boys 2.0: Frontrunning, transaction reordering, and consensus instability in DEXes,” in *IEEE S&P*, 2020.
- [14] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, “Order-fairness for Byzantine consensus,” in *CRYPTO*, 2020.
- [15] J. Bebel and D. Ojha, “Ferveo: Threshold decryption for mempool privacy in BFT networks,” ePrint 2022/898.
- [16] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, “Zcash protocol specification,” 2016 and revisions.
- [17] S. Noether, “Ring signature confidential transactions for Monero,” ePrint 2015/1098.
- [18] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, “Zexe: Enabling decentralized private computation,” ePrint 2018/962.
- [19] Z. Williamson, “The Aztec protocol,” 2018.
- [20] Aztec Network, “Aztec protocol documentation: Private execution environment (PXE) and fee abstraction,” 2025.
- [21] Namada, “Protocol specifications: Fee system and MASP,” 2025.

- [22] Zama, “The Zama protocol: Confidential blockchain computing with fully homomorphic encryption,” litepaper, 2025.
- [23] J. Groth, “On the size of pairing-based non-interactive arguments,” in *EUROCRYPT*, 2016.
- [24] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge,” ePrint 2019/953.
- [25] S. Bowe, J. Grigg, and D. Hopwood, “Recursive proof composition without a trusted setup” (Halo), ePrint 2019/1021.
- [26] S. Setty, “Spartan: Efficient and general-purpose zkSNARKs without trusted setup,” in *CRYPTO*, 2020.
- [27] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, “HyperPlonk: PLONK with linear-time prover and high-degree custom gates,” in *EUROCRYPT*, 2023.
- [28] U. Haböck, “Multivariate lookups based on logarithmic derivatives” (LogUp), ePrint 2022/1530.
- [29] A. Arun, S. Setty, and J. Thaler, “Jolt: SNARKs for virtual machines via lookups,” ePrint 2023/1217; S. Setty, J. Thaler, and R. Wahby, “Lasso: Unlocking the lookup singularity,” ePrint 2023/1216.
- [30] U. Haböck, D. Levit, and S. Papini, “Circle STARKs,” ePrint 2024/278.
- [31] Polygon Zero, “Plonky2: Fast recursive arguments with PLONK and FRI,” 2022; Plonky3 toolkit, version 0.7.0, 2026.
- [32] L. Grassi, D. Khovratovich, and M. Schofnegger, “Poseidon2: A faster version of the Poseidon hash function,” in *AFRICACRYPT*, 2023.
- [33] RISC Zero, “Scalable, transparent arguments of RISC-V integrity,” 2023; RISC Zero Security Model, 2024.
- [34] Succinct Labs, “SP1: A performant, general-purpose zkVM,” 2024; “SP1 Hypercube: Realtime proving of Ethereum blocks,” 2025.
- [35] 0xMiden (Polygon), “Miden VM: A STARK-based virtual machine,” documentation, 2024–2026.
- [36] Axiom, “OpenVM: A modular and extensible RISC-V zkVM,” 2025.
- [37] Matter Labs, “Boojum: A Plonkish arithmetization with FRI over the Goldilocks field,” 2023.
- [38] Scroll, “Euclid upgrade: Migrating to an OpenVM prover,” 2025.
- [39] EthProofs, “Real-time block proving dashboard,” ethproofs.org, accessed 2026.
- [40] Ethereum Foundation, “Shipping an L1 zkEVM #1: Realtime proving,” 2025.
- [41] NIST, “FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM),” 2024.
- [42] NIST, “FIPS 204: Module-Lattice-Based Digital Signature (ML-DSA),” 2024.

- [43] Wormhole Contributors, “Wormhole: A generic message-passing protocol between blockchains,” whitepaper and reference implementation, 2022.
- [44] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, “SoK: Communication Across Distributed Ledgers,” in *Financial Cryptography and Data Security*, 2021.
- [45] S.-S. Lee, A. Murashkin, M. Derka, and J. Gorzny, “SoK: Not Quite Water Under the Bridge: Review of Cross-Chain Bridge Hacks,” in *IEEE ICBC*, 2023.