The Alberta Buck - Notes Flow (v1.2)

How One BUCK Transfer Becomes N Unlinkable Notes

Perry Kundert

2026-06-10

One `Notes.mint` transaction takes 1,200 BUCK from Alice and splits it, atomically, into six notes of denominations `[1000, 100, 25, 25, 25, 25]` – and the public chain learns nothing about who received what. This document walks that example end to end, executably: the SNARK-verified batch mint that folds the note commitments into the on-chain Merkle tree, the off-chain delivery and offline verification of each recipient's opening, and the three *Identity-M* redemption flows – `spendCoupledA2` (addressed, private issuer), `spendCoupledA1` (addressed, public issuer), and `spendCoupledB1` (bearer, public issuer) – by which a note is deposited by proving, in zero knowledge, both *which Identity may spend it* and *that the Identity is a registered member* of the on-chain accumulator.

The walkthrough demonstrates five concrete guarantees – atomic value conservation, recipient-blinded commitments, per-note disclosure to recipients only, offline receipt reconstruction from the transaction ID alone, and soundness against an arbitrarily hostile issuer-side EVM and prover – and audits the privacy surface step by step: each party can *name* its counterparty under compulsion, while no third party can name either, or link mint to spend. Every cryptographic step is exercised by the shipped `alberta_buck.wallet` code in one seeded, deterministic session, with the output captured in place. (PDF, Text)

Contents

1 Introduction

This document is the *narrative and executable* companion to alberta-buck-notes.org (architectural overview; together they are the two living Notes references) and alberta-buck-ethereum.org (Solidity surface). It walks one concrete example – Alice mints a 1,200-BUCK transfer into six notes of denominations `[1000, 100, 25, 25, 25, 25]` – end-to-end through the SNARK-verified batch-mint pipeline, and then walks the *inverse* flows: A2 deposit (Bob redeems his identity-targeted note via the deposit-coupling sigma + membership proof) and B1 deposit (Carol redeems a bearer note while the issuer learns who cashed it). Every actor, every artifact, and every check is named.

The mint pipeline uses SNARK-verified batch minting: the circuit takes `oldRoot` as a public input, folds N commitments into the rolling tree using witness sibling paths, and emits a SNARK-attested `newRoot` for the contract to record. On-chain mint cost is essentially constant in N – ~380K plus ~500 gas per leaf of calldata.

The *identity-M* spend model makes the KYC-bound Identity point M, not the EOA, the authority axis (the notes-identity-axis model). Every spend is one gadget – an EIP-196 Okamoto sigma producing a committed point P, plus a Poseidon-Merkle membership proof *bound to that same* P (`identity_membership_g1tie.circom`, ~350K gas on chain) – instantiated three ways: `Notes.spendCoupledA2` (addressed, private issuer, `verifyDepositCoupling`), `spendCoupledA1` (addressed, public issuer; the recipient's own Identity is committed), and `spendCoupledB1` (bearer, public issuer, `verifyDepositorBinding`). The membership closes the collusion gap: a note whose committed Identity decrypts to a non-member reverts at spend. All three are wired and tested end to end (`alberta_buck.wallet.unilateral_a2`, `.unilateral_a1`, `.b1_binding`; `IdentityRegistry.verifyDepositCoup` `.verifyDepositorBinding`; the G1-tie membership verifier; the on-chain incremental `identityRoot`). A fourth gate completes the addressed spends: the note $\leftrightarrow$ *eEnc tie* (`circuits/note_binding.circom` for A2 and `note_binding_a1.circom` for A1, behind one `INoteBindingVerifier`) proves the deposit-coupling ciphertext is keyed to the identity material *the spent note committed in its* $idHash$ – A2: a re-encryption, under M_{rec}, of the committed $eIss$; A1: keyed to the same M_{rec} the note's $eNote$ was addressed to, the spend's public `face` pinning the plaintext – enforcing addressed-binding (a stolen opening cannot be redeemed with a self-addressed $eEnc$) and closing A2 collusion against ciphertext substitution (*Proofs*, Theorem 12). Only a production Powers-of-Tau ceremony remains before deployment.

The cryptographic walkthroughs in this document are *executable*: the Python blocks run against the shipped `alberta_buck.wallet` modules in one seeded, deterministic session, and their printed output is captured below each block.

The aim is to show, by walking the path, that the system delivers five concrete guarantees:

1. **Atomic conservation** – if the transaction succeeds, the BUCK balance leaves Alice exactly once and matches the public total of the new notes. If the proof fails, no BUCK moves and no commitments are appended.

2. **Issuer-bound, recipient-blinded artifacts** – the public commitments name the issuer (`msg.sender`) but reveal nothing about per-note values or recipients.

3. **Per-note disclosure to recipients only** – each recipient receives an *opening* off-chain that lets them reconstruct exactly their own note's value and the issuer's verifiable identity, plus the fact that N siblings exist whose hidden values sum to `totalFace` – and nothing more.

4. **Offline receipt reconstruction** – given only the mint transaction ID, anyone holding an opening can independently verify the note against the chain, with no help from Alice or any indexer.

5. **Adversarial-prover soundness** – even if Alice runs a maliciously patched EVM and an arbitrarily hostile witness generator, the honest validators executing the deployed `Notes.sol` and `MintGroth16Verifier.sol` reject every batch that does not satisfy the mint relation. The on-chain truth is what governs.

The one structural caveat (the pinned-N dispatch set under `mint_batch.circom`) is addressed at the end and cross-referenced into the proofs and implementation docs.

2 Cast of Characters

The example uses concrete names so the message flow is easy to track:

- **Alice** – the issuer. She holds 1,200 BUCK and wants to break it into six notes. She is registered with the IdentityRegistry so that her `msg.sender` address resolves to a cryptographically attested identity.
- **Bob, Carol, Dave, Eve, Frank, Grace** – six recipients. Bob will get the 1,000-BUCK note; Carol the 100-BUCK note; Dave/Eve/Frank/Grace one 25-BUCK note each. Recipients do *not* need to be on-chain known to Alice – the recipient identity binding is a field inside the commitment, not an on-chain registration step at mint.
- **The honest EVM** – every Ethereum validator running stock client software. This is the only EVM whose verdict matters. It executes the deployed `Notes.sol`, `MintGroth16Verifier.sol`, `MintVerifierAdapter.sol`, and `Buck.sol` bytecode.
- **Alice's hostile EVM (Eve-EVM)** – a fictional entity Alice can run locally. She may patch any contract logic she wants, run any witness generator, and produce any proof bytes. The point of the design is that nothing she runs locally affects what the honest EVM accepts.
- **A passive observer (Mallory-the-watcher)** – someone scanning the public chain after the fact, trying to learn *which* note went to *whom* for *how much*. Mallory has full chain history and unlimited compute.

3 The Promise

By the end of the mint transaction Alice will have:

- A single Ethereum transaction `T_mint` on-chain whose effects are:
  - 1,200 BUCK debited from Alice and credited to the `Notes` pool address.
  - A batch of `N*` opaque `uint256` commitments (here `N* = 16`: six live plus ten dummies) folded into the note Merkle tree, moving the root from `oldRoot` to the SNARK-attested `newRoot`, and the tree size from `nextLeafIndex` to `nextLeafIndex + N*`.
  - One `Minted(issuer=Alice, totalFace=1200e18, startIndex=K, count=N*, newRoot)` event; no per-leaf `Appended` events – the commitments themselves live in the transaction calldata.
  - One `Transfer(Alice, Notes, 1200)` ERC-20 event.
  - One Groth16 proof (256 bytes, shape-independent of N) in calldata, accepted by the on-chain per-N verifier.

- Six off-chain artifacts, one per *live* recipient, each containing the *opening* of exactly one commitment plus the mint-transaction reference. The ten dummy openings are discarded or retained privately by Alice as anonymity- set filler.

The key invariant: looking at the chain alone, Mallory sees that Alice escrowed 1,200 BUCK and folded `N*` hashes into the tree. She cannot tell which hash hides which value, who any recipient is, or even whether two of the recipients are the same person – or which entries are live notes vs. zero-value padding. Looking at his off-chain artifact, Bob sees his 1,000 BUCK is real, was minted by Alice in `T_mint`, and that the *live* siblings among the other fifteen have hidden values summing to exactly 200 BUCK (the rest are `v = 0` padding he cannot distinguish from live notes by chain inspection alone) – but he does not learn what those values are or who holds them.

4 What's Inside a Note (Conceptually)

A "note" is just a five-tuple known by Alice and the recipient:

```
opening_i = ( flavor_i, v_i, rho_i, idHash_i, predicate_i )
cm_i      = Poseidon5( flavor_i, v_i, rho_i, idHash_i, predicate_i )
```

- `flavor_i` – which note variant (A1 identified cheque, A2 private cheque, B1 bearer cheque); see Notes for the menu.
- `v_i` – the face value in BUCK base units (18-decimal scaled integer).
- `rho_i` – a fresh, uniformly random nonce. This is the source of unlinkability: swapping `rho_i` changes `cm_i` to a uniformly fresh hash even if every other field is identical.
- `idHash_i` – a Poseidon-hashed handle on the recipient's identity (e.g. a Poseidon-hash of the recipient's registered ElGamal credential, or zero for a pure bearer note).
- `predicate_i` – a domain-separating tag for any optional spending condition (escrow, time-lock, etc.). Zero in the simplest case.

The Poseidon-5 hash gives the commitment two cryptographic properties that make the whole flow work:

- **Hiding**. Without `rho_i`, the commitment leaks no information about the other fields. Mallory sees `cm_i` and gains nothing about `v_i`, `idHash_i`, or `flavor_i`.
- **Binding**. No prover, hostile or honest, can find two distinct openings hashing to the same `cm_i` (collision-resistance of Poseidon on BN254, modelled as a random oracle in alberta-buck-proofs.org Part IV).

The on-chain contract stores only `cm_i`. Openings live exclusively off-chain.

5 The Mint, Step by Step

5.1 Step 0: Pre-flight (Off-Chain, In Alice's Wallet)

Alice's wallet has done all the slow work before any transaction is broadcast:

- She owns 1,200 BUCK on-chain.
- She has decided on the denomination split: `[1000, 100, 25, 25, 25, 25]`. N=6. The deployed mint pipeline pins a small set of batch sizes – recommended {16, 128, 1024} – one circom circuit and one verifier per pin. Alice's wallet rounds her requested split up to the smallest pin `N* >= 6`, here `N* = 16`, and pads with `16 - 6 = 10` "blinded zero" openings whose `v_i = 0`. The padding leaves cost the same SNARK constraints as live notes but contribute zero to `totalFace` and are spendable for zero BUCK; in effect they are anonymity-set filler whose openings Alice may discard. The narrative below tracks the six live notes and ignores the ten dummies (they show up in the on-chain commitments[] vector alongside the live ones, indistinguishable to Mallory).
- She has chosen an `idHash_i` for each recipient (typically a Poseidon hash of the recipient's IdentityRegistry-bound ElGamal credential, derived once at out-of-band introduction time).
- Her CSPRNG produces a fresh `rho_i` nonce per leaf – live or padded.

She now has 16 opening tuples in memory: 6 live, 10 dummies.

5.2 Step 1: Build Commitments

For `i = 0..15` (six live, ten dummies) her wallet computes

```
cm_i = Poseidon5( flavor_i, v_i, rho_i, idHash_i, predicate_i )
```


This is pure off-chain Poseidon arithmetic in `alberta_buck.wallet.notes`. Sixteen `uint256` field elements drop out. Their order in the eventual on-chain array is fixed by Alice's local list – the contract preserves whatever order she submits, and the SNARK assumes leaves land in the tree at indices `[nextLeafIndex, nextLeafIndex+1, ..., nextLeafIndex+N-1]` in submission order.

Alice's wallet must also keep a *local mirror of the rolling tree state*. The Tornado-style Merkle tree the contract maintains is parameterised by the standard `filledSubtrees[d]` for `d = 0..TREE_DEPTH-1` – the rightmost partially-built node at each level. The wallet snapshots this state from the chain (or replays it from the mint event log) immediately before assembling the proof, then derives, for each of the 16 leaves *in submission order*, the depth-20 sibling sequence the in-circuit insertion will consume. That derivation is the one delicate piece of wallet-side mint code; everything else is straightforward Poseidon arithmetic.

5.3 Step 2: Generate the Batch-Mint Groth16 Proof

Alice's wallet runs `scripts/snark/prove_mint_batch.js`. The `mint_batch.circom` circuit at her chosen pin `N* = 16` takes a witness that covers not just the openings but the tree update itself, because the Merkle insertion lives *inside* the SNARK.

Public signals (circom emits the circuit OUTPUTS first, then the public inputs; arity `2N+4`):

```
issuerMode[i]   # i in 0..N-1: per-leaf PUBLIC=1 / PRIVATE=2 (a circuit OUTPUT,
                #   the deterministic flavor projection -- Notes.mint gates on it)
oldRoot         # the live note root she read from chain
newRoot         # the root after folding her N leaves
nextLeafIndex   # the live tree size she read from chain
totalFace       # 1200e18 -- range-bounded to 128 bits
cm[i]           # i in 0..N-1: the commitments, exposed DIRECTLY (not a batch hash)
```

The commitments are exposed as public inputs directly (no batch digest): each input costs ~6K gas in the EVM verifier, cheaper than recomputing a keccak over calldata on-chain, and far cheaper than an in-circuit Poseidon over `N` elements. (A2 / private-issuer batches use a *separate* circuit, `mint_batch_a2`, that exposes each leaf's `eIss` instead of `issuerMode`; see *The Redemption* and the Decryptability companion.)

Private witness (per-leaf and per-level):

```
For i in 0..N-1:
  flavor[i], v[i], rho[i], idHash[i], predicate[i]    # the opening
  pathSiblings[i][d] for d in 0..TREE_DEPTH-1         # 20 siblings per leaf
```

The `pathSiblings` for leaf `i` are derived by the wallet from its local `filledSubtrees` mirror plus the previously-folded leaves of *this same batch* (leaves `0..i-1` sit at known positions and contribute their Poseidon hashes to `i`'s path on the right). The wallet does this work; the chain consumes only `oldRoot`, `newRoot`, and the calldata `cms[]`.

In-circuit constraints:

1. **Per-leaf opening.** For each `i`, `cm[i] === Poseidon5(flavor[i], v[i], rho[i], idHash[i], predicate[i])`. N Poseidon-5 evaluations, N equality constraints.
2. **Range bounds.** Each `v[i]` and `totalFace` passes through `Num2Bits(128)`.
3. **Sum conservation.** `sum(v[0..N-1]) === totalFace`.
4. **Rolling Merkle insertion.** The circuit threads `oldRoot` and `nextLeafIndex` through N successive single-leaf insertions – the standard Tornado-style incremental-tree fold, performed in-circuit: at each step it folds `cm[i]` up the tree using `pathSiblings[i][*]` and `Switcher(...)` per level, advances the rolling state, and finally asserts the resulting root equals `newRoot`. N x TREE_DEPTH Poseidon-2 hashes (~4.3K constraints per leaf).
5. **Flavor range + issuer mode.** Each `flavor[i] \in {A1=1, A2=2, B1=3}`, and `issuerMode[i]` is the deterministic projection (A2 → PRIVATE, else PUBLIC), exposed as a public output. This is what makes a bearer leaf provably public-issuer (and a private-issuer bearer note unmintable – see the Decryptability companion, *Why B2 is impossible*).

Constraint counts work out to roughly **21,800 R1CS per leaf** (opening + range-bound + dual Merkle walk: one path consumed to attest `oldRoot`, one to advance `newRoot`); the shipped pinned-N set measures 21,914 R1CS at N=1, 348,705 at N=16, and 697,281 at N=32. The doubling per pin holds linearly through the projected sizes (N=128 ~2.8M R1CS, N=1024 ~22M). At N=16 the 2^19 PTAU and ~6 s prover wall on a dev laptop are comfortable; at N=128 it is rapidsnark-and-minutes; at N=1024 it is hours plus a multi-GB zkey. The `scripts/snark/setup.sh` ceremony emits one verifier per pin.

The witness is processed by snarkjs's Groth16 prover. Tens of seconds to minutes later the wallet has a fixed-size proof object `(pi_A: G1, pi_B: G2, pi_C: G1)`, encoded as ABI-static `(uint256[2], uint256[2][2], uint256[2])` – exactly 256 bytes regardless of N.

Observation. The Groth16 proof reveals *nothing* about the witness. Soundness guarantees that no `proofBytes` can satisfy the verification equation unless a witness exists; zero-knowledge guarantees that the verification equation reveals nothing about which witness was used. Mallory, even with the proof in hand, cannot back out `v[i]`, `rho[i]`, `idHash[i]`, `flavor[i]`, `predicate[i]`, or the per-leaf Merkle paths – only `oldRoot`, `newRoot`, `nextLeafIndex`, `totalFace`, the `cm[i]`, and the per-leaf `issuerMode[i]` (flavor-class: public vs private, no identity) leak.

5.4 Step 3: Submit `Notes.mint` On Chain

Alice now broadcasts a single transaction `T_mint` calling

```
// Minting is gated-only: a nameable issuer Identity is bound to every batch.
// Alice (a public Corporate Identity here) uses the PUBLIC overload:
notes.mint(
    proofBytes,
    oldRoot, newRoot, nextLeafIndex,
    1200e18,                        // totalFace
    [cm_0, cm_1, ..., cm_15],       // cms.length === N* (here 16)
    [PUBLIC, ..., PUBLIC],          // issuerMode (a SNARK public input of mint_batch)
    issuerSig                       // Schnorr by Alice's registered key over keccak(cms)
);
// A private (A2) issuer uses the other overload instead:
//   notes.mint(..., cms, issuerMode, a2Bindings)  // routes the proof to mint_batch_a2
//                                                  //   and ties each leaf's eIss
// There is no unbound mint path -- an unnameable note cannot be created.
```

The honest EVM walks `Notes.mint` (the PUBLIC overload):

```
// Stale-state guards + field bounds (_verifyMintOrRevert).
require(cms.length > 0,                            "Notes: empty mint");
require(issuerMode.length == cms.length,           "Notes: issuerMode/cms length");
require(oldRoot       == roots[currentRootIndex],  "Notes: stale oldRoot");
require(nextLeafIndex == self.nextLeafIndex,       "Notes: stale nextLeafIndex");
require(newRoot != 0 && newRoot < FIELD_R,         "Notes: bad newRoot");

// SNARK verify: the proof binds issuerMode + cm[] + the roots.  The
// MintVerifierAdapter dispatches by cms.length to the per-N Groth16 verifier;
// public-signal vector is [issuerMode[0..N), oldRoot, newRoot, nextLeafIndex,
// totalFace, cm[0..N)] (outputs lead), arity 2N+4.
require(mintVerifier.verifyMint(
            proofBytes, issuerMode, oldRoot, newRoot, nextLeafIndex, totalFace, cms
        ),                                         "Notes: bad mint proof");

// Public-issuer binding (gated-only): every leaf must be PUBLIC-mode,
// msg.sender must be a registered PUBLIC Identity, and issuerSig must be a valid
// Schnorr over keccak256(cms).  (The PRIVATE-mode overload instead routes the
// proof to mint_batch_a2 and verifies one A2 re-encryption binding per leaf,
// tying each committed leaf's eIss.)
require(identityRegistry.isPublicIdentity(msg.sender),
                                                   "Notes: public-mode leaf needs public issuer");
require(identityRegistry.verifyIssuerSchnorr(
            msg.sender, keccak256(abi.encodePacked(cms)), issuerSig),
                                                   "Notes: bad issuer binding");

// Pull BUCK only after the proof is accepted, then advance (_advanceAndPull).
require(buck.transferFrom(msg.sender, address(this), totalFace),
                                                   "Notes: transfer failed");
currentRootIndex = (currentRootIndex + 1) % ROOT_HISTORY_SIZE;
roots[currentRootIndex] = newRoot;
self.nextLeafIndex     += cms.length;
noteFaceSum            += totalFace;

// Single batch event; cms[] live in calldata.
emit Minted(msg.sender, totalFace, /*startIndex=*/ nextLeafIndex,
            /*count=*/ cms.length, newRoot);
```

Five things happen in strict order, and **all of them are atomic** (one transaction, one EVM state transition):

a. **Stale-state guards.** The contract asserts that `oldRoot` equals the current ring-buffer head and `nextLeafIndex` equals the live tree size. Either fails if another mint landed since Alice's wallet snapshotted state (more on this under **Prover Contention and the Rollup Race**

below). b. **Calldata binding.** The `cms[]` are exposed **directly** as Groth16 public inputs (no batch digest), alongside the per-leaf `issuerMode[]` the circuit emits, so any tampering with `cms[]` or `issuerMode[]` en route from the prover to the chain mismatches the SNARK and the verify call rejects. c. **SNARK verify.** The `MintVerifierAdapter` dispatches by `cms.length` to the per-N verifier (one circuit per pin, one verifier contract per pin), unpacks the 256-byte `proofBytes` into Groth16 form, builds the `2N+4`-entry public-signal vector `[issuerMode[0..N), oldRoot, newRoot, nextLeafIndex, totalFace, cm[0..N)]` (circuit outputs lead), and invokes the appropriate `MintBatchN${N}Groth16Verifier`. If the verifier returns false, the entire transaction reverts – nothing is written, no BUCK moves. d. **BUCK escrow.** `buck.transferFrom` debits `totalFace` from Alice and credits the `Notes` pool. Allowance-driven pull, gated by Alice's prior `approve(notes, 1200e18, E_notes, pi_CP)`. This CP-bound approve is mandatory: Alice is a private EOA, and mutual decryptability requires a per-pair CP fragment for every private party (the `identityHash` fallback is only valid for Public-Identity contracts). The pool was bound under a Public Identity (`bindContract(notes, pk, E, true)`), so the pool side falls back to `_identityHash(notes)` without requiring a fragment from the pool. The Notes operator decrypts Alice's CP ciphertext from the `IdentityExchange` event under subpoena. e. **Root advancement.** The contract writes `newRoot` into the next ring-buffer slot, advances `currentRootIndex` and `nextLeafIndex`, accumulates `noteFaceSum`, and emits a single `Minted(issuer, totalFace, startIndex, count, newRoot)` event. The new root immediately becomes the live root that the next minter must read.

If any of (a)-(e) fails – stale root, stale leafIndex, bad proof, insufficient BUCK, missing approval – the entire transaction reverts and the chain state is indistinguishable from no submission at all. **Alice gets one shot per transaction**; she cannot retry a partial state.

Note on duplicate commitments. The contract carries no `commitmentExists[cm]` duplicate check – not because duplicates are harmless, but because they are *economically self-punishing*: if Alice submits the same opening twice (whether across two batches or twice in one), she pays `totalFace` on each occurrence, the leaves land at distinct tree indices, but the spend nullifier `Poseidon3(rho, idHash, 4242)` is *deterministic in the opening*. After the first spend nullifies `cm`, the second copy is unspendable forever. Net result: Alice burns BUCK she could have kept. The chain's job is value conservation, not saving Alice from a buggy `rho` RNG.

5.5 Step 4: Off-Chain Note Delivery

After `T_mint` confirms, Alice transmits each live opening to its intended recipient by any out-of-band channel: an encrypted message, a printed QR code, a USB stick. The payload to recipient `i` is:

```
note_artifact_i = {
  txHash:    T_mint,                # Ethereum transaction ID
  leafIndex: startIndex + i,        # absolute position in the commitment vector
  cm:        cm_i,                  # for cross-checking against calldata
  opening:   ( flavor_i, v_i, rho_i, idHash_i, predicate_i ),
  meta:      { issuer: Alice's IdentityRegistry handle, batchSize: N*, ... },
}
```

The chain itself never sees `opening`. `cm_i` is already folded into `newRoot`; the artifact carries it explicitly only as a convenience so the recipient does not have to recompute it before locating it in the batch's calldata `cms[]`.

The ten dummy openings Alice generated to pad to the pinned N=16 circuit receive no artifact; they are discarded (or, at Alice's option, retained privately as unspendable anonymity-set filler).

From the recipient's perspective the presence of dummies is invisible: Bob's artifact only names his own leaf, and the batch total `totalFace = 1,200 BUCK` is exactly the sum of the live `v_i`.

5.6 Step 5: Recipient Verification (Offline, Independent)

Bob receives his artifact for the 1,000-BUCK note. Without trusting Alice and without any help from an indexer, he runs:

1. **Recompute and check.** Bob computes `cm_check = Poseidon5(flavor, 1000e18, rho, idHash, predicate)` and asserts `cm_check == cm`. If this fails, Alice lied about the opening; Bob discards the artifact and demands a real one. (Binding of Poseidon makes substitution impossible.)

2. **Look up the mint transaction.** Bob reads `T_mint` on-chain, parses the call data into `(proofBytes, oldRoot, newRoot, nextLeafIndex, totalFace, cms[])` and reads `msg.sender` from the transaction envelope.

3. **Confirm `msg.sender` is Alice.** Cross-check `msg.sender` against the IdentityRegistry: it must resolve to Alice's PS-credentialed identity. If Bob wanted an A1 identified cheque, the issuer field of his opening must match.

4. **Confirm calldata membership.** Assert `cms[leafIndex - nextLeafIndex] == cm` (where the artifact's `leafIndex` is the absolute position). The chain's acceptance of `T_mint` confirms that every entry in `cms[]` was folded into `newRoot` and that `newRoot` was added to the root ring buffer. Bob therefore has chain-anchored evidence that `cm` is officially a commitment in the pool.

5. **Confirm batch totals.** Read `totalFace` from the call data. The chain's acceptance of `T_mint` is itself a proof that the Groth16 verifier accepted – which means `sum(v[0..N-1]) === totalFace` holds in the witness. Bob does not need to re-verify the proof; the EVM already did. But if he wants belt-and-suspenders, he can re-run `MintGroth16Verifier.verify(proofBytes, [issuerMode[0..N), oldRoot, newRoot, nextLeafIndex, totalFace, cm[0..N)])` locally against the same on-chain bytecode, assembling the public-signal vector from the calldata exactly as the contract does.

After (1)-(5) Bob knows:

- His note encodes exactly 1,000 BUCK. (Step 1.)
- It was minted in `T_mint` by Alice. (Steps 2, 3.)
- It is one of `N*` (here 16) commitments folded into the tree by that mint, the others being `cms[] \ cm`. (Step 4.)
- The *live* values among the other siblings sum to exactly `1200 - 1000 = 200` BUCK (the residual of `totalFace` once Bob subtracts his own `v`); the batch may also contain dummy openings with `v = 0`. (Step 5 + the per-note opening relation: he knows *his* value, the chain attests to the *sum*.)
- He learns nothing else about the other notes – not their individual values, not their recipients, not their flavors, not which are live and which are padding. Each `cm_j` is hiding under a fresh `rho_j` he does not have.

This is the unlinkability promise made concrete: each recipient can pin down their own slice and the batch total, without learning anything about siblings.

6 Why Alice Cannot Cheat (Even With a Hostile EVM and Prover)

Alice runs whatever software she wants on her laptop. She can fork the snarkjs witness generator, patch the circom compiler, run Anvil with a malicious `MintGroth16Verifier` that always accepts. None of it matters, because the truth on the public chain is determined by what *the honest validators* run. Three pillars prop up the soundness story:

6.1 Pillar 1: The On-Chain Verifier Is Fixed Code

`MintGroth16Verifier.sol` (one per pinned N) was emitted from the Groth16 verification key produced by the trusted setup (production ceremony pending; dev-only entropy in `scripts/snark/setup.sh` today). Its bytecode is deployed once and immutable. Validators run whatever bytecode sits at that address; Alice cannot tamper with it without consensus-breaking the chain.

Knowledge soundness of Groth16 means: for every `proofBytes` the verifier accepts on public inputs `[issuerMode[0..N), oldRoot, newRoot, nextLeafIndex, totalFace, cm[0..N)]`, there exists a witness `(flavor[i], v[i], rho[i], idHash[i], predicate[i], pathSiblings[i][*])_i` such that:

- `cm[i] = Poseidon5(flavor[i], v[i], rho[i], idHash[i], predicate[i])` for all `i`
- each `v[i]` and `totalFace` fit in 128 bits and `sum(v[i]) = totalFace`
- folding `cm[0..N-1]` into the tree rooted at `oldRoot`, starting at position `nextLeafIndex`, using `pathSiblings[i][*]` as the right-of-insert context, yields exactly `newRoot`
- each `issuerMode[i]` is the deterministic flavor projection (A2 → PRIVATE, else PUBLIC) of the witness `flavor[i]`.

Alice is free to *find* such a witness any way she likes – she's the prover. But she cannot find a witness for a *false* statement; that's what soundness prohibits.

6.2 Pillar 2: BUCK Escrow Is Inside the Same Atomic Call

The proof check, the BUCK pull, and the root advancement all live in the same `Notes.mint` function. An EVM transaction is all-or-nothing: the only two outcomes are "the entire call's state changes commit" or "the entire call's state changes revert".

This means Alice cannot, for instance:

- Verify a proof, then refuse to pay – the verifier acceptance and the `transferFrom` live in the same call, gated by the same `require`.
- Pay 1,200 BUCK but then get away with advancing only part of the tree – `newRoot` is SNARK-attested as the deterministic fold of `all` of `cms[]` onto `oldRoot` at `nextLeafIndex`, and the contract writes it in one store.
- Replay a prior `proofBytes` – the SNARK is bound to `(oldRoot, newRoot, nextLeafIndex, totalFace)` and the `cm[]` / `issuerMode[]` vectors, and the contract's stale-state guards reject any submission whose `oldRoot` no longer matches the live root or whose `nextLeafIndex` no longer matches the live tree size.

- Submit the same opening twice *in one batch* to double-spend at spend time – the nullifier is deterministic in `(rho, idHash)`, so whichever duplicate lands first at spend will consume both. (See the duplicate-commitment note under Step 3.)

6.3 Pillar 3: Public Inputs Are Sealed Into the Proof

The Groth16 verifier consumes the public-signal vector `[issuerMode[0..N), oldRoot, newRoot, nextLeafIndex, totalFace, cm[0..N)]`. The proof was generated against the specific values Alice committed to before submission. If Alice tries to submit the proof with a different `totalFace`, a different `newRoot`, a different `nextLeafIndex`, or a permuted or tampered `cms[]`, the verifier rejects.

The `cm[i]` are public inputs to the SNARK directly: the contract passes the calldata `cms[]` straight into the verifier's public-signal vector, so any mismatch between the commitments the proof sealed and the commitments the transaction carries reverts at verify time.

The refreshed `test/MintVerifier.t.sol` should cover at least these tamper paths, each expected to revert with `Notes: bad mint proof` or `Notes: stale oldRoot` or `Notes: stale leafIndex` or an ABI decode error:

- `totalFace + 1`
- `newRoot` flipped by one bit
- `oldRoot` spoofed to a stale value
- `nextLeafIndex` spoofed to a fictional size
- `cms[0] ^ 1` (commitment tampered in calldata after proving)
- `cms[]` length differs from the dispatched verifier's pinned N
- malformed 256-byte `proofBytes`

Soundness is not a proof of *absence* of attack, but every documented attack surface is closed by a test that triggers a verifier reject.

6.4 Putting the Pillars Together: The Adversarial Walk

Suppose hostile-Alice tries to mint two **contradictory** batches that both claim to be the canonical breakdown of her `T_mint` transfer. She would need:

- Two different `proofBytes` and/or `cms[]`, both passing the on-chain verifier.

That requires either:

- Two valid Groth16 witnesses for the same public inputs (allowed – but they yield the same `cms`, so they are not *contradictory*), or
- Two valid Groth16 witnesses with *different* public inputs. Allowed too, but each one demands its own `transferFrom` for its own `totalFace` – **that** is the duplicate-spend problem ERC-20 solves. Alice cannot transfer the same 1,200 BUCK twice without holding 2,400 BUCK to start with.

So a single 1,200-BUCK escrow yields exactly one accepted batch. The only freedom Alice has is *which* batch she lays down – which denominations and which recipients – and that freedom is hers to keep, by design.

What about hostile-Alice trying to mint *more* commitment value than she escrowed? She would need a witness whose `sum(v[i])` equals the integer `totalFace` *that the EVM sees*, but whose individual `v[i]` values *actually* sum to more. In `F_r` with `r ~ 2^254` and no range check on `v[i]`, the constraint `sum(v[i]) === totalFace` would be satisfiable by witnesses whose integer sum differs from `totalFace` by a multiple of `r`. `mint_batch.circom` closes this with a `Num2Bits(128)` range constraint on each `v[i]` and on `totalFace`, mirrored by `spend.circom`'s check on `face` and the witness `v`. A 128-bit cap is comfortably above any plausible BUCK denomination and well below `r/2`, so an integer overshoot can never wrap.

What about hostile-Alice trying to mint a commitment that is *already* in the tree (to collide on a nullifier she does not control, or to attempt a hash-collision attack)? She cannot collide on a nullifier without knowing somebody else's opening – the nullifier is a Poseidon-PRF output, so forging a collision is as hard as inverting Poseidon. She *can* re-mint her own opening (the contract no longer blocks duplicates), but doing so forces her to pay `totalFace` a second time and buys her nothing: the nullifier is deterministic in the opening, so whichever of her two copies she spends first permanently nullifies both. This is the value-conservation property in action – the chain does not need a `commitmentExists` check because the nullifier mechanism already collapses duplicate openings to one spend, and Alice bears the cost of any duplication.

7 Why Outside Observers Cannot Link

Mallory has full read access to the chain after `T_mint` confirms. What does she see?


```
T_mint:
  from:  Alice (a registered IdentityRegistry account)
  to:    Notes (a system-public contract)
  value: 0 (BUCK is ERC-20, not ETH)
  data:  Notes.mint(proofBytes, oldRoot, newRoot, nextLeafIndex,
                    1200e18, [cm_0..cm_15])

events emitted (in order):
  Buck.Transfer(Alice, Notes, 1200e18)
  Notes.Minted(Alice, 1200e18, startIndex=K, count=16, newRoot)
```

The batch-mint contract emits one `Minted` event per batch rather than per-leaf `Appended` events; the commitments themselves live in calldata for off-chain provers and wallets to read. (Calldata is about an order of magnitude cheaper per byte than LOG data, which is part of why the N-independent cost bound holds.)

What can Mallory infer?

- That Alice (publicly) escrowed 1,200 BUCK. **Yes** – her address is `msg.sender`.
- That sixteen new commitments entered the pool at indices `[K..K+15]`. **Yes** – `Minted` announces `startIndex` and `count`, and `cms[]` is in calldata.
- *Which* commitment is worth *how much*. **No** – each is a fresh `Poseidon5` output hidden by an independent `rho_i`. Without a `rho_i`, distinguishing the 1,000-BUCK hash from the 25-BUCK hash (or from a 0-BUCK dummy) is computationally equivalent to inverting Poseidon on a uniform field element, modelled here as a random oracle.

- /Which entries are live notes and which are dummies.* **No** – the padding leaves are constructed with the same Poseidon-5 shape and a fresh `rho`; nothing about them is chain-distinguishable from a live `cm` with a small `v`.
- *Who* holds each live note. **No** – `idHash_i` is also inside the hash.
- Whether two commitments share a recipient. **No** – two notes addressed to the same `idHash` but with distinct `rho_i` produce uniformly independent `cm_i`, so the chain shows nothing distinguishable from independent draws.

What about *linking back to a specific note artifact* that Bob later possesses? Mallory sees `cm_0..cm_5` published by Alice in `T_mint`. Bob's artifact references some `cm_i` and `T_mint`, but the artifact lives off-chain. Mallory cannot get the artifact unless Bob (or Alice) shows it to her, and the chain alone gives her no way to single out `cm_i` as "the one Bob holds" against the other five.

The remaining linkability surface is the issuer. `msg.sender` is publicly Alice, so every commitment in the batch carries a public **issuer** tag. This is intentional and desired: a recipient must be able to verify who minted his note. The privacy goal is unlinkability of recipients and amounts, not of issuers.

7.1 Spend-Time Unlinkability

The shipped spend circuit (`spend.circom`; see alberta-buck-ethereum.org) lets Bob deposit his note without revealing *which* `cm` in the on-chain set he opened. The spend SNARK proves "I know an opening to *some* commitment in the Merkle tree under the public `noteRoot` that hashes to this nullifier", with `cm` as a private witness rather than a public input. Unlinkability of *spend events* to *mint events* is now a chain-observable property: Mallory sees a fresh nullifier, a public face, and a recipient – nothing that points back to a specific commitment in the tree. The full walkthrough of the redemption flow is in **The Redemption (Spend), Step by Step** below.

8 Atomicity, No Double-Mint, No Replay

Three properties together rule out duplicated or partial mints:

8.1 Atomicity

Already covered above: `Notes.mint` performs proof verification, BUCK pull, and commitment appends in one EVM call. Either all happen or none.

8.2 No Double-Mint From One Transfer

The transfer and the commitments are bound in the same call: there is no "pre-deposit, then mint" or "mint, then deposit later" flow that an attacker could desynchronize. Each `Notes.mint` invocation does its own `transferFrom`; each `transferFrom` debits Alice's BUCK balance once. Two minting transactions require two `transferFrom` debits, which require Alice to actually hold 2 * `totalFace` BUCK.

8.3 No Replay

A literal replay – broadcasting the exact same `T_mint` calldata in a new transaction – fails for three independent reasons:

a. **Stale-state guards.** The contract checks `oldRoot == roots[currentRootIndex]` and `nextLeafIndex == self.nextLeafIndex`. The first mint has already advanced both, so the replay's `oldRoot` and `nextLeafIndex` public inputs no longer match the live chain state. The contract reverts before the SNARK is even called, with `Notes: stale oldRoot` or `Notes: stale leafIndex`. b. **SNARK binding.** Even if a future minter coincidentally re-created the same `oldRoot` and `nextLeafIndex` state (which is prohibited by the monotonic advancement of `nextLeafIndex`), the SNARK was bound to the specific `(cms[], newRoot)` pair of the prior batch. A replay would fold the same `cms[]` a second time and produce a different `newRoot` than the SNARK attested to, breaking the rolling-tree relation. c. **Allowance and balance.** Even setting the above aside, the second mint would re-debit Alice's BUCK and re-consume her allowance to the `Notes` pool – economic-level double-escrow protection on top of the structural guards.

Replay of the same opening across batches is structurally allowed but economically self-punishing: Alice escrows `totalFace` again, the duplicate commitment lands at a fresh leaf index, and the deterministic nullifier `Poseidon3(rho, idHash, 4242)` means the first spend nullifies every copy. Alice pays twice for a spend-once outcome.

8.4 Cross-Chain or Cross-Deployment Replay

The `MintGroth16Verifier` verification key is fixed by the trusted-setup ceremony for that deployment. A proof generated against the dev-key in `scripts/snark/setup.sh` will not verify against a production-key contract on mainnet, and vice versa. Public inputs are also sealed into the proof: a proof valid on Sepolia for `totalFace=1200` cannot be reused on mainnet because the on-chain verifier address (and key) differ.

The only cross-context vulnerability would be a *chain-id collision* on identical verification keys, which the production setup ceremony precludes by construction.

9 Prover Contention and the Rollup Race

Batch minting carries one genuinely operational concern: a mint transaction is a *rollup-style* transaction. Alice's proof is bound to the specific `oldRoot` and `nextLeafIndex` she snapshotted when she began proving; if any other mint lands between her snapshot and her submission, those values go stale and her transaction reverts at one of the stale-state guards.

This is not a *safety* concern – Alice loses no BUCK (the contract reverts before `transferFrom` runs) and produces no bad on-chain state. It is a *liveness and efficiency* concern: Alice wasted the prover time spent constructing a proof that is no longer accepted, and has to re-prove against the fresh state.

9.1 Who Wins When Two Minters Race?

When Alice and Bob both submit `T_mint_A` and `T_mint_B` targeting the same `oldRoot`:

- One of the two (by block ordering) lands first and updates the root.
- The other transaction executes, discovers the stale-state mismatch, and reverts at the `require(oldRoot == roots[currentRootIndex])` line before any BUCK moves.

Transaction fees are paid by the loser for the reverted transaction (gas for the revert path, not for the full mint). The loser then re-proves against the new `oldRoot` and `nextLeafIndex` and resubmits. The new proof folds *their* commitments onto the root that *now* includes the winner's commitments – no commitment has been lost; the loser's tree-position is just different from the one they originally predicted.

9.2 Mitigation Layers

Three mitigation layers, in ascending complexity:

1. **Optimistic retry (wallet-level).** On revert, the wallet re-fetches `(currentRoot, nextLeafIndex)` from the chain, rederives the sibling paths for the new insertion positions, re-runs the prover, and resubmits. For small N this is fast enough to not matter. This is the recommended initial deployment strategy.

2. **Retry budget and back-off (wallet-level).** For large N where prover time is nontrivial (minutes to hours at N=1024), the wallet should expose a "chance of contention" estimate to the user and offer to batch at the smallest pinned N that fits the job, or to enqueue the mint on a private relay channel during low-traffic windows.

3. **Sequencer / proof aggregation (future).** If real-world contention warrants it, an external sequencer can batch multiple users' mints into a single proof per slot, preserving the N-independent on-chain cost while amortising prover time. Sequencer capture is a separate trust axis and is deferred; the shipped protocol is sequencer-free.

9.3 What Does Not Regress

- **Atomicity.** A reverted mint leaves *no* chain-visible state change; it is strictly indistinguishable from no submission.
- **Value conservation.** No reverted transaction debits BUCK, so there is no way for a loser to pay the pool without appending commitments.
- **Unlinkability.** A re-submitted mint on a fresh root still produces the same (anonymity-preserving) calldata and event shape; Mallory cannot tell a retry from a first attempt.
- **Spend-time invariants.** The spend circuit takes the current `noteRoot` (or any recent-window root); a late-arriving mint does not invalidate any previously-proved spend.

The rollup race is the price of replacing ~800K gas/leaf with ~500 gas/leaf, and for practical block-rate contention it is negligible.

10 Receipt Construction From the Transaction ID Alone

A practical question: months later, Bob wants to prove to Carol that his note was legitimately issued. He needs no help from Alice and no live indexer:

1. Bob fetches `T_mint` from any Ethereum archive node.

2. He decodes the call data into `(proofBytes, oldRoot, newRoot, nextLeafIndex, totalFace, cms)` using the standard `Notes.mint` ABI.

3. He notes `msg.sender` and resolves it to Alice's identity via the IdentityRegistry on the same chain (a single `eth_call`).

4. He locates his `cm` in `cms[]` at `leafIndex - nextLeafIndex` and asserts equality.

5. He recomputes `Poseidon5(flavor, v, rho, idHash, predicate) == cm` from the opening.

6. (Optional.) He re-runs `MintGroth16Verifier.verify(proofBytes, [issuerMode[0..N), oldRoot, newRoot, nextLeafIndex, totalFace, cm[0..N)])` against the per-N verifier keyed by `cms.length`, assembling the public-signal vector from the calldata exactly as the contract does, convincing himself the EVM did its job.

The "receipt" Carol receives is therefore `(T_mint, opening, idHash mapping)`. Its validity is independently checkable against the public chain. Nothing here requires ongoing cooperation from Alice – which is exactly the property a receipt should have.

11 The Redemption (Spend), Step by Step

This section walks the *inverse* of the mint. Bob holds the opening for the 1,000-BUCK note `cm_0`; he wants to redeem it for 1,000 BUCK credited to his on-chain account, without revealing *which* commitment in the pool he consumed.

There is **one** spend path, the *Identity-M-bound deposit*: Bob redeems by proving he is the KYC-registered Identity the note names – *not* a particular account – so he may use *any* Fountain account bound to his Identity point M_{rec}, and he reveals no Identity to the chain. Every spend composes the same two halves over a single committed point P:

1. **The note proof** (this section, Steps 1–3): the kept, flavor-agnostic `spend.circom` Groth16 proves the note's commitment `cm` is in the pool tree under a public `noteRoot` and emits a fresh, well-formed `nullifier` – hiding *which* commitment was consumed. `Notes.spendCoupled{A1,A2,B1}` all reuse `spendVerifier.verifySpend` for exactly this.

2. **The identity binding** (Step 4): an EIP-196 Okamoto sigma + a membership proof bound to the *same* P, proving the spender is a registered Identity. The flavor selects which sigma and what P commits – the issuer's M_I (A2), or the spender's own $M_{\text{rec}}/M_{\text{dep}}$ (A1/B1). This is detailed per-flavor in 12; the walk below uses Bob's **A2** note (addressed, private issuer) as the running example.

Three properties of the flow are worth stating up front, because they shape every step:

- **The anonymity set is the full pool, not only same-denomination.** The spend proof takes `cm` as a *private* witness under a public `noteRoot` – the chain-observable anonymity set is every unspent commitment in the tree. The **face value is a public output**, because the BUCK contract must know how much to pay out; amount-level linkability narrows the effective set to commitments whose hidden `v` could equal the revealed `face`. Same-denomination issuance is therefore the strongest-privacy regime, but the underlying cryptographic hiding is over the whole tree, not over a denomination class.

- **The BUCK ERC-20 does not verify the SNARK itself.** The authorization chain is: valid SNARK + accepted root + fresh nullifier (+ for A-spends, valid identity binding) → the relevant `spendCoupled{A1,A2,B1}` entrypoint proceeds → the entrypoint invokes

`buck.transfer(recipient, face)` from the pool (Buck routes it through the Carrying path). The proof is a capability token for entry into the appropriate spend method; the resulting ERC-20 movement is a routine pool-to-recipient call gated by that entry. Since the `Notes` pool is a system-public account in the IdentityRegistry, the pool's outbound transfer does not itself require a Chaum-Pedersen receipt – the recipient's own registration is what makes them acceptable as a transfer destination.

- **Public inputs are bound into the proof, including `chainId`.** `spend.circom` ghost-binds `recipient` and `chainId` (via `x*x` rows) so Groth16's IC[] commitment makes them non-malleable from the mempool. A front-runner who steals Bob's `proofBytes` and resubmits with a different `recipient` or on a different chain id is rejected by the verifier.

11.1 Step 0: Bob Holds a Verified Note

From the mint walk (**Step 5: Recipient Verification**) Bob already has:

```
note_artifact_0 = {
  txHash:    T_mint,
  index:     K + 0,
  cm:        cm_0,
  opening:   ( flavor_A1, 1000e18, rho_0, idHash_bob, predicate_0 ),
  meta:      { issuer: Alice, ... },
}
```

He has independently confirmed that `cm_0` is in `Notes.commitments[]` at index `K` and that `Poseidon5(opening)` reproduces it. The opening has been sitting in his wallet for weeks, months, or years – addressed A-notes are cryptographically safe for indefinite holding (Notes).

11.2 Step 1: Fetch the Commitment Tree and Build a Merkle Path

`Notes.sol` maintains a Tornado-style incremental Poseidon-T3 Merkle tree of `TREE_DEPTH = 20` (max ~1M leaves). The per-leaf insertion math lives inside the mint SNARK; the contract just records the SNARK-attested `newRoot` at each `mint()` call. The current root is exposed as `noteRoot()`. Two things happen:

a. Bob (or, more practically, any light-node helper running off-chain) reconstructs the tree off-chain. The commitments themselves are available from each mint transaction's calldata `cms[]`, keyed by the `Minted(issuer, totalFace, startIndex, count, newRoot)` event that indexes the batch. A replayer walks all `Minted` events in order, takes the `cms[]` slice out of each mint's calldata, and appends them to its local tree using the same `ZERO_VALUE` and Poseidon-T3 wrapper the in-circuit insertion uses. Bob's Merkle path for `cm_0` – a sequence of sibling hashes plus left/right indices up to the root – falls out of that rebuilt tree. The path is purely public information; everyone can reproduce it, and Bob does not have to trust any indexer because calldata is canonical.
b. Bob reads the current `noteRoot()` via `eth_call`. He may instead pin any root in the recent-roots window: `Notes.sol` retains the last `ROOT_HISTORY_SIZE = 30` roots in a ring buffer (`roots[]`, advanced once per inserted leaf), and the on-chain `isAcceptedRoot()` scans that ring backward from `currentRootIndex`. This is the standard zk-rollup hygiene (Zcash, Tornado Cash) for tolerating prover latency: a spend assembled against a slightly-stale root remains acceptable so long as fewer than 30 mints have landed in the meantime.

11.3 Step 2: Compute the Nullifier

The shipped `spend.circom` derives the nullifier as a Poseidon-3 hash of the witness `rho`, the witness `idHash` bound at mint, and a constant tag:

$$nf \;=\; \text{Poseidon}_3\big(\rho_0,\ \text{idHash}_0,\ 4242\big).$$

The tag (4242) is a fixed nonzero field constant that domain-separates nullifier preimages from the Poseidon-5 commitment preimages, so a structural collision between a nullifier and a commitment is impossible. Three properties follow:

- **Deterministic.** The same `(rho, idHash)` always yields the same `nf`. So Bob can only spend `cm_0` once; a second attempt re-derives the same `nf` and is rejected by `nullifiers[nf] = true` on the first spend.
- **Unforgeable without the opening.** An attacker who sees `cm_0` and the tree cannot compute `nf` without `rho_0` and `idHash_0`, both of which live inside the opening (and `rho_0` is not derivable from the recipient's identity alone).
- **The note proof binds the opening and nullifier, not the identity.** The flavor-agnostic `spend.circom` treats `idHash` as a witness scalar – it does not constrain it to a registered Identity. That binding is the *separate* Step-4 deposit gate: the EIP-196 deposit-coupling (A1/A2) or depositor-binding (B1) sigma proves the spender is bound to the named Identity scalar, and the G1-tie membership – bound to the sigma's committed point P – proves that Identity is registered. Splitting the identity binding out of the SNARK (into cheap EIP-196 sigmas, ~36K gas) avoids ~+250K R1CS of native BN254 $\mathbb{G}_1$ arithmetic in non-native circom. And because the deposit is *identity*-keyed (not account-keyed), an addressed note survives the loss of any single account key: re-register a fresh Fountain account on the same M_{rec} and deposit (recovery, not finality).

The nullifier is the chain-observable anchor of the spend. Publishing it marks `cm_0` as consumed from the tree's *semantic* point of view, even though the tree itself remains append-only and nothing is deleted.

11.4 Step 3: Generate the Spend Proof

Bob's wallet runs `scripts/snark/prove_spend.js` (the spend-side analogue of `prove_mint_batch.js`, using the depth-20 PTAU produced by `scripts/snark/setup.sh`). `spend.circom`'s signal layout is exactly:

```
public  noteRoot;         # an accepted root from the on-chain ring buffer
public  nullifier;        # the nf from Step 2
public  face;             # 1000e18 -- the BUCK to release (range-bounded to 128 bits)
public  recipient;        # Bob's on-chain address (the payout destination)
public  chainId;          # replay-domain separation

private flavor, v, rho,
        idHash, predicate;        # the Poseidon-5 opening
private pathElements[depth];      # depth-20 sibling hashes
private pathIndices[depth];       # left/right bit per level
```

The shipped circuit enforces five relation groups:

1. **Range bounds and value match.** Both the public `face` and the witness `v` pass through `Num2Bits(128)`, then `face === v`. This closes the field-overflow gap: `v` cannot wrap around `r` to claim more than its in-range share.
2. **Opening binds.** `cm = Poseidon5(flavor, v, rho, idHash, predicate)`.
3. **Merkle membership.** `MerkleProof(20)` walks (`cm`, `pathElements`, `pathIndices`) up the tree using `Switcher` + `Poseidon(2)` per level and asserts the climb reaches `noteRoot`.
4. **Nullifier derivation.** `nullifier === Poseidon3(rho, idHash, 4242)`.
5. **Public-input ghost binding.** `recipient` and `chainId` each appear in a trivial `x*x` row. This is the minimal anchor needed for circom not to optimise out unreferenced public signals; once anchored, Groth16's IC[] commitment makes them non-malleable from the mempool's perspective.

Critically, `cm` is *not* a public input; it is recomputed inside the proof from the witness opening. The proof convinces the verifier that *some* `cm` in the tree satisfies (1)-(4); it does not reveal *which*.

This `spend.circom` note proof is *flavor-agnostic*: A1, A2, and B1 all use the same circuit and the same on-chain `spendVerifier.verifySpend` check for the cm-membership + nullifier discharge (nullifier tag 4242). Bob's wallet serializes the Groth16 proof into the standard `(uint256[2], uint256[2][2], uint256[2])` ABI tuple (256 bytes). The flavor-specific *identity* material – `eIss` and the deposit-coupling proof `dc` (A1/A2), or `eDepForIss` and the depositor-binding proof (B1), plus the G1-tie membership proof – is assembled separately and submitted alongside the note proof in Step 4.

11.5 Step 4: Submit `Notes.spendCoupledA2` On Chain

Bob's note is A2 (private issuer), so he deposits via `spendCoupledA2`. Beyond the Step 1–3 note proof he supplies `eEnc` – a *fresh re-encryption* of the leaf ciphertext $eIss$ under M_{rec} (same plaintext, new randomness, so the chain never sees the mint-time ciphertext again) – the deposit-coupling sigma `dc` (`alberta_buck.wallet.unilateral_a2.deposit_couple_prove`), the G1-tie membership proof `membershipProof`, and the note-binding proof `noteBindingProof` (`alberta_buck.wallet.note_binding`):

```
notes.spendCoupledA2(proofBytes, noteRoot, nullifier, 1000e18, bob,
                     eEnc, dc, membershipProof, noteBindingProof);
```

The honest EVM walks `Notes.spendCoupledA2` (the function as it ships – a thin wrapper over the shared `_spendCoupled`):

```
require(address(identityRegistry) != address(0), "Notes: identity registry not set");
require(recipient != address(0),  "Notes: zero recipient");
require(face      > 0,            "Notes: zero face");
require(_isAcceptedRoot(root),    "Notes: unknown root");
require(!nullifiers[nullifier],   "Notes: already spent");

// (1) Note proof: cm in the pool tree under ‘root‘, nullifier well-formed.
require(spendVerifier.verifySpend(
    proof, root, nullifier, face, recipient, block.chainid
),                                "Notes: bad spend proof");

// (2) Identity-M binding, half 1 -- the deposit-coupling sigma.  msg.sender is
//     Bob’s deposit account; it proves the account is bound to m_rec AND that
//     eEnc decrypts under m_rec to the point committed (blinded) in dc.P_I.
require(identityRegistry.verifyDepositCoupling(msg.sender, eEnc, dc),
                                  "Notes: bad deposit coupling");
```

```
nullifiers[nullifier] = true;     // mark spent before paying out
noteFaceSum          -= face;

// (3) Identity-M binding, half 2 -- membership of dc.P_I's point, bound to the
//     SAME dc.P_I the coupling just constrained (the verifier derives its
//     public-input limbs from dc.P_I, not from the proof bytes).
_verifyIdentityMembership(membershipProof, dc.P_I.X, dc.P_I.Y);

// (4) Identity-M binding, half 3 -- the note<->eEnc re-encryption tie, binding
//     eEnc to THIS note: the Groth16 proof's public inputs are derived from
//     the caller's nullifier, eEnc, and dc.P_I.
_verifyNoteBinding(noteBindingProof, nullifier, eEnc, dc.P_I.X, dc.P_I.Y);

require(buck.transfer(recipient, face),   // pool isCarrying => Carrying path
        "Notes: transfer failed");
emit SpentCoupledA2(nullifier, face, recipient, dc.P_I.X, dc.P_I.Y);
```

Six invariants are enforced in order:

a. `root` appears in the recent-roots window. A root that has rotated out is stale – Bob must rebuild the proof against a fresher root. b. `nullifier` has never been published before (Theorem 10.5 in the proofs says this single-bit check is sufficient for double-spend prevention). c. The note proof is valid against `SpendGroth16Verifier` via the `SpendVerifierAdapter`, which packs `[root, nullifier, face, recipient, chainid]` into the public-signal vector. By soundness Bob holds a Poseidon-5 opening of a `cm` under `root`, with `nullifier = Poseidon3(rho, idHash, tag)`. d. `verifyDepositCoupling(msg.sender, eEnc, dc)` succeeds: msg.sender's registered account is bound to the identity scalar m_{rec} and `eEnc` decrypts under it to `dc.P_I` – an EIP-196 Okamoto sigma, all Identities hidden (M_{iss} is blinded in $P_I = M_{\text{iss}} + bH$). e. `_verifyIdentityMembership(membershipProof, dc.P_I.X, dc.P_I.Y)` succeeds: the G1-tie Groth16 proves `dc.P_I`'s underlying point is a member of the on-chain `identityRoot`. Because the contract feeds `dc.P_I` (not the proof bytes) as the public input, the membership is bound to *exactly* the point the coupling decrypted `eEnc` to – a colluding pair cannot key the ciphertext to a non-member and still spend. f. `_verifyNoteBinding(noteBindingProof, nullifier, eEnc, dc.P_I)` succeeds: the note-binding Groth16 (`circuits/note_binding.circom`) proves `eEnc` is a re-encryption, under M_{rec}, of the ciphertext committed in *this note's* `idHash` – the same `idHash` inside the nullifier gate (b) just consumed – and that its decryption is the same point `dc.P_I` commits. The adapter (`NoteBindingVerifierAdapter`) derives all 25 public inputs from the caller's `nullifier` / `eEnc` / `dc.P_I`, so the proof bytes carry only the Groth16 triple: an accept is necessarily about *this* spend. Without (f), gates (d)-(e) verify a ciphertext of the depositor's choosing; with it, only the addressed M_{rec} can spend, and an un-nameable A2 note stays un-spendable even under ciphertext substitution (*Proofs*, Theorem 12). Only after (a)-(f) does the contract pay out and emit `SpentCoupledA2(nullifier, face, recipient, P_I.X, P_I.Y)`.

If any of (a)-(f) reverts, the whole transaction reverts: no BUCK moves, the nullifier bit is never set, `noteFaceSum` is unchanged. **Bob gets one attempt per root window**; a failed spend retries with a refreshed proof, but never consumes a nullifier without paying out the BUCK.

An A1 note (public issuer) takes the identical path via `spendCoupledA1` – only `eIss`'s meaning differs (it encrypts M_{rec} under itself, so the membership certifies the *recipient*). A B1 bearer note takes `spendCoupledB1`, where the sigma is `verifyDepositorBinding` and `P` commits the depositor's M_{dep}. All three are detailed in 12.

11.6 Step 5: BUCK ERC-20 Pays Out (the Carrying path)

The call inside `Notes.spendCoupledA2` is a plain `buck.transfer(bob, 1000e18)`. Because the `Notes` pool is bound `isCarrying` in the IdentityRegistry, BUCK routes that transfer through its

internal `_carryingTransfer` path on this single egress. (`spendCoupledA1` and `spendCoupledB1` take the same path.) Nothing in `Buck.sol` is special-cased for the pool: it is just a registered, `isCarrying` account making an outbound transfer of 1,000 BUCK to a registered recipient. The result on-chain:

- `Buck.balanceOf(Notes)` decreases by exactly 1,000 BUCK – nominal balance only.
- `Buck.balanceOf(Bob)` increases by exactly 1,000 BUCK.
- `Buck.Transfer(Notes, Bob, 1000e18)` is emitted (a standard ERC-20 event; the carrying apportionment is internal, with no distinct event).
- `Notes.Spent(nullifier, 1000e18, Bob)` is emitted.
- `Notes.nullifiers[nullifier]` becomes `true`.

What "carrying" means in the ledger. BUCK demurrage is a balance-time integral: each account stores its `buckSeconds` (the crystallised $\int B\,dt$) plus the `timestamp` of its last touch. A transfer *from a non-Carrying account* settles the sender's accrued fee (a one-shot burn) and hands the recipient fresh, zero-age BUCK. A transfer from an `isCarrying` account – the `_carryingTransfer` path – differs in two surgical ways:

1. **No sender-side fee burn.** The pool sheds a proportional slice of its live buck-seconds rather than burning a fee; its per-BUCK age is preserved.
2. **Recipient absorbs the pool's age.** The recipient inherits that slice of buck-seconds (`carried = liveBs * amount / poolRaw`), added to its own:


```
liveBs   = pool.buckSeconds + pool.raw * elapsed;
carried  = liveBs * amount / pool.raw;
pool.buckSeconds       = liveBs - carried;   // pool sheds its share
recipient.buckSeconds += carried;            // recipient inherits the age
```

System-wide BUCK-age is preserved across the call (no fee is burned, no fresh-BUCK dilution at the recipient); the demurrage that the pool would have burned on a standard outgoing transfer instead rides into Bob's account as inherited buck-seconds. Bob will pay it the next time he makes a standard outgoing transfer.

Why this is the right rule for the pool. Per-note age is opaque – the pool holds aggregate BUCK behind opaque commitments and cannot tell Bob's note's age from any other note's. The Carrying path collapses the calculation to the pool's average per-BUCK age at spend time: every spender of the same `face` absorbs the same buck-seconds slice, keeping the spend-time anonymity set intact. The trade-off is that Bob inherits the pool's average age regardless of how long *his specific* note sat in the tree. Anyone whose note rested longer than the pool average comes out ahead; this is the cost of anonymity, and the upside for long-term holders.

The full discussion (one new BUCK method, one new event, no per-account mode flag) is in **Demurrage-Carrying Transfers** below.

11.7 What the Observer Sees at Spend (Mallory Returns)

After `T_spend` confirms, Mallory can read:

```
T_spend:
  from:  Bob (a registered IdentityRegistry account)
  to:    Notes (system-public)
  data:  Notes.spendCoupledA2(proofBytes, noteRoot, nullifier, 1000e18, Bob,
                              eIss, dc, membershipProof)

events:
  Buck.Transfer(Notes, Bob, 1000e18)
  Notes.SpentCoupledA2(nullifier, 1000e18, Bob, P_I.x, P_I.y)
```

What she infers:

- **That a note was spent.** Yes – the nullifier is fresh and the pool balance dropped by 1,000 BUCK.
- **That the spend was for 1,000 BUCK to Bob.** Yes – `face` and `recipient` are public inputs (they must be, to tell the ERC-20 whom to pay).
- *Which commitment in the tree was consumed?* **No** – `cm` is a private witness to the SNARK. Mallory's anonymity set is every unspent commitment of flavor `A` in the pool whose hidden `v` could equal 1,000 BUCK. At deployment scale this is potentially the entire pool minus prior nullified entries; in a same-denomination regime (many 1,000-BUCK notes in circulation) it is exactly the 1,000-BUCK subset.
- *Which mint did the spent note come from?* **No** – there is no cross-reference to `T_mint` in `T_spend`. The only link back to any specific mint would be the nullifier, but nullifiers are Poseidon-PRF outputs over secret material; inverting one to recover `(flavor, M_rec, rho)` is exactly the pre-image resistance we assume.
- *Who held the note before Bob?* **No**. If Alice gave the note to Carol who handed it to Dave who sold it to Bob, none of those transfers touched the chain. The chain sees only "at mint Alice deposited, later Bob withdrew" – the intermediate holders are invisible.

The residual linkability surfaces are (i) `face` (the revealed amount) and (ii) `recipient` (the on-chain payout address). Both are unavoidable at the SNARK/ERC-20 boundary – the contract must know where and how much to pay. The community- standardized mitigation is fixed-denomination notes (so revealing `face` does not subdivide the anonymity set) and batched, relay-mediated payouts (so revealing `recipient` need not reveal the *true* beneficiary). Both are orthogonal to the core spend circuit and can be layered on top.

11.8 Why Alice Cannot Steal Bob's Note (Even Between Rooms)

Can *any* actor other than Bob – Alice-the-issuer, a prior holder, or a random Mallory – redeem `cm_0`? Three protections apply, the third being the Identity-M deposit binding (the deposit-coupling sigma + the bound membership):

1. **Opening secrecy.** `spend.circom` requires the full Poseidon-5 opening `(flavor, v, rho, idHash, predicate)` as a private witness. Without `rho_0` and `idHash_0` the circuit is unsatisfiable, so no one without the opening can produce a valid nullifier or proof. For Alice's

secrets, this is the load- bearing protection on bearer-flavor (B1) notes: she could in principle race the bearer to deposit, which is the civil-trust semantics of bearer cash and is discussed at length in alberta-buck-notes.org §Long-Term Holding Safety.

2. **Recipient-binding at the payout site.** `recipient` is a public SNARK input ghost-bound by `recipient * recipient`, so Groth16's IC[] commitment makes it non-malleable. A front-runner who intercepts Bob's `proofBytes` and resubmits with a different `recipient` is rejected by the verifier. `chainId` is bound identically, blocking cross-chain replay.

3. **Identity binding via the deposit-coupling sigma + bound membership.** For an A2 deposit Bob provides `eIss` and the coupling proof `dc`; `spendCoupledA2` calls `verifyDepositCoupling(msg.se` `eIss, dc)`, which reads Bob's registered `(pk_dep, E_dep)` from storage and confirms a single `sk_dep` and identity scalar m_{rec} bind the account (`C_d === m_rec * G + sk_dep * R_d`) *and* decrypt `eIss` under m_{rec} to the committed point `dc.P_I`. The G1-tie membership then proves `dc.P_I`'s underlying point is in the on-chain `identityRoot`, bound to the *same* `dc.P_I` the coupling fixed. Alice-the-issuer cannot redeem to herself even knowing `idHash_0` and `rho_0`: she is not bound to m_{rec}, so the coupling fails. The sigma costs ~36K gas (EIP-196 ecMul/ecAdd + keccak Fiat-Shamir); the membership Groth16 verify is ~350K, constant.

The strict identity-binding guarantee is now a chain-observable property: the `verifyDepositCoupling` + membership booleans are the gate. The note proof enforces the cm-membership + nullifier discharge; the coupling + membership enforce that the depositor is the registered Identity the note named – all A1/A2 identity, security, and privacy properties preserved (and, for A2, the issuer is provably nameable – the collusion gap closed).

11.9 B1 Bearer Notes

A B1 note (bearer, public issuer) is redeemed the same way, with the depositor binding playing the role the coupling does for A2: `spendCoupledB1` calls `verifyDepositorBinding` (the depositor re-encrypts its own M_{dep} under the public issuer's key, coupled to its payout account) plus a membership proof of M_{dep}. Knowledge of the Poseidon-5 opening is still the *bearer* authorisation – if two parties hold copies of the same B1 opening (a paper note photographed and forwarded), whichever submits first wins, the intended civil-trust semantics of bearer cash – but the depositor is additionally proven to be a registered Identity the issuer can name (see 12).

11.10 Recapping Property C (Only Valid Notes Redeem)

The original Property C question – *subsequently only those N valid notes can be deposited* – becomes concrete at this layer. Claim: no actor can redeem a commitment `cm` unless

a. `cm` was appended to `commitments[]` by a prior `Notes.mint` transaction whose Groth16 proof verified (i.e., `cm` is part of an honest mint batch), *and* b. The actor holds the opening of `cm`, *and* c. For A-spends, the actor is the depositor keyed by `M_rec` in that opening.

Proof sketch. The spend circuit's relation 2 (Merkle membership) rejects any `cm` not in `noteRoot`. The mint circuit's relations 1-2 (opening binding and sum conservation) are what places a `cm` in `noteRoot` in the first place, and by the mint-side soundness story in **Why Alice Cannot Cheat** every such `cm` corresponds to a well-formed opening whose `v` is included in the mint's `totalFace` escrow. The spend circuit's relation 5 (`face === v`) then ensures the BUCK released equals exactly the share of the escrow that `cm` was allocated. Summing over every accepted spend of a given mint batch, the cumulative BUCK released equals at most the `totalFace` escrowed – property C follows from mint-side value conservation plus spend-side nullifier uniqueness.

Field wrap-around is foreclosed by range bounds: `mint_batch.circom` range-bounds each `v[i]` to 128 bits and `spend.circom` range-bounds both `face` and the witness `v` identically, so an integer overshoot of `totalFace` by a multiple of `r` is structurally impossible at either end. Property C rests on those range bounds plus the nullifier-uniqueness check and Merkle membership.

12 The Identity-M Spend Path (A1, A2, B1)

Step 4 above showed the A2 deposit in the redemption walk; this section gives the per-flavour detail for all three (the notes-identity-axis unification). The spend is *one gadget*, parameterised by flavour: an EIP-196 Okamoto sigma producing a single committed point P, plus a Poseidon-Merkle membership proof *bound to that same* P, directly enforcing "the counterparty Identity is a registered KYC Identity" at spend time – without the circuit overhead of non-native $\mathbb{G}_1$ arithmetic. Three entrypoints instantiate it:

- **A2 (addressed, private issuer)** – `Notes.spendCoupledA2`. The recipient deposits via `verifyDepositCoupling`, proving its account is bound to the M_{rec} that decrypts the note ciphertext $eIss$ to the issuer Identity committed in P_I; the membership certifies that issuer is registered – closing the A2 recipient-key collusion gap.
- **A1 (addressed, public issuer)** – `Notes.spendCoupledA1`. Identical on-chain path; the note encrypts M_{rec} under itself, so P_I commits the *recipient* and the membership certifies *them*. The issuer is public, named off chain at mint (the batch Schnorr).
- **B1 (bearer, public issuer)** – `Notes.spendCoupledB1`. The dual: the depositor re-encrypts its own M_{dep} under the public issuer's key (`verifyDepositorBinding`), coupled to its payout account and to a committed P_{dep}; the membership certifies M_{dep}, and the issuer decrypts the event to name the depositor.

All three gate on the *membership* of the committed point in the registry-Identity accumulator $\mathsf{rt}_{\mathsf{id}}$, whose root is the single on-chain `IdentityRegistry.identityRoot`, updated incrementally by each `register()` / `bindContract()`. In every case the membership is *bound* to the sigma's own P – the verifier derives its public inputs from P, not from the prover's bytes, so the two halves cannot be decoupled.

The *addressed* flavours (A1, A2) add a fourth gate: the note $\leftrightarrow$ *eEnc tie* (`circuits/note_binding.circom` for the A2 payload layout, `circuits/note_binding_a1.circom` for A1, behind one `INoteBindingVerifier`). The flavour-agnostic note proof exposes no $idHash$, so without the tie nothing binds the deposit-coupling ciphertext to the *specific note being spent* – a holder of a stolen opening could supply a self-addressed $eEnc$, and an A2 coalition could substitute a nameable one. The tie proves, in one Groth16 proof, that the spend's $eEnc$ is keyed to the identity material committed in the `idHash` behind *this spend's nullifier* (A2: $eEnc$ re-encrypts the committed $eIss$; A1: $eEnc$ and the note's own $eNote$ share one M_{rec}, the spend's public `face` pinning the $eNote$ plaintext), and that the decrypted point is the very P_I the sigma committed (*Proofs*, Theorem 12).

12.1 The Shared Primitives

Before the flavour-specific walkthroughs, three primitives are common to all three:

- **The identity accumulator.** Identity points $M = mG$ for every registrant are folded into a Poseidon Merkle tree whose root $\mathsf{rt}_{\mathsf{id}}$ lives on chain as `IdentityRegistry.identityRoot`, updated incrementally by each `register()` / `bindContract()` (the Tornado-style `filledSubtrees` pattern, depth 10). A newly issued M is usable only after its insertion – the *commit-before-use* discipline from the identity-axis unification (now "Mutual Decryptability" in notes.org and this "Identity-M" section). The one canonical tree is `alberta_buck.registry.tree.IdentityMerkleTree` (the wallet and the sim share it; `alberta_buck.wallet.unilateral_a2.IdentityTree` is a thin point-centric facade over it), matched byte-for-byte by the contract.

- **The membership-plus-G1-tie SNARK (shipped and bound).** A single Groth16 circuit, `circuits/identity_membership_g1tie.circom` (verified by `IdentityMembershipG1TieVerifier.sol`, ~350K gas, constant), proves both halves at once: $\ell(M) = \text{Poseidon}(M.x, M.y) \in \mathsf{rt_{id}}$ AND the G1 tie $P_I = M + bH$ (one BN254 $\mathbb{G}_1$ point addition via circom-lib's `EllipticCurveAddOptimised`). Its nine public inputs are $[\mathsf{rt_{id}},\ PI_x[0..3],\ PI_y[0..3]]$ – the committed point P_I as 64-bit $\mathbb{F}_q$ limbs. **The binding:** the on-chain verifier derives those limbs from the deposit-coupling sigma's *own* P_I, not from prover-supplied bytes, so a Groth16 accept proves membership of *exactly* the point the sigma decrypted $eIss$ to. A colluding pair cannot answer the coupling with one P_I and the membership with another (`IdentityMembershipBinding.t.sol`, 5 tests).
- **The aggregator.** Multiple registries and feature authorities feed sub-roots into a `CentralMerkleService` (depth 10, up to 2^10 sub-trees), whose single aggregated root is the on-chain `identityRoot`. Prototyped in `alberta_buck.registry.merkle_service` (49 tests).

12.2 A2 Identity-Targeted Spend (The Unilateral Receipt Flow)

Bob's note is flavour A2 (addressed, private issuer). The note opening carries $eIss = (R_e, C_e)$ – the issuer's Identity M_I encrypted under Bob's Identity point M_{rec}, plus the anti-framing binding (`verifyIssuerReenc`) that ties $eIss$ to the issuer's own registered Identity. Bob holds the one secret m_{rec} that can decrypt it.

==== Step A0: Bob Holds the A2 Note ====

The artifact Bob received off chain:

```
note = (
  flavor = A2,  v = 1000e18,  rho,  idHash,  predicate,  # the commitment opening
  eIss  = (R_e, C_e),                                    # M_I under M_rec
  binding,                                               # verifyIssuerReenc
  mint_tx_ref,                                           # where cm was folded
)
```


==== Step A1: Bob Chooses a Deposit Account ====

Bob selects *any* registered Fountain account bound to his identity scalar m_{rec}. His wallet reads the account's registered $(pk_{\text{dep}}, E_{\text{dep}} = (R_d, C_d))$ from the IdentityRegistry. Because all accounts share the same m_{rec}, he is free to use a fresh account every time – no re-registration, no key pinning.

==== Step A2: Generate the Deposit-Coupling Proof ====

Bob's wallet builds the *deposit-coupling proof* (`deposit_couple_prove` in `alberta_buck.wallet.unilateral_a`
The prover proves knowledge of $(m_{\text{rec}}, sk_{\text{dep}}, b)$ such that:

(E4)	$pk_{\text{dep}} = sk_{\text{dep}}\, G$	– the real account key;
(E2)	$C_d = m_{\text{rec}}\, G + sk_{\text{dep}}\, R_d$	– the account is bound to $M_{\text{rec}} = m_{\text{rec}} G$;
(E3)	$C_e - P_I = m_{\text{rec}}\, R_e - b\, H$	– $eIss$ decrypts under m_{rec} to $P_I - bH$.

This is a 3-witness Okamoto sigma over EIP-196, producing 8 struct members: (e, s_m, s_s, s_b, A2, A3, A4, P_I). Zero identities are revealed; the issuer Identity M_I is perfectly hidden in $P_I = M_I + bH$.

==== Step A3: Generate the Identity Membership Proof ====

The wallet generates the G1-tie Groth16 proof (`circuits/identity_membership_g1tie.circom`). Its *public* inputs are the on-chain `identityRoot` and the committed point $P_I = M_I + bH$ – the

same P_I the deposit-coupling sigma carries – as 64-bit $\mathbb{F}_q$ limbs. Its *private* witnesses are the issuer Identity M_I, the blind $T = bH$, and the Merkle path. The circuit proves, in one proof,

$$P_I = M_I + T \quad \wedge \quad \ell(M_I) = \text{Poseidon}(M_I.x, M_I.y) \in \mathsf{rt_{id}},$$

i.e. *the point the sigma decrypted to is a genuine, registered KYC Identity.* M_I never appears – only the blinded P_I is public, so the issuer stays hidden from Mallory while being provably a member.

The wallet does not put P_I into the proof *bytes* it submits; only the Groth16 triple (a, b, c) travels. The contract supplies P_I to the verifier from the deposit-coupling sigma (Step A2), which is what *binds* the two halves – see Step A5.

==== Step A4: Submit the Deposit ====

Bob submits the identity-M-bound deposit entrypoint, `Notes.spendCoupledA2`:

```
notes.spendCoupledA2(
    proof,                          // Groth16: note opening + noteRoot membership
    noteRoot,
    nullifier,                      // Poseidon(rho, idHash, A-tag)
    face,                           // 1000e18
    recipient,                      // Bob's payout address
    eIss,                           // (R_e, C_e): the leaf ciphertext
    dc,                             // DepositCouplingProof (e,s_m,s_s,s_b,A2,A3,A4,P_I)
    membershipProof,                // G1-tie Groth16 (a,b,c) bytes; P_I supplied by the contract
);
```

`eIss`, `dc`, and `membershipProof` are the identity-M additions. Crucially, `membershipProof` carries *only* the Groth16 triple; the committed point P_I it is checked against is read by the contract from `dc.P_I` – the wallet cannot decouple them.

==== Step A5: On-Chain Verification (the binding) ====

`spendCoupledA2` gates on five conditions, reading the committed point `dc.P_I` *once* and feeding it to *both* identity halves:

1. Standard note spend: `noteRoot` is accepted, `nullifier` is fresh, the Poseidon-5 opening Groth16 proof verifies (`spendVerifier.verifySpend`).

2. Deposit coupling: `verifyDepositCoupling(msg.sender, eIss, dc)` accepts – the EIP-196 sigma confirms the depositing account is bound to M_{rec} and $eIss$ decrypts under it to `dc.P_I`. (7 Solidity + 11 Python tests.)

3. Bound identity membership: `verifyMembership(membershipProof, identityRoot, dc.P_I.X, dc.P_I.Y)` accepts – the G1-tie Groth16 proves `dc.P_I`'s underlying M_I is a registered KYC Identity. The verifier derives the circuit's P_I limbs from `dc.P_I`, *not* from the proof bytes, so this is provably the *same* point gate 2 constrained.

4. BUCK transfer: pool pays `face` to `recipient` via the Carrying path.

5. Nullifier recorded: `nullifiers[nullifier] = true`; the contract emits `SpentCoupledA2(nullifier, face, recipient, P_I.X, P_I.Y)`.

If any check fails, the whole transaction reverts. No BUCK moves, no nullifier is consumed. (End-to-end: `NotesCoupledA2.t.sol`, 8 tests.)

Why this closes the collusion gap. Suppose a colluding issuer keys $eIss$ to a throwaway point so the recipient "cannot" name the issuer. Gate 2 still passes (the depositor can prove $eIss$ decrypts to *some* point P_I under their m_{rec}). But that point's underlying M_I is then *not* a

registered Identity, so *no* G1-tie proof exists for it, and gate 3 fails: **the deposit reverts**. There is no spendable-but-unnameable note. An honest note, by contrast, decrypts to the issuer's registered M_I, a member, and spends cleanly.

==== Step A6: The Unilateral Receipt ====

After the spend confirms, Bob's wallet produces a *unilateral receipt* naming both parties – from m_{rec} *alone*:

$$\begin{aligned} M_I &= C_e - m_{\text{rec}} R_e && \text{(the issuer Identity);} \\ M_{\text{rec}} &= m_{\text{rec}} G && \text{(the recipient's own).} \end{aligned}$$

A *verifiable decryption* proof (`verifiable_decrypt_prove`, a Chaum-Pedersen DLEQ) makes the receipt third-party-checkable without revealing m_{rec}. The receipt tuple:

$$\mathcal{R} = \big(M_I,\ M_{\text{rec}},\ v,\ eIss,\ \text{vd},\ \text{binding},\ \text{issuer addr},\ \text{chainid}\big).$$

A court or auditor checks four links (`verify_receipt`):

#	Check	Establishes
1	`verifyIssuerReenc` vs. issuer's on-chain record	$eIss$ re-encrypts the issuer's *registered* M_I (no framing
2	`verifiable_decrypt_verify(eIss, M_rec, M_I)`	$eIss$ decrypts under M_{rec} to M_I
3	$\ell(M_I) \in \mathsf{rt_{id}}$	the issuer Identity is registered (the *coupling*)
4	$\ell(M_{\text{rec}}) \in \mathsf{rt_{id}}$	the recipient Identity is registered

Full specification and tests: *alberta-buck-notes-unilateral*.

12.3 A2 in Code: The Full Cycle, End to End

The cryptography above is small enough to run. This section is a complete, *executing* transcript against the shipped wallet (`alberta_buck.wallet.unilateral_a2` over BN254 $\mathbb{G}_1$) – mirroring `test_unilateral_a2.py`. The blocks below are org-babel source blocks sharing one Python session: re-exporting this document re-runs them top to bottom, and the printed `RESULTS` are regenerated. The session is seeded, so a re-run reproduces this transcript byte for byte – and fails loudly if the wallet API drifts.

The cast: two Identities, one of them with two accounts. An *Identity* is a KYC-bound scalar m and its point $M = mG$. The Identity Fountain lets one m back unlimited unlinkable *accounts*, each an ElGamal encryption $E_{\text{addr}} = (R, C) = (rG,\ M + r\,pk)$ of the *same* M under that account's own key.

```
import random
from alberta_buck.wallet.bn254 import G1, ORDER, mul, add, neg, eq, rand_scalar
from alberta_buck.wallet.elgamal import elgamal_encrypt, elgamal_decrypt
from alberta_buck.wallet.unilateral_a2 import (
    IdentityTree, mint_unilateral_a2,
    deposit_couple_prove, deposit_couple_verify,
    make_receipt, verify_receipt,
)

_seed = random.Random(0xA1BE27AB0CC)     # ONE deterministic transcript
rng   = lambda: _seed.getrandbits(256)
```

```
CHAINID = 1
def account(m):                    # one Fountain account bound to identity m
    sk = rand_scalar(rng); pk = mul(G1, sk)
    M  = mul(G1, m)                # the shared Identity point
    E  = elgamal_encrypt(M, pk, rand_scalar(rng))    # registered credential
    return dict(m=m, M=M, sk=sk, pk=pk, E=E)

def pt(P):                         # compact point display
    return f"({int(P[0]) % 10**8:>8d}..., {int(P[1]) % 10**8:>8d}...)"

m_iss, m_rec = rand_scalar(rng), rand_scalar(rng)
issuer = account(m_iss)                      # the payer (private issuer)
rec0   = account(m_rec)                      # recipient account #0
rec1   = account(m_rec)                      # recipient account #1 (SAME identity)
M_rec  = mul(G1, m_rec)                      # what the issuer learns out of band

tree = IdentityTree()                        # the registry-Identity accumulator
for a in (issuer, rec0):                     # one leaf per *identity*
    tree.insert(a["M"])

print(f"issuer Identity    M_I   = {pt(issuer['M'])}")
print(f"recipient Identity M_rec = {pt(M_rec)}  (accounts rec0, rec1)")
print(f"identity tree root        = {hex(tree.root())[:20]}...")
```

```
issuer Identity    M_I   = (34794633..., 55370756...)
recipient Identity M_rec = (78777287...,  1501867...)  (accounts rec0, rec1)
identity tree root        = 0x1b94c62584dc6075ad...
```

Mint: encrypt the issuer's own Identity under the recipient's point. The issuer addresses the note to *the Identity* M_{rec}, not an account. It encrypts its *own* registered M_I under M_{rec} used as a public key; the anti-framing binding (`verifyIssuerReenc`) proves $eIss$ re-encrypts the issuer's OWN registered M_I – it cannot name a victim.


```
note = mint_unilateral_a2(issuer["sk"], issuer["E"], M_rec,
                          v=1000, rho=rand_scalar(rng),
                          issuer=0xA11CE, chainid=CHAINID, rng=rng)
assert eq(note.M_I, issuer["M"])
print(f"minted A2 note: v = {note.opening.v}")
print(f"  cm     = {hex(note.cm)[:20]}...       (the only on-chain artifact)")
print(f"  idHash = {hex(note.idHash)[:20]}...   (Poseidon8(eNote, eIss))")
print(f"  eIss   = ElGamal(M_I under M_rec)  R = {pt(note.eIss.R)}")
```

```
minted A2 note: v = 1000
  cm     = 0x119e5f967cfb3426ea...       (the only on-chain artifact)
  idHash = 0x5c689616da10e765ed...   (Poseidon8(eNote, eIss))
  eIss   = ElGamal(M_I under M_rec)  R = (24688725..., 93913945...)
```

Delivery: the recipient names the issuer with one secret. The decryption secret is m_{rec} – shared by *every* account the recipient owns – because $M_{\text{rec}} = m_{\text{rec}}G$:

$$C_e - m_{\text{rec}}R_e = (M_I + r'M_{\text{rec}}) - m_{\text{rec}}(r'G) = M_I.$$

```
M_I = elgamal_decrypt(note.eIss, m_rec)
assert eq(M_I, issuer["M"])
print("recipient decrypts eIss with m_rec -> the issuer's registered Identity:")
print(f"  C_e - m_rec*R_e = {pt(M_I)} == M_I")
```

```
recipient decrypts eIss with m_rec -> the issuer's registered Identity:
  C_e - m_rec*R_e = (34794633..., 55370756...) == M_I
```

Deposit from ANY account bound to m_{rec}. The issuer never knew which account would cash the note. Both `rec0` and `rec1` can, because both share m_{rec}. The blind b is drawn once here because the note-binding SNARK below must commit the *same* $P_I = M_I + bH$ the sigma publishes.

```
b = rand_scalar(rng)              # the P_I blind -- shared with the SNARK below
for name, acct in (("rec0", rec0), ("rec1", rec1)):
    dc = deposit_couple_prove(m_rec, acct["sk"], acct["E"], note.eIss,
                              account=0xB0B, chainid=CHAINID, b=b, rng=rng)
    ok = deposit_couple_verify(acct["pk"], acct["E"], note.eIss, dc,
                               account=0xB0B, chainid=CHAINID)
    print(f"deposit-coupling sigma from {name}: verifies = {ok}  "
          f"P_I = {pt(dc.P_I)}")
```

```
deposit-coupling sigma from rec0: verifies = True  P_I = (74064912..., 52659236...)
deposit-coupling sigma from rec1: verifies = True  P_I = (74064912..., 52659236...)
```

`dc` is the EIP-196 Okamoto sigma (E4/E2/E3 of Step A2). Its only Identity-derived public value is $P_I = M_I + bH$, *perfectly hidden* by the fresh blind b. On chain, `Notes.spendCoupledA2` hands this same `dc.P_I` to the membership verifier *and* to the note-binding verifier – the binding:

```
// inside Notes._spendCoupled (abridged; spendCoupledA2 passes a1Layout=false,
// spendCoupledA1 true -- selecting which binding circuit answers):
require(reg.verifyDepositCoupling(msg.sender, eEnc, dc), "bad deposit coupling");
//                                                  dc.P_I --- the same point ---v
_verifyIdentityMembership(membershipProof, dc.P_I.X, dc.P_I.Y);
_verifyNoteBinding(noteBindingProof, nullifier, a1Layout, face,
                   eEnc, dc.P_I.X, dc.P_I.Y);
```


The note-binding witness: tie *eEnc* to THIS note. The spend supplies *eEnc*, a *fresh re-encryption* of the leaf's *eIss* (same plaintext, new randomness s) – the chain never sees the mint-time ciphertext again. The `note_binding` SNARK proves *eEnc* re-encrypts the ciphertext committed in the *idHash* behind this spend's nullifier, and that its decryption is the P_I point the sigma committed. The witness builder runs the whole relation in Python (every constraint is asserted before anything is proved):

```
from alberta_buck.wallet.note_binding import make_note_binding_witness

s = rand_scalar(rng)                      # fresh re-encryption randomness
witness = make_note_binding_witness(
    rho=note.opening.rho, eNote=note.eNote, eIssCommitted=note.eIss,
    s=s, m_rec=m_rec, b=b, M_I=note.M_I, r_iss=note.r_prime,
)
print(f"binding witness built: {len(witness)} signals, all relations asserted")
print(f"  public nullifier = {hex(int(witness['nullifier']))[:20]}...")
print(f"  eEnc.R limb0     = {witness['eEncRx'][0]}  (fresh -- not the leaf's eIss.R)")
R0x_leaf = witness["R0_limb"][0][0]
print(f"  eIss.R limb0     = {R0x_leaf}")
assert witness["eEncRx"][0] != R0x_leaf   # re-encryption, not reuse
```

```
binding witness built: 20 signals, all relations asserted
  public nullifier = 0x315cd0f43dcbfc5235...
  eEnc.R limb0     = 7194077872496329669  (fresh -- not the leaf's eIss.R)
  eIss.R limb0     = 13585057060835952469
```

Proving and verifying this witness is the heavyweight step – a 2.4M-constraint Groth16 circuit – so it runs in the build pipeline rather than in this document's session. The real transcript (Apple Silicon, `make nix-snark-note-binding`):

```
$ make nix-snark-note-binding
--- Compiling circuit (circom --c --no_asm --O2) ---
  R1CS: 471M   (2.4M constraints, 0 linear after --O2, 25 public inputs)
--- Building C++ witness generator ---     (~9 min, gcc -O3 -fno-strict-aliasing)
--- Generating witness ---                 (~9 s, 73 MB witness.wtns)
--- Checking witness (snarkjs wtns check) ---
[INFO]  snarkJS: WITNESS IS CORRECT
--- Groth16 setup (ptau: pot22_final.ptau) ---   (1.3 GB zkey)
--- Proving (rapidsnark) ---               (seconds, vs ~33 s for snarkjs)
[INFO]  snarkJS: OK!
Result: Valid proof                        (rapidsnark verifier agrees)
  -> src/NoteBindingGroth16Verifier.sol (EIP-197 proof-B swap applied)
  -> test/vectors/note_binding/proof.json
$ forge test --match-contract NoteBindingVerifier
Suite result: ok. 9 passed; 0 failed       (~501K gas raw, ~508K via adapter)
```

The unilateral receipt: the recipient names BOTH parties, alone. From m_{rec} *alone* – no issuer cooperation – the recipient recovers both Identities and produces a third-party-checkable receipt:

```
receipt = make_receipt(m_rec, note, issuer=0xA11CE, chainid=CHAINID,
                       tree=tree, rng=rng)
res = verify_receipt(receipt, issuer["pk"], issuer["E"], tree.root(), tree)
assert res.valid
assert eq(res.issuer_M, issuer["M"])     # the payer, named
assert eq(res.recipient_M, M_rec)        # the payee, named
assert res.value == 1000
print(f"unilateral receipt: {res.reason}")
print(f"  names issuer    {pt(res.issuer_M)}")
print(f"  names recipient {pt(res.recipient_M)}")
print(f"  value           {res.value}")
```



```
unilateral receipt: VALID
  names issuer    (34794633..., 55370756...)
  names recipient (78777287...,  1501867...)
  value           1000
```

Collusion fails twice: un-nameable is un-spendable, and substitution cannot bind. First the classic collusion: an issuer keys $eIss$ to a *throwaway* point. The mint anti-framing binding still accepts (it only forces $eIss$ over the issuer's own M_I), but the recovered point is not a tree member – the receipt is INVALID and the membership proof cannot exist. Second, the substitution escape the note-binding closes: the colluders *can* build a binding witness over their bogus ciphertext (they know its randomness), but its $idHash$ is not the spent note's – the SNARK's nullifier disagrees with the one the spend consumed, and the on-chain public-input derivation rejects the proof.

```
pk_bogus = mul(G1, rand_scalar(rng))                # a key the recipient lacks
r_bogus  = rand_scalar(rng)                         # colluders know their own r
eIss_bogus = elgamal_encrypt(issuer["M"], pk_bogus, r_bogus)
M_I_bogus = elgamal_decrypt(eIss_bogus, m_rec)      # garbage, not issuer["M"]
print(f"colluding mint: eIss keyed to a throwaway -- decrypts to {pt(M_I_bogus)}")
print(f"  registered identity?  {tree.contains(M_I_bogus)}  -> membership unprovable")

b_bogus = rand_scalar(rng)
witness_sub = make_note_binding_witness(
    rho=note.opening.rho, eNote=note.eNote, eIssCommitted=eIss_bogus,
    s=rand_scalar(rng), m_rec=m_rec, b=b_bogus, M_I=M_I_bogus, r_iss=r_bogus,
```

```
)
print(f"substitute-ciphertext nullifier = {hex(int(witness_sub['nullifier']))[:20]}...")
print(f"the spent note's nullifier      = {hex(int(witness['nullifier']))[:20]}...")
assert witness_sub["nullifier"] != witness["nullifier"]
print("  -> public-input mismatch: the substitute proof cannot bind THIS spend")
```

```
colluding mint: eIss keyed to a throwaway -- decrypts to (27018772..., 43291458...)
  registered identity?  False  -> membership unprovable
substitute-ciphertext nullifier = 0x271cc47267f3952333...
the spent note's nullifier      = 0x315cd0f43dcbfc5235...
  -> public-input mismatch: the substitute proof cannot bind THIS spend
```

That is the whole guarantee in one screen: *the recipient holds the single secret that names both parties, and the issuer cannot make a spendable note that the recipient is unable to name – nor can any holder bind a different ciphertext to it.*

12.4 A1 in Code: The Public-Issuer Addressed Cycle

A1 is A2 with the issuer in the clear: $idHash$ commits the issuer's public Identity and batch Schnorr instead of an encrypted $eIss$, and the leaf ciphertext `eRec` encrypts M_{rec} *under itself* – so the deposit coupling commits the *recipient's* Identity and the membership certifies *them*. (The batch Schnorr is checked at the mint transaction level – `verifyIssuerSchnorr`; here it is a synthetic signature, since the session has no chain.)


```
from alberta_buck.wallet.unilateral_a1 import (
    mint_unilateral_a1, make_receipt_a1, verify_receipt_a1,
)

k = rand_scalar(rng)
sigma_R, sigma_s = mul(G1, k), (k + rand_scalar(rng) * rand_scalar(rng)) % ORDER

note_a1 = mint_unilateral_a1(M_rec, v=100, rho=rand_scalar(rng),
                             m_issuer=m_iss, sigma_R=sigma_R, sigma_s=sigma_s,
                             rng=rng)
print(f"minted A1 note: v = {note_a1.opening.v}")
print(f"  idHash = Poseidon8(eNote, m_issuer, sigma)  = {hex(note_a1.idHash)[:20]}...")
print(f"  eRec   = ElGamal(M_rec under M_rec)         R = {pt(note_a1.eRec.R)}")

rcpt_a1 = make_receipt_a1(m_rec, note_a1, issuer["M"], issuer=0xA11CE,
                          chainid=CHAINID, tree=tree, rng=rng)
res_a1 = verify_receipt_a1(rcpt_a1, tree.root(), tree)
assert res_a1.valid and res_a1.value == 100
print(f"A1 receipt: {res_a1.reason} -- issuer public, recipient proves the note")
print(f"  addressed THEIR identity: eRec decrypts under m_rec to M_rec")
```

```
minted A1 note: v = 100
  idHash = Poseidon8(eNote, m_issuer, sigma)  = 0x2ff6525153c4aa14df...
  eRec   = ElGamal(M_rec under M_rec)         R = (98448865..., 79624823...)
A1 receipt: VALID -- issuer public, recipient proves the note
  addressed THEIR identity: eRec decrypts under m_rec to M_rec
```

The A1 note-binding witness: tie $eEnc$ to THIS note, face public. An A1 $idHash$ commits $(eNote, m_{\text{issuer}}, \sigma)$ – no second ciphertext – so the A2 binding relation is not constructible for it; A1 spends carry the sibling circuit `note_binding_a1.circom` instead. The tie runs through the note's *own* value ciphertext $eNote = (r_nG,\ vG + r_nM_{\text{rec}})$: the SNARK proves $eNote$ was addressed to the *same* M_{rec} that keys the spend's $eEnc$ and opens the sigma's P_I. The note face v is a *public* input – `Notes.spendCoupledA1` passes the spend's `face`, itself bound to the note's

committed value by the spend SNARK over the same nullifier. That public pin is load-bearing: ElGamal is not key-committing, and the opening (including r_n) travels to the spender, so with a free plaintext a thief could re-key *eNote* to *any* identity by absorbing the difference; fixing the plaintext at vG makes $m_{\text{rec}} = (\log_G C_n - v)\, r_n^{-1}$ unique (*Proofs*, Theorem 12, A1 statement). The witness builder asserts the whole relation (a fresh local generator keeps the shared transcript untouched):

```
from alberta_buck.wallet.note_binding import make_note_binding_a1_witness

_seed_a1 = random.Random(0xA1B1D)         # local; does not perturb the session rng
rng_a1   = lambda: _seed_a1.getrandbits(256)

t  = rand_scalar(rng_a1)                  # eEnc total randomness (r' + s)
b1 = rand_scalar(rng_a1)                  # the P_I blind
witness_a1 = make_note_binding_a1_witness(
    rho=note_a1.opening.rho, eNote=note_a1.eNote, v=note_a1.opening.v,
    m_issuer=m_iss, sigma_R=sigma_R, sigma_s=sigma_s,
    r_note=note_a1.r_note, m_rec=m_rec, t=t, b=b1,
)
print(f"A1 binding witness built: {len(witness_a1)} signals, all relations asserted")
print(f"  public face v    = {witness_a1['v']}  (pins the addressed identity)")
print(f"  public nullifier = {hex(int(witness_a1['nullifier']))[:20]}...")

# The re-keying escape the public face closes: a thief holding the full
# opening (rho, eNote, r_note, sigma...) but a different identity m' has
# no satisfying witness -- eNote.C != v*G + r_note*(m'*G) for m' != m_rec.
m_thief = rand_scalar(rng_a1)
try:
    make_note_binding_a1_witness(
        rho=note_a1.opening.rho, eNote=note_a1.eNote, v=note_a1.opening.v,
        m_issuer=m_iss, sigma_R=sigma_R, sigma_s=sigma_s,
        r_note=note_a1.r_note, m_rec=m_thief, t=t, b=b1)
    print("thief witness with m' != m_rec: BUILT (must not happen!)")
except AssertionError as exc:
    print(f"thief witness with m' != m_rec: REFUSED ({exc})")
```



```
A1 binding witness built: 20 signals, all relations asserted
  public face v    = 100  (pins the addressed identity)
  public nullifier = 0x15cdafc732324264d6...
thief witness with m' != m_rec: REFUSED (eNote.C != v*G + rn*M_rec)
```

Proving runs in the build pipeline (`make nix-snark-note-binding-a1`; ~2.86M non-linear constraints, every EC operation fixed-base because each point in the relation is a known multiple of G); the exported `NoteBindingA1Groth16Verifier` answers `INoteBindingVerifier.verifyNoteBindingA1` behind the same adapter as the A2 tie, and `test/NotesE2E.t.sol` verifies a real A1 binding proof on chain inside the full lifecycle (~377K gas).

12.5 B1 Bearer Spend with Depositor Naming (The A2 Dual)

Carol holds a B1 note from a public issuer. She has never been identified to the issuer; at spend she reveals her Identity to the issuer *alone*, through an encryption only the issuer can decrypt.

==== Step B0: Carol Holds a B1 Note ====

Carol's artifact carries the issuer's public Identity M_{issuer} in the clear with a Schnorr signature, but *no* per-recipient ciphertext (bearer semantics).

==== Step B1: Carol Encrypts Her Identity for the Issuer ====

At deposit Carol's wallet encrypts her Identity under the public issuer's key:

$$E_{\text{dep}\to\text{iss}} = (R_f, C_f) = (rG,\ M_{\text{Carol}} + r\, pk_{\text{iss}}).$$

Only sk_{iss} recovers M_{Carol}; to Mallory the ciphertext is IND-CPA noise.

==== Step B2: Generate the Depositor-Binding Proof (A2 dual) ====

Carol's wallet proves (`b1_bind_prove`, `verifyDepositorBinding`) knowledge of $(m_{\text{Carol}}, sk_{\text{dep}}, r)$ such that the ciphertext encrypts the *same* Identity bound to Carol's payout account:

$$\begin{aligned}
&\textbf{(E4)} && pk_{\text{dep}} = sk_{\text{dep}}\, G;\\
&\textbf{(E2)} && C_d = m_{\text{Carol}}\, G + sk_{\text{dep}}\, R_d;\\
&\textbf{(F1)} && R_f = r\, G;\\
&\textbf{(F2)} && C_f = m_{\text{Carol}}\, G + r\, pk_{\text{iss}};\\
&\textbf{(P)} && P_{\text{dep}} = m_{\text{Carol}}\, G + b\, H.
\end{aligned}$$

The shared witness m_{Carol} couples (E2), (F2), and (P): the Identity bound to Carol's payout account is exactly the one encrypted for the issuer *and* the one committed (blinded) in P_{dep} – so a colluding depositor cannot substitute a different Identity, and the membership below is bound to the right point. (11 Python + 6 Solidity tests.)

==== Step B3: Submit `Notes.spendCoupledB1` ====

Carol calls the membership-bound B1 deposit:

```
notes.spendCoupledB1(
    proof, noteRoot, nullifier, face, recipient,
    issuer,                         // public issuer address
    eDepForIss,                     // (R_f, C_f) encrypting M_Carol
    b1Proof,                        // verifyDepositorBinding (incl. P_dep)
    membershipProof,                // membership of M_Carol, bound to P_dep
);
```


The contract verifies the binding, then the membership of `b1Proof.P_dep` (the *same* point the binding committed), and emits `SpentCoupledB1(nullifier, face, recipient, issuer, eDepForIss)` with the ciphertext only the issuer can decrypt. Carol is now provably a registered Identity the issuer can name – the membership-bound dual of the A2 recipient receipt (`test/NotesCoupledB1.t.sol`, 6 tests).

==== Step B4: Issuer Decrypts and Produces a Receipt ====

The issuer scans `SpentCoupledB1` events, decrypts M_{Carol} with sk_{iss}, and *alone* produces a plaintext receipt naming both parties:

$$\mathcal{R}_{\text{iss}} = \big(M_{\text{issuer}},\ M_{\text{Carol}},\ v,\ E_{\text{dep}\to\text{iss}},\ \text{vd},\ \text{chainid}\big).$$

(`make_issuer_receipt` in `alberta_buck.wallet.b1_binding`.)

12.6 B1 in Code: The Issuer Names the Depositor

The same session, continued. Carol is a registered Identity the (public) issuer has never met; she holds a bearer note's opening. At spend she builds `eDepForIss` – her own M_{dep} encrypted under the issuer's key – and the five-relation depositor-binding sigma couples it to her payout account and to the blinded membership commitment P_{dep}:

```
from alberta_buck.wallet.b1_binding import (
    b1_bind_prove, b1_bind_verify, make_issuer_receipt, verify_issuer_receipt,
)
```

```
m_carol = rand_scalar(rng)
carol   = account(m_carol)
tree.insert(carol["M"])

pk_iss = issuer["pk"]
binding, eDepForIss = b1_bind_prove(m_carol, carol["sk"], carol["E"], pk_iss,
                                    account=0xCA801, chainid=CHAINID, rng=rng)
ok = b1_bind_verify(carol["pk"], carol["E"], pk_iss, eDepForIss, binding,
                    account=0xCA801, chainid=CHAINID)
print(f"B1 depositor binding from Carol: verifies = {ok}")
print(f"  eDepForIss = ElGamal(M_carol under pk_iss)  R = {pt(eDepForIss.R)}")
print(f"  P_dep      = {pt(binding.P_dep)}  (blinded; membership binds it)")
```

```
B1 depositor binding from Carol: verifies = True
  eDepForIss = ElGamal(M_carol under pk_iss)  R = (71317108..., 92478725...)
  P_dep      = (15608053..., 69485306...)  (blinded; membership binds it)
```

The issuer alone – reading `eDepForIss` from the `SpentCoupledB1` event – decrypts and *proves* the depositor's Identity, the dual of the A2 unilateral receipt. And the collusion-substitution is rejected: a ciphertext under any other key breaks the coupled sigma, so the depositor cannot misname itself.

```
rcpt_iss = make_issuer_receipt(issuer["sk"], issuer["M"], eDepForIss,
                               value=25, issuer=0xA11CE, chainid=CHAINID,
                               tree=tree, rng=rng)
res_b1 = verify_issuer_receipt(rcpt_iss, tree.root(), tree)
assert res_b1.valid
assert eq(res_b1.recipient_M, carol["M"])
print(f"issuer-unilateral receipt: {res_b1.reason}")
print(f"  names depositor {pt(res_b1.recipient_M)} == M_carol")

eDep_sub = elgamal_encrypt(carol["M"], mul(G1, rand_scalar(rng)), rand_scalar(rng))
ok_sub = b1_bind_verify(carol["pk"], carol["E"], pk_iss, eDep_sub, binding,
                        account=0xCA801, chainid=CHAINID)
print(f"substituted eDepForIss verifies = {ok_sub}  (collusion-substitution rejected)")
```

```
issuer-unilateral receipt: VALID
  names depositor (55296567..., 79403012...) == M_carol
substituted eDepForIss verifies = False  (collusion-substitution rejected)
```


12.7 Summary: What the Observer Sees (Mallory)

In all three flows the chain reveals:

- A fresh nullifier and a spent face amount.
- The payout address.
- For A1/A2: the coupled ciphertext $eIss$ and the point commitment P_I (both semantically secure; for A1 P_I commits the recipient, for A2 the private issuer).
- For B1: the issuer address and $E_{\text{dep}\to\text{iss}}$ (IND-CPA).
- The on-chain `identityRoot` (a single uint256, updated on registration, revealing nothing about which identities are active).
- *No* plaintext Identities, *no* link between the spend and any specific commitment in the tree, and *no* link between mint and spend events.

13 The Prover Bill of Materials: Bandwidth, Storage, Time

Everything above *names* the proofs; this section prices them. The short answer to "is wallet-side proving expensive?" is **no – proving is seconds; distribution is the cost**. A Groth16 wallet never performs the trusted setup (the multi-GB Powers-of-Tau and the hours of zkey ceremony are a one-time, project-side event); it downloads two finished artifacts per circuit – the **proving key** (`.zkey`) and the **witness generator** (a wasm module, or a faster platform C++ binary plus its `.dat` constants) – and should pin their hashes against the published release. Verification costs a wallet *nothing* to distribute: the verifiers are contracts already on chain. And note the asymmetry that matters for UX: **receipts need no SNARK at all** – both parties' AB-RCPT/1 slips (Receipt) are sigma proofs and decryptions, milliseconds of wallet math – the SNARKs are paid only at mint and spend, once per note.

Who proves what:

- **Issuer, at mint**: one batch-mint proof – `mint_batch` (B1/A1) or `mint_batch_a2` (A2) at the chosen batch pin $N \in \{1, 2, 4, 8, 16, 32\}$ – plus artifact-free wallet sigmas (the batch Schnorr for B1/A1; the per-leaf `issuer_reenc` binding for A2; sub-millisecond each).
- **Depositor, at spend**: the flavor-agnostic `spend` proof, the bound `identity_membership_g1tie` proof, and – for the *addressed* flavors only – the note↔=eEnc= tie (`note_binding` for A2, `note_binding_a1` for A1); plus the artifact-free deposit sigma (deposit-coupling / depositor-binding) and the `eEnc` re-encryption.
- **B1 is the light flavor twice over**: no tie circuit at spend, and no addressed ciphertexts at mint.

The blocks below are **live**: they read the artifact sizes from the local SNARK build (`build/snark/`, produced by `make snark` and friends) and then *actually run* every wallet prover once over the committed e2e fixture worlds, timing each. (The printed timing lines vary with hardware – Apple-silicon laptop here; every other line is deterministic.)

13.1 What the wallet downloads and stores

```
from pathlib import Path
SNARK = Path("build/snark")

def mb(*paths):
    return sum(Path(p).stat().st_size for p in paths) / 1e6

ARTIFACTS = [
    # circuit                          consumer    zkey                                                witness generator (+ .dat)
    ("mint_batch N=1   (B1/A1 mint)",  "issuer",
     "mint_batch_n1/mint_batch_n1_final.zkey",
     ["mint_batch_n1/mint_batch_n1_js/mint_batch_n1.wasm"]),
    ("mint_batch N=32  (B1/A1 mint)",  "issuer",
     "mint_batch_n32/mint_batch_n32_final.zkey",
     ["mint_batch_n32/mint_batch_n32_js/mint_batch_n32.wasm"]),
    ("mint_batch_a2 N=1  (A2 mint)",   "issuer",
     "mint_batch_a2_n1/mint_batch_a2_n1_final.zkey",
     ["mint_batch_a2_n1/mint_batch_a2_n1_js/mint_batch_a2_n1.wasm"]),
    ("mint_batch_a2 N=32 (A2 mint)",   "issuer",
     "mint_batch_a2_n32/mint_batch_a2_n32_final.zkey",
     ["mint_batch_a2_n32/mint_batch_a2_n32_js/mint_batch_a2_n32.wasm"]),
    ("spend            (all spends)",  "depositor",
     "spend/spend_final.zkey",
     ["spend/spend_js/spend.wasm"]),
    ("g1tie membership (all spends)",  "depositor",
```

```
     "g1tie/g1tie_0001.zkey",
     ["g1tie/identity_membership_g1tie_js/identity_membership_g1tie.wasm"]),
    ("note_binding     (A2 spend)",    "depositor",
     "note_binding/note_binding_0001.zkey",
     ["note_binding/note_binding_cpp/note_binding",
      "note_binding/note_binding_cpp/note_binding.dat"]),
    ("note_binding_a1  (A1 spend)",    "depositor",
     "note_binding_a1/note_binding_a1_0001.zkey",
     ["note_binding_a1/note_binding_a1_cpp/note_binding_a1",
      "note_binding_a1/note_binding_a1_cpp/note_binding_a1.dat"]),
]
print(f"{'circuit':34s} {'consumer':10s} {'zkey':>10s} {'witness gen':>12s}")
print("-" * 70)
sizes = {}
for name, who, zkey, gens in ARTIFACTS:
    zs = mb(SNARK / zkey)
    gs = mb(*(SNARK / g for g in gens))
    sizes[name] = (zs, gs)
    print(f"{name:34s} {who:10s} {zs:>8.1f}MB {gs:>10.1f}MB")
```

```
circuit                            consumer         zkey  witness gen
----------------------------------------------------------------------
mint_batch N=1   (B1/A1 mint)      issuer         10.0MB        2.6MB
mint_batch N=32  (B1/A1 mint)      issuer        319.6MB        5.3MB
mint_batch_a2 N=1  (A2 mint)       issuer         10.5MB        3.8MB
mint_batch_a2 N=32 (A2 mint)       issuer        332.8MB        6.7MB
spend            (all spends)      depositor       5.5MB        2.9MB
g1tie membership (all spends)      depositor      18.8MB        2.3MB
note_binding     (A2 spend)        depositor    1432.7MB       31.6MB
note_binding_a1  (A1 spend)        depositor    1645.9MB       35.1MB
```


The two tie zkeys dominate – they are the proving keys for the ~2.4M (A2) / ~2.86M (A1) constraint circuits, and Groth16 proving keys scale with constraint count. Everything else a wallet ever fetches is 2-19 MB; the small circuits (`mint_batch` N=1 at ~22K constraints, `spend` at ~12K, `g1tie` at ~39K) have correspondingly small keys.

13.2 Running every wallet prover, timed

One real proof per circuit, over the same fixture worlds the live-EVM transcript spends (Receipt, *The Whole Flow, Executed*): snarkjs (wasm witness + prove) for the small circuits, the circom C++ witness generator

- rapidsnark for the big ties – exactly the pipeline

`scripts/snark/gen_e2e_fixtures.sh` runs.

```
import json, shlex, subprocess, tempfile, time

E2E    = SNARK / "e2e"
FIX    = Path("alberta_buck/test/vectors/e2e")
RAPID  = next(Path("lib").glob("rapidsnark-*/bin/prover"))
tmp    = Path(tempfile.mkdtemp(prefix="docflow-prove-"))

def timed(cmd):
    t0 = time.monotonic()
    subprocess.run(cmd, check=True, capture_output=True)
    return time.monotonic() - t0

times = {}

# Issuer: the batch-mint proofs (witness + Groth16 via snarkjs).
for flavor, script in (("b1", "prove_mint_batch.js"),
```

```
                      ("a2", "prove_mint_batch_a2.js")):
    args = json.loads((FIX / f"{flavor}.json").read_text())["mint_args"]
    args = [a.replace(f"e2e_{flavor}", f"docflow_{flavor}") for a in args]
    times[f"mint {flavor} (issuer)"] = timed(
        ["node", f"scripts/snark/{script}", *args])

# Depositor: the flavor-agnostic spend proof (against the docflow mint).
payout = json.loads((FIX / "a2.json").read_text())["payout"]
times["spend (depositor, all)"] = timed(
    ["node", "scripts/snark/prove_spend.js",
     "build/snark/mint_batch_a2_n1/fixtures/docflow_a2.json",
     "0", payout, "1", "docflow_a2"])

# Depositor: the bound G1-tie membership proof.
g1 = SNARK / "g1tie"
times["membership (depositor, all)"] = (
    timed(["npx", "snarkjs", "wtns", "calculate",
           str(g1 / "identity_membership_g1tie_js/identity_membership_g1tie.wasm"),
           str(E2E / "a2/g1tie_input.json"), str(tmp / "g1tie.wtns")])
  + timed(["npx", "snarkjs", "groth16", "prove", str(g1 / "g1tie_0001.zkey"),
           str(tmp / "g1tie.wtns"),
           str(tmp / "g1tie_proof.json"), str(tmp / "g1tie_public.json")]))

# Depositor, addressed flavors: the note<->eEnc tie (C++ witness gen +
# rapidsnark -- the wasm path works too, several times slower).
for flavor, nb in (("a2", "note_binding"), ("a1", "note_binding_a1")):
    gen  = SNARK / nb / f"{nb}_cpp" / nb
    zkey = next((SNARK / nb).glob("*_0001.zkey"))
    wit  = timed(["bash", "-c",
                  f"ulimit -s 65520 && {shlex.quote(str(gen))} "
                  f"{shlex.quote(str(E2E / flavor / 'note_binding_input.json'))} "
                  f"{tmp}/{nb}.wtns"])
    prv  = timed([str(RAPID), str(zkey), f"{tmp}/{nb}.wtns",
                  f"{tmp}/{nb}_proof.json", f"{tmp}/{nb}_public.json"])
    times[f"note tie {flavor} (depositor)"] = wit + prv

for k, v in times.items():
    print(f"{k:30s} {v:6.2f}s")
```



```
mint b1 (issuer)                1.08s
mint a2 (issuer)                0.95s
spend (depositor, all)          0.75s
membership (depositor, all)     1.72s
note tie a2 (depositor)        11.64s
note tie a1 (depositor)        13.70s
```

13.3 The bill, per flavor and role

Download = first-run fetch (zkey + witness generator); prove = wall time for every SNARK that party runs for one note, from the measurements above. The artifact-free sigma legs (Schnorr / issuer_reenc / deposit coupling / depositor binding / eEnc re-encryption / receipt vd proofs) add milliseconds and nothing to download.

```
BILL = [
    ("issuer    b1/a1 (N=1)", ["mint_batch N=1   (B1/A1 mint)"],
     ["mint b1 (issuer)"]),
    ("issuer    a2    (N=1)", ["mint_batch_a2 N=1  (A2 mint)"],
     ["mint a2 (issuer)"]),
    ("depositor b1",          ["spend            (all spends)",
                               "g1tie membership (all spends)"],
     ["spend (depositor, all)", "membership (depositor, all)"]),
    ("depositor a1",          ["spend            (all spends)",
                               "g1tie membership (all spends)",
                               "note_binding_a1  (A1 spend)"],
```

```
     ["spend (depositor, all)", "membership (depositor, all)",
      "note tie a1 (depositor)"]),
    ("depositor a2",          ["spend            (all spends)",
                               "g1tie membership (all spends)",
                               "note_binding     (A2 spend)"],
     ["spend (depositor, all)", "membership (depositor, all)",
      "note tie a2 (depositor)"]),
]
print(f"{'party / flavor':24s} {'download+store':>15s} {'prove (per note)':>17s}")
print("-" * 58)
for who, arts, ts in BILL:
    dl = sum(sum(sizes[a]) for a in arts)
    pt = sum(times[t] for t in ts)
    print(f"{who:24s} {dl:>13.1f}MB {pt:>16.2f}s")
```

```
party / flavor            download+store  prove (per note)
----------------------------------------------------------
issuer    b1/a1 (N=1)             12.7MB             1.08s
issuer    a2    (N=1)             14.3MB             0.95s
depositor b1                      29.5MB             2.47s
depositor a1                    1710.5MB            16.16s
depositor a2                    1493.7MB            14.10s
```

13.4 What this means for the wallet UX

- **Lazy, tiered artifact fetch.** A receive-and-hold wallet needs *zero* SNARK artifacts. A B1-spending wallet needs ~25 MB (`spend` + `g1tie`). The 1.4-1.7 GB tie keys are needed only to *spend an addressed note*, so fetch them in the background on first A1/A2 receipt (the user will plausibly spend later), with hash-pinned, resumable downloads. An issuer wallet sizes its mint key to its batch habit: 10 MB at N=1, ~330 MB at N=32.
- **One progress bar that matters.** Every proof except the tie is under ~2 s – below UX-noticeable for a payment flow. The tie's ~10-15 s is the one moment to design for: prove in the background between "note received and verified" and "deposit", or show explicit progress at spend. Proving is local and private – nothing leaves the device – and never blocks the counterparty (the issuer's proof is done before delivery; the depositor's at their leisure).
- **Per-payment chain bandwidth is negligible** next to the artifacts: the depositor replays the commitment tree from `Minted` calldata (32 B per leaf in the pool plus log overhead) and reads one identity-tree path; both are KB-scale at today's pool sizes and cacheable incrementally.
- **Verification is free to distribute**: the Groth16 verifiers are on chain, and the bilateral receipts re-verify with no SNARK artifacts at all – a third party checks a slip with the pure-Python wallet alone.

14 Demurrage-Carrying Transfers

The Note-spend payout in Step 5 went out as a plain `buck.transfer` that BUCK routed through its internal carrying path because the `Notes` pool is bound `isCarrying`. This section unpacks why a single per-account flag – not a separate method, not a marker interface – is sufficient, and what its ledger effect looks like.

14.1 Two Ways to Handle Demurrage at Transfer Time

BUCK demurrage is a balance-time integral: each account stores its `buckSeconds` (the crystallised $\int B\,dt$) plus the `timestamp` of its last touch, and its accrued fee is `rate \cdot (buckSeconds + balance \cdot elapsed)`. Two ledger responses to that accrued fee are distinguishable at transfer time, selected by the *sender's* `isCarrying` flag:

- *Deducting transfer (sender is non-Carrying, the default).* On a 1,000-BUCK outgoing, Alice's accrued fee for the pre-transfer balance is settled at Alice – one-shot burned against her balance, and her `buckSeconds` is re-crystallised from the new balance. The 1,000 BUCK then lands at Bob *fresh*: his inflow begins accruing age from now, diluting his effective age. From a system perspective, accrued fees are burned at each transfer and the recipient enters fresh.
- *Carrying transfer (sender is `isCarrying`).* On the same 1,000-BUCK outgoing, nothing is burned at the sender: the pool sheds a proportional slice of its live buck-seconds (`carried = liveBs \cdot 1000 / poolRaw`) and keeps its per-BUCK age. Bob inherits that slice (`buckSeconds +` carried=), so he absorbs the age of the BUCKs he just received. Nominal balances move by exactly 1,000 on each side; no fee is burned; total system BUCK-age is preserved.

The invariant the carrying variant keeps is: **nominal BUCK out = nominal BUCK in** (no silent burn at the transfer site), with the recipient's forward index shift representing the demurrage they will pay on their next standard outgoing transfer.

14.2 Why the Note Pool Needs It

Three structural features of the pool make the deducting default wrong:

1. **Per-note age opacity.** When Alice mints six notes, the pool records only six opaque Poseidon commitments. The pool cannot deduct "the demurrage *Alice's specific note* owes" at spend time, because the chain does not distinguish one commitment's age from another. Any per-note fee would have to be computed off the pool aggregate, which would force Bob to pay fees that the pool's other depositors owed – and, worse, differentiate one commitment's payout from another, destroying the anonymity set.
2. **Face-to-payout identity.** `face` is the public SNARK output and the value the mint circuit committed to. A deducting pool transfer would deliver `face - fee` to Bob while the proof claimed `face`. This would break the mint-to-spend value-conservation invariant: the SNARK's promise would not match the ERC-20 ledger movement, and the spend circuit would have to become aware of demurrage accounting to compensate.
3. **Pool as bookkeeping buffer.** The pool is not a long-term BUCK holder; every BUCK that enters is earmarked for one future redeemer. Deducting fees at the pool's egress would siphon value into a burn that the pool has no way to replenish, producing a cumulative insolvency as outstanding notes circulate.

The carrying path fixes all three: no per-note age, `face` to payout, no pool insolvency.

14.3 Shipped Surface in `Buck.sol`

There is no separate `transferCarrying` method and no `TransferCarrying` event. The carrying behaviour is selected by a per-account flag, `isCarrying`, set when the account is bound:

```
// IdentityRegistry: governance sets isCarrying at bind time.
function bindContract(address t, ..., bool isPublicIdentity_, bool isCarrying_) external;
mapping(address => bool) public isCarrying;
```

The standard `transfer` path then dispatches on the *sender's* flag:

```
if (identity.isCarrying(from)) _carryingTransfer(from, to, amount);
else                           _nonCarryingTransfer(from, to, amount);
emit Transfer(from, to, amount);     // one standard event either way
```

The preconditions are exactly those of any transfer (sender and recipient both `isVerified`, the bilateral receipt-fragment / `_identityHash` identity check), because it *is* an ordinary transfer. `_carryingTransfer` sheds a proportional slice of the sender's live buck-seconds onto the recipient rather than burning a fee at the sender:

```
liveBs   = from.buckSeconds + from.raw * elapsed;
carried  = liveBs * amount / from.raw;
from.buckSeconds       = liveBs - carried;   // sender keeps its per-BUCK age
recipient.buckSeconds += carried;            // recipient inherits the age
```

Carrying accounts hold no NFT-backed credit and cannot go negative (the path asserts `amount` < raw=). What makes an account pool-friendly is the *flag*, set once at bind time; there is no per-call method to invoke and no balance precondition beyond that non-negative assertion.

14.4 Who Else Is Carrying

The Note pool is the canonical `isCarrying` account, but the flag is not pool-private: governance binds AMM pools, the Jubilee, and the Notes pool `isCarrying`, and could bind any transitory-holder contract the same way – escrow accounts whose beneficiaries are the true owners, payroll contracts pulling from an employer, cross-chain bridge egresses. The semantic invariant is "the account is a transitory holder of BUCK earmarked for its recipients," and in that case carrying the age forward matches the economic reality.

For ordinary peer-to-peer accounts the deducting default remains correct: the sender is paying from their own pocket and is the natural place to burn their own accrued fee, so EOAs are bound non-Carrying.

14.5 Open Question: Identity Receipts on Carry

Because a carrying transfer *is* an ordinary `transfer`, it uses the same identity predicate – both parties `isVerified`, with the receipt-fragment fallback to `_identityHash` on the public side when a party has `isPublicIdentity` = true=. The Notes pool side relies on its bound Public Identity to satisfy the bilateral check. Whether a carrying transfer between two Encrypted-Identity counterparties should require its own per-call Chaum-Pedersen receipt is an open question; for the Notes pool it is moot (the pool is bound under a Public Identity), but if transitory-holder carry is ever opened to private-to-private pairs an approve-style receipt may be wanted for auditability. See the **Open Work Items** in alberta-buck-ethereum.org.

15 Putting the Guarantees Side-by-Side

The mint walkthrough established five guarantees; the spend walkthrough adds four more that become concrete only once the redemption path exists.

Stage	Guarantee	Mechanism
Mint	Atomic conservation	Single EVM call: proof verify, `transferFrom`, append-cms all-or-nothing
Mint	Issuer-bound, recipient-blinded	`msg.sender` is public; `cm_i` is Poseidon-hidden over `(flavor, v, rho, idHash, pred)`
Mint	Per-note disclosure (recipient only)	Off-chain opening contains exactly one note's witness fields
Mint	Offline receipt reconstruction	`T_mint` calldata + opening + on-chain Identity lookup is sufficient and self-checkable
Mint	Adversarial-prover soundness	Groth16 knowledge soundness on fixed on-chain verifier; ERC-20 balance/allowance integri
Spend	Redemption unlinkable to specific `cm`	`cm` is a *private* witness under public `noteRoot`; only nullifier + face + recipient revealed
Spend	Double-spend prevented	Deterministic nullifier + one-bit `nullifiers` mapping; rejected on reuse
Spend	Recipient-bound payout	`recipient` is a public SNARK input; proof rejects if re-targeted in-flight
Spend	Value conservation across mint<->spend	Range-bounded mint `v[i]` + spend `face === v`; cumulative payouts $\leq$ `totalFace` per batc

All nine rows are enforced by code in the tree. On the mint side the Merkle insertion lives *inside* the SNARK (`mint_batch.circom`, per-N Groth16 verifiers, per-N adapter dispatch); the chain writes the SNARK-attested `newRoot` into the 30-slot root-history ring `Notes.sol` maintains for the spend side. On the spend side: the flavor-agnostic `spend.circom` note proof, `SpendGroth16Verifier`, `SpendVerifierAdapter`, the on-chain Merkle accumulator, and BUCK's `isCarrying` carrying-transfer path. The IdentityRegistry provides the issuer-attestation and recipient-verification legs (extended with the Identity-M deposit gates – `verifyDepositCoupling` for A1/A2, `verifyDepositorBinding` for B1, plus the G1-tie membership verifier), and the formal proofs (Theorems 7-11) provide the cryptographic backstop. Each spend's identity binding – an EIP-196 Okamoto sigma proving the spender is the named registered Identity, bound to a Poseidon-Merkle membership of the *same* committed point – costs ~36K gas for the sigma plus a constant ~350K-gas membership verify, much cheaper than proving native BN254 $\mathbb{G}_1$ equality inside the spend SNARK (~+250K R1CS).

Why the spend carries this apparatus at all – anonymity is the cost. A direct `transfer` reaches the same mutual-decryptability guarantee with no SNARK, because the bilateral `approve` names both parties at once. The Notes detour pays the mint SNARK, the recipient-key blinding, and the deposit-coupling purely to buy mint <-> spend unlinkability: the issuer's registered anchor lives at mint, the recipient's at spend, and they are never co-present, so the apparatus is what bridges them. Public-issuer notes (A1/B1) keep the cheap, fully-sound path; A2 is the premium that hides the issuer too. See *alberta-buck-notes* ("Why A2 Costs More: Anonymity Is the Cost") and the privacy audit below ("The EOA Transfer Mirror").

16 Deployment Status and Open Questions

The redemption walk above *is* the identity-M spend – the only spend path: one gadget (`verifyDepositCoupling` / `verifyDepositorBinding` + the G1-tie membership `identity_membership_g1tie.circom`, wired through `spendCoupled{A1,A2,B1}`). All of its load-bearing pieces are *shipped*:

1. **On-chain identity accumulator.** `IdentityRegistry.identityRoot` is an incremental Poseidon accumulator (depth 10, Tornado-style `filledSubtrees`) updated on every `register()` / `bindContract()`, with the *commit-before-use* discipline so a newly issued M is usable only once the root reflects it. The one canonical off-chain tree (`alberta_buck.registry.tree.IdentityMerkleTre` shared by the wallet and the sim) matches it byte-for-byte; the `CentralMerkleService` aggregator (49 tests) feeds sub-roots in.

2. **Non-native $\mathbb{G}_1$ tie in the membership circuit.** `identity_membership_g1tie.circom` proves the native Poseidon-Merkle membership *and* the G1 tie $P_I = M + bH$ in one Groth16

proof (one BN254 $\mathbb{G}_1$ addition via circom-lib), exported to `IdentityMembershipG1TieVerifier.sol` (on-chain verified, ~350K gas).

3. **The spend routed through the Identity-M gates, with the binding.** `Notes.spendCoupledA2` calls `verifyDepositCoupling` and the membership verifier, **passing `dc.P_I` to both** so the two halves are bound to the same committed point (the verifier derives its public-input limbs from `dc.P_I`, not from the proof bytes – `IdentityMembershipBinding.t.sol`, `NotesCoupledA2.t.sol`). This is the gate that closes the A2 recipient-key *collusion* gap: a bogus $eIss$ makes P_I's point a non-member, so no membership proof exists and the deposit reverts. A1 and B1 take the identical bound-membership path: `spendCoupledA1`, and `spendCoupledB1`, which binds `verifyDepositorBinding`'s committed point `P_dep` to the membership exactly as A2 binds `dc.P_I` (`NotesCoupledB1.t.sol`).

4. **The note $\leftrightarrow$ *eEnc* tie, both layouts.** The flavour-agnostic note proof exposes no $idHash$, so the gates above alone would verify a ciphertext *of the depositor's choosing* – leaving addressed-binding ("only M_{rec} can spend") and the A2 substitution escape ("attach a different, nameable *eEnc*") unenforced. `circuits/note_binding.circom` (~2.4M non-linear constraints; the ElGamal-structure optimisation keeps every EC operation fixed-base) proves the spend's *eEnc* re-encrypts the ciphertext committed in the $idHash$ behind *this spend's nullifier*, and that its decryption is the sigma's own committed P_I point. `circuits/note_binding_a1.circom` (~2.86M constraints) is the A1-layout sibling: an A1 $idHash$ commits $(eNote, m_{\text{issuer}}, \sigma)$, so the tie runs through the note's own $eNote$, with the spend's public `face` pinning the plaintext (and hence the addressed identity). `NoteBindingVerifierAdapter` derives all public inputs on chain (25 for A2; 26 for A1, `face` at index

   (a) from the caller's `nullifier` / `face` / `eEnc` / `dc.P_I`; the

   governance slot `Notes.setNoteBindingVerifier` rotates the verifier. (*Proofs*, Theorem 12; toolchain history in *alberta-buck-verifier-implementation*.)

With all four pieces shipped, every mint and spend is Identity-M-bound, the on-chain `identityRoot` gates every spend, and the note's counterparty Identity is always recoverable and – because the membership is *bound* to the sigma's committed point, and the verified ciphertext is *bound to the spent note* – always nameable, even under collusion and ciphertext substitution. The only remaining pre-deployment item is a proper Powers-of-Tau ceremony (the dev ptaus, including the pot22 behind the note-binding verifier, use dev-only entropy).

Additional items tracked across the series:

- **Mint-side deferred-approve handshake (shipped).** The public-issuer Schnorr binding (`verifyIssuerSchnorr`) and the private-issuer re-encryption binding (`verifyIssuerReenc`) are deployed and tested. Every mint carries a verified issuer Identity binding.

- **Variable-`N` batch size (resolved via pinned set).** The wallet rounds any requested split up to the smallest pinned N >= N_requested from the deployed set (recommended {16, 128, 1024}), padding with `v = 0` dummy openings. Each pinned N has its own verifier contract; `MintVerifierAdapter` dispatches on `cms.length`.

- **Prover contention (resolved operationally, not protocolly).** Batch minting makes each mint a rollup-style transaction with a stale-state guard on `oldRoot` and `nextLeafIndex`. Losers in a race re-prove; no BUCK is lost. Initial deployment uses wallet-side optimistic retry; a sequencer/aggregator is deferred until real-world contention warrants it.

17 Privacy Audit: Data Revealed Per Step

The whole construction threads one needle: publish enough that each party can *name* the other under compulsion, while publishing so little that no *third party* can name either, or link payments in bulk. This section consolidates the per-step disclosure analysis for the shipped Identity-M surface.

17.1 Requirements and adversaries

Three properties, against three adversaries:

- **(Bulk-tracing resistance.)** *Mallory*, a passive observer with the full chain history and unbounded compute but no party's secrets, cannot reconstruct who paid whom, how much, or link a mint to its spend(s).
- **(Third-party non-disclosure.)** Nothing on chain reveals either party's KYC-bound Identity point M to a non-participant – not even to the *counterparty's* counterparty, and not in aggregate across many payments.
- **(Mutual decryptability.)** Each of the two participants, from data they hold plus public chain state, can produce a sound receipt naming the other's registered M – and a *colluding* pair cannot arrange a payment for which no such receipt exists (the non-deniable-receipt invariant).

The first two are *confidentiality* against outsiders; the third is *accountability* between insiders. They pull in opposite directions, so every published datum below is justified against all three.

17.2 The on-chain footprint of each step

Step	Published on chain
`register` / `bindContract`	addr, `pk`, `E_addr` (ElGamal ct of M under own `pk`), `isPublicIdentity`, `identityRoot`
`mint` (any flavor)	`Minted(issuer, totalFace, startIndex, count, newRoot)`; `cms[]` in calldata (opaque Poseidon)
`mint` public (A1/B1)	+ `IssuerBound(issuer, newRoot)`; one Schnorr over `keccak(cms)`
`mint` private (A2)	+ `IssuerReencBound(issuer, newRoot, count)`; per-leaf blinded binding $(Q, U, \hat{T}, A_1..A_5)$
A1/A2 spend	`SpentCoupledA1/A2(nf, face, recipient, P_I)`; $eEnc$ (fresh re-encryption), the coupling sigma, the membership
B1 spend	`SpentCoupledB1(nf, face, recipient, P_{dep})`; $eDepForIss$, the depositor-binding sigma, the membership proo

Two facts do the heavy lifting throughout. First, `cms[]` are Poseidon commitments: a leaf reveals *nothing* about $(flavor, v, \rho, idHash, predicate)$ without ρ. Second, the spend nullifier $nf = \text{Poseidon}_3(\rho, idHash, tag)$ is a PRF output over secret material, so it is unlinkable to any `cm` without the opening (*Proofs*, Theorem 10).

17.3 Why bulk-tracing and third-party privacy hold

Mint is opaque; spend is unlinkable to mint. At mint Mallory sees opaque `cms[]` and a `totalFace`; she cannot read any v_i, recipient, or flavor (Poseidon hiding). At spend she sees a fresh nf, a `face`, and a `recipient`; nf is a PRF over $(\rho, idHash)$, so inverting it to a specific `cm` is pre-image resistance. No field of a spend points back to a mint: the spend's $eEnc$ is a *fresh re-encryption* of the leaf ciphertext (uniformly re-randomised; the note-binding relation deliberately proves re-encryption *equivalence*, never equality with the public mint-time $eIss$). The only residual linkage surfaces are the two that *must* be public for the ERC-20 to pay out – `face` and `recipient` – mitigated by fixed denominations and relayed payouts, orthogonal to this document.

Identities never appear on chain – the load-bearing check. A registered E_{addr} is an ElGamal *ciphertext* of M; reading it yields nothing without sk. The danger is in the *bindings*, which manipulate M and the recipient point algebraically. Concretely, the A2 mint binding must publish enough to prove "$eIss$ re-encrypts the issuer's registered M_{iss} under the recipient's point" without revealing either. This is exactly where the first draft failed – the canonical near-miss, kept here because it is the clearest way to see the invariant:

```
Near-miss (rejected draft).  The binding published T = r' * pk_rec.
But the leaf is C_i = M_iss + r' * pk_rec, so ANY observer computes
    M_iss = C_i - T.
At mint msg.sender IS the issuer, so this publicly bound the issuer's
address to its Identity -- collapsing A2 to A1 and breaking BOTH
third-party non-disclosure and bulk unlinkability (every A2 mint by the
same issuer now carries the same recoverable M_iss).
```

The shipped binding blinds it: it publishes $\hat{T} = r'pk_{rec} + \gamma G$ for a fresh γ. Now $C_i - \hat{T} = M_{iss} - \gamma G$, a uniformly random offset from M_{iss} – so $\hat{T}$ reveals nothing. The recipient's point is likewise hidden (inside the perfectly-hiding Q and the uniform $U, \hat{T}$), and soundness is undisturbed because γ cancels in the verification relations. An A2 mint's on-chain footprint is, to Mallory, a vector of uniformly-distributed group elements.

The spend-side sigmas reveal the spender's action, not any Identity. The deposit-coupling and depositor-binding sigmas are HVZK; their only Identity-derived public values are the blinded commitments $P_I = M_I + bH$ / $P_{dep} = M_{dep} + bH$, perfectly hiding for uniform b. The membership and note-binding Groth16 proofs are zero-knowledge; their public inputs are the nullifier (PRF), $eEnc$ (uniform), and P (hiding). `msg.sender` is visible – exactly as the sender of any ERC-20 transfer is – but no *Identity* is.

17.4 Why mutual decryptability holds – the final data set

For each flavor, the data each party ends up holding, and the predicate it satisfies (the receipt verifiers in `alberta_buck.wallet` are the public predicate RcptVerify; see the executable sections above):

Flavor	Recipient names issuer via	Issuer names depositor via
B1	in-clear M_{iss} + batch Schnorr (`verifyIssuerSchnorr`)	`SpentCoupledB1`: decrypt $eDepFo$
A1	in-clear M_{iss} + batch Schnorr	chose M_{rec} at mint (symmetric kr
A2	decrypt $eIss$ under m_{rec}; binding + membership prove it is the **registered** minter (A2 in Code)	chose M_{rec} at mint

The A2 row is the subtle one. The recipient's receipt composes three facts: the mint anti-framing binding (the ciphertext re-encrypts the issuer's *registered* record), their own verifiable decryption (it opens under m_{rec} to the named point), and the membership of that point in $\mathsf{rt}_{\mathsf{id}}$. No issuer cooperation is needed – the receipt is *unilateral* – and no new proof is needed at receipt time: the coupling falls out of composing the mint binding with the recipient's own decryption.

17.5 The colluding pair, and where each escape is closed

Completeness (an honest recipient *can* produce a receipt) is met off chain. The third clause – no deniability even when payer and payee *collude* – needs the binding to be *unavoidable*. Three escapes existed on the road here, each now closed at a specific gate:

1. *A binding-less leaf.* Closed at **mint**: `Notes.mint` requires a verified issuer binding (Schnorr or re-encryption) for every leaf; the SNARK's per-leaf `issuerMode` signal makes a private-issuer bearer leaf unprovable (no B2 codepoint).

2. *A nameable note deposited by the wrong Identity.* Closed at **spend** by the coupling sigma + bound membership: the depositing account must hold the Identity scalar that decrypts the verified ciphertext.

3. *An un-nameable note made spendable by ciphertext substitution.* Closed at **spend** by the note-binding tie: the verified *eEnc* must be keyed to the identity material committed behind *this spend's nullifier* (A2: re-encrypt the committed $eIss$; A1: share the $eNote$'s M_{rec}, the public `face` pinning the plaintext) (*Proofs*, Theorem 12) – demonstrated executably in the A2 and A1 in Code counter-examples above.

18 Security Properties and the Tests That Witness Them

Every property claimed in this document is pinned by an executable witness – the Python suites exercise the wallet reference, the Forge suites the on-chain verifiers, and the cross-artifact vectors (`alberta_buck.wallet` → `vm.readFile`) keep the two byte-for-byte aligned.

Suite	Count	What it demonstrates
`alberta_buck/test/test_unilateral_a2.py`	11	A2 mint binding; delivery decrypt; deposit coupling (any account); unilateral receipt na
`alberta_buck/test/test_unilateral_a1.py`	7	A1 mint; `idHash` binds the public issuer + Schnorr; recipient-side receipt names both; t
`alberta_buck/test/test_b1_binding.py`	11	B1 depositor binding; issuer-unilateral receipt names the depositor; collusion-substituti
`test/UnilateralA2.t.sol`	7	on-chain `verifyDepositCoupling` parity + soundness (tampered response / P_I / ciphe
`test/B1Binding.t.sol`	6	on-chain `verifyDepositorBinding` parity + soundness
`test/IdentityMembershipBinding.t.sol`	5	the P binding: bound point verifies; mismatched P_x/P_y/root rejected
`test/NotesCoupledA1.t.sol`	8	end-to-end `spendCoupledA1`: completeness + payout; tamper/double-spend reverts
`test/NotesCoupledA2.t.sol`	10	end-to-end `spendCoupledA2`: completeness + payout + `SpentCoupledA2`; bad coupling /
`test/NotesCoupledB1.t.sol`	6	end-to-end `spendCoupledB1`: the depositor-binding + bound membership path
`test/NoteBindingVerifier.t.sol`	18	the note ↔ $eEnc$ tie, both layouts: real Groth16 proofs verify (raw + adapter); tampe
`test/NotesE2E.t.sol`	9	the full lifecycle per flavor with EVERY verifier real (batch mint, spend, sigma, membe

Build/test entry points: `make nix-test` (Forge), `make nix-test-python` (wallet reference), `make nix-snark-note-binding` / `nix-snark-note-binding-a1` (note-binding circuits + trusted setups + vectors), `make nix-snark-e2e-fixtures` (the all-real-verifier lifecycle fixtures).

19 Cross-References

- alberta-buck-notes.org – architectural overview of the Notes primitive (flavors, Merkle tree, demurrage interaction). Identity-M edition. The other living half of the Notes documentation pair (this document is the worked, executable half); its *Document Series* section is the canonical map of which satellite memos are living, forward designs, or historical.
- alberta-buck-proofs.org – formal correctness arguments (Theorems 7-12; Open Questions in Part IV).
- *alberta-buck-verifier-implementation.org* – SNARK toolchain engineering: the witness-generation root cause, rapidsnark integration, trusted setups.
- alberta-buck-ethereum.org – Solidity surface, gas costs, phase-by-phase implementation status, Open Work Items.
- alberta-buck-identity.org – IdentityRegistry, PS credentials, and the bilateral ElGamal handshake that anchors `msg.sender` to a verifiable identity.
- The unified identity-M / "one gadget" / Registry-Identity accumulator design (commit-before-use) is now in alberta-buck-notes.org ("Mutual Decryptability", "one gadget") and here in "The Identity-M Spend Path" + accumulator sections. (Superseded the original memo.)
- alberta-buck-notes-unilateral.org – *historical*: the A2 corner as first built; absorbed into "A2 Identity-Targeted Spend" / "A2 in Code" above.
- Non-deniable-receipt invariant: see alberta-buck-notes.org "The Non-Deniable-Receipt Invariant"; per-step privacy audit and "EOA Transfer Mirror" are above in this document; theorems in alberta-buck-proofs.org. The mint-side issuer bindings and note <-> eEnc tie are implemented via note_binding circuits (see "A2 Identity-Targeted Spend" etc.).
- `circuits/mint_batch.circom` – the batch-mint circuit (one compile per pinned N).
- `circuits/spend.circom` – the flavor-agnostic spend (note-proof) circuit.
- `circuits/note_binding.circom`, `circuits/note_binding_a1.circom` – the note $\leftrightarrow$ *eEnc* tie (A2 and A1 payload layouts).
- `src/Notes.sol`, `src/MintGroth16Verifier*.sol`, `src/MintVerifierAdapter.sol`, `src/SpendGroth16Veri` `src/SpendVerifierAdapter.sol`, `src/NoteBindingGroth16Verifier.sol`, `src/NoteBindingA1Groth16Ver` `src/NoteBindingVerifierAdapter.sol` – the on-chain code paths walked above.
- `scripts/snark/prove_mint_batch.js`, `scripts/snark/prove_spend.js` – the reference provers used in tests; `scripts/snark/setup_note_binding.sh`, `scripts/snark/setup_note_binding_a1.sh` – the note-binding circuits + trusted-setup pipelines.
- `test/MintVerifier.t.sol`, `test/SpendVerifier.t.sol`, `test/NoteBindingVerifier.t.sol` – the Foundry tests that pin the soundness stories to executable assertions.